Parlez-nous de l'expérience de téléchargement de PDF.

Documentation C#

Apprenez à écrire une application à l'aide du langage de programmation C# sur la plateforme .NET.

Apprendre à programmer en C#

BIEN DÉMARRER

Découvrir le C# | Tutoriels, cours, vidéos et bien plus encore

VIDEO

Série de vidéos pour débutants C#

Flux débutant C#

Série de vidéos pour intermédiaires C#

DIDACTICIEL

Tutoriels auto-guidés

Tutoriel dans le navigateur

RÉFÉRENCE

C# sur Q&A

Langues sur les forums de la communauté technique .NET

C# sur Stack Overflow

C# sur Discord

Notions de base C#

VUE D'ENSEMBLE

Visite guidée de C#

À l'intérieur d'un programme C#

Série de vidéos sur les points clés du C#

CONCEPT

Système des types

Programmation orientée objet

Techniques fonctionnelles

Exceptions

Style de codage

DIDACTICIEL

Afficher la ligne de commande

Introduction aux classes

C# orienté objet

Conversion de types

Critères spéciaux

Utiliser LINQ pour interroger les données

Concepts clés

VUE D'ENSEMBLE

Concepts de programmation

DÉMARRAGE RAPIDE

Méthodes

Propriétés

Indexeurs

Iterators

Délégués

Événements

CONCEPT

Types références Nullables

Migrations de référence pouvant accepter la valeur Null

Résoudre les avertissements pouvant accepter la valeur Null

LINQ (Language-Integrated Query)

Gestion de version

Nouveautés

NOUVEAUTÉS

Nouveautés de la version 11 de C#

Nouveautés dans C# 10

Nouveautés dans C# 9.0

Nouveautés de C# 8.0

DIDACTICIEL

Explorer les types d'enregistrement

Explorer les instructions de niveau supérieur

Explorez les nouveaux modèles

Mettre à jour les interfaces en toute sécurité

Créer des mixins avec des interfaces

Explorer des index et des plages

Types références Nullables

Explorer les flux asynchrones

Écrire un gestionnaire d'interpolation de chaîne personnalisé

RÉFÉRENCE

Changements cassants du compilateur C#

Compatibilité des versions

Informations de référence sur le langage C#

RÉFÉRENCE

Informations de référence sur le langage

Mots clés C#

Opérateurs C#

Configurer la version de langage

Spécification du langage C# - Brouillon C# 7 en cours

Rester en contact

RÉFÉRENCE

Communauté des développeurs .NET

YouTube

Twitter

Visite guidée du langage C#

Article • 25/02/2023

C# (prononcez « Si Sharp ») est un langage de programmation moderne, orienté objet et de type sécurisé. C# permet aux développeurs de créer de nombreux types d'applications sécurisées et robustes qui s'exécutent dans .NET. C# prend sa source dans la famille de langages C et sera immédiatement reconnaissable aux programmeurs en C, C++, Java et JavaScript. Cette visite guidée offre une vue d'ensemble des principaux composants du langage dans C# 8 et antérieur. Si vous voulez explorer le langage à travers des exemples interactifs, essayez les tutoriels de présentation de C#.

C# est un langage de programmation orienté objet, ***orienté composant***. C# fournit des constructions de langage qui prennent directement en charge ces concepts, ce qui fait de C# un langage naturel pour créer et utiliser des composants logiciels. Depuis son apparition, C# a ajouté des fonctionnalités pour prendre en charge de nouvelles charges de travail et des pratiques de conception logicielle émergentes. À la base, C# est un langage ***orienté objet***. Vous définissez des types et leur comportement.

Plusieurs fonctionnalités C# permettent de créer des applications robustes et durables. ***Garbage Collection*** récupère automatiquement la mémoire occupée par des objets inutilisés inaccessibles. Les ***types nullables*** protègent contre les variables qui ne font pas référence à des objets alloués. La ***gestion des exceptions*** offre une approche structurée et extensible de la détection et de la récupération des erreurs. Les ***expressions lambda*** prennent en charge les techniques de programmation fonctionnelle. La syntaxe ***LINQ (Language Integrated Query)*** crée un modèle commun pour utiliser les données de n'importe quelle source. La prise en charge du langage pour les ***opérations asynchrones*** fournit une syntaxe pour créer des systèmes distribués. C# a un ***système de type unifié***. Tous les types C#, y compris les types primitifs tels que `int` et `double`, héritent d'un seul type `object` racine. Tous les types partagent un ensemble d'opérations communes. Les valeurs de n'importe quel type peuvent être stockées, transportées et manipulées de manière cohérente. De plus, C# prend en charge à la fois les types référence et les types valeur définis par l'utilisateur. C# autorise l'allocation dynamique d'objets et le stockage in-line de structures légères. C# prend en charge les méthodes et les types génériques, qui offrent une cohérence des types et des performances accrues. C# fournit des itérateurs qui permettent aux implémenteurs de classes de collection de définir des comportements personnalisés pour le code client.

C# met l'accent sur la ***gestion de versions*** pour garantir que les programmes et les bibliothèques peuvent évoluer de manière compatible. Certains aspects de la conception de C# ont été directement influencés par la gestion de versions, notamment

les modificateurs `virtual` et `override` distincts, les règles de résolution des surcharges de méthodes et la prise en charge des déclarations explicites de membres d'interface.

Architecture .NET

Les programmes C# s'exécutent sur .NET, un système d'exécution virtuel appelé common language runtime (CLR) et un ensemble de bibliothèques de classes. Le CLR est l'implémentation de Microsoft de la CLI (Common Language Infrastructure), qui est une norme internationale. La CLI est la base de la création d'environnements d'exécution et de développement où langages et bibliothèques fonctionnent ensemble de façon fluide.

Le code source écrit en C# est compilé dans un langage intermédiaire (IL) qui est conforme à la spécification CLI. Le code et les ressources IL, comme les bitmaps et les chaînes, sont stockés dans un assembly, en général avec une extension *.dll*. Un assembly contient un manifeste qui fournit des informations sur ses types, sa version et sa culture.

Lorsque le programme C# est exécuté, l'assembly est chargé dans le CLR. Le CLR effectue une compilation juste-à-temps (JIT, Just-In-Time) pour convertir le code IL en instructions machine natives. Le CLR fournit d'autres services liés au Garbage collection, à la gestion des exceptions et à la gestion des ressources. Le code exécuté par le CLR est parfois appelé « code managé ». Le « code non managé » est compilé dans le langage machine natif qui cible une plateforme spécifique.

L'interopérabilité des langages est une fonctionnalité clé de .NET. Le code IL produit par le compilateur C# est conforme à la spécification CTS (Common Type Specification). Le code IL généré à partir de C# peut interagir avec le code généré à partir des versions .NET de F#, Visual Basic et C++. Il existe plus de 20 autres langages conformes à la spécification CTS. Un seul assembly peut contenir plusieurs modules écrits dans différents langages .NET. Les types peuvent faire référence les uns aux autres comme s'ils avaient été écrits dans le même langage.

En plus des services d'exécution, .NET comprend également des bibliothèques complètes. Ces bibliothèques prennent en charge un grand nombre de charges de travail différentes. Elles sont organisées en espaces de noms qui fournissent un large éventail de fonctionnalités utiles. Les bibliothèques englobent tout, de l'entrée/la sortie des fichiers à la manipulation de chaînes et à l'analyse XML, en passant par les frameworks d'application web et les contrôles Windows Forms. Une application C# typique utilise beaucoup la bibliothèque de classes .NET pour gérer les tâches fastidieuses courantes.

Pour plus d'informations sur .NET, consultez Vue d'ensemble de .NET.

Hello World

Le programme « Hello, World » est souvent utilisé pour présenter un langage de programmation. Le voici en C# :

```C#
using System;

class Hello
{
    static void Main()
    {
        Console.WriteLine("Hello, World");
    }
}
```

Le programme « Hello, World » commence par une directive `using` qui fait référence à l'espace de noms `System`. Les espaces de noms représentent un moyen hiérarchique d'organiser les bibliothèques et les programmes C#. Les espaces de noms contiennent des types et d'autres espaces de noms ; par exemple, l'espace de noms `System` contient plusieurs types, notamment la classe `Console` référencée dans le programme, et d'autres espaces de noms, tels que `IO` et `Collections`. Une directive `using` qui fait référence à un espace de noms donné permet l'utilisation non qualifiée des types membres de cet espace de noms. En raison de la directive `using`, le programme peut utiliser `Console.WriteLine` comme raccourci pour `System.Console.WriteLine`.

La classe `Hello` déclarée par le programme « Hello, World » a un membre unique, la méthode nommée `Main`. La méthode `Main` est déclarée avec le modificateur `static`. Si les méthodes d'instance peuvent faire référence à une instance d'objet englobante particulière avec le mot clé `this`, les méthodes statiques fonctionnent sans référence à un objet particulier. Par convention, une méthode statique nommée `Main` sert de point d'entrée d'un programme C#.

La sortie du programme est générée par la méthode `WriteLine` de la classe `Console` dans l'espace de noms `System`. Cette classe est fournie par les bibliothèques de classes standard, qui, par défaut, sont référencées automatiquement par le compilateur.

Types et variables

Un *type* définit la structure et le comportement de toutes les données dans C#. La déclaration d'un type peut inclure ses membres, le type de base, les interfaces qu'il

implémente et les opérations autorisées pour ce type. Une *variable* est un nom qui fait référence à une instance d'un type spécifique.

Il existe deux genres de types en C# : les *types référence* et les *types valeur*. Les variables des types valeur contiennent directement leurs données. Les variables des types référence stockent des références à leurs données, connues sous le nom d'objets. Avec les types référence, deux variables peuvent faire référence au même objet et les opérations sur une variable peuvent affecter l'objet référencé par l'autre variable. Avec les types valeur, les variables ont chacune leur propre copie de données, et les opérations sur une variable ne peuvent pas affecter l'autre (sauf pour les variables de paramètre `ref` et `out`).

Un ***identificateur*** est un nom de variable. Un identificateur est une séquence de caractères Unicode sans espace blanc. Un identificateur peut être un mot réservé C#, s'il est préfixé par `@`. L'utilisation d'un mot réservé comme identificateur peut être utile lors d'une interaction avec d'autres langages.

Les types valeur de C# sont divisés entre *types simples*, *types enum*, *types struct*, *types valeur nullable* et *types valeur tuple*. Les types référence de C# sont divisés entre *types de classe*, *types d'interface*, *types de tableau* et *types de délégué*.

La structure suivante offre une vue d'ensemble du système de types de C#.

- Types de valeur
  - Types simples
    - Intégral signé : `sbyte`, `short`, `int`, `long`
    - Intégral non signé : `byte`, `ushort`, `uint`, `ulong`
    - Caractères Unicode : `char`, qui représente une unité de code UTF-16
    - Nombre binaire à virgule flottante IEEE : `float`, `double`
    - Nombre décimal à virgule flottante haute précision : `decimal`
    - Booléen : `bool`, qui représente les valeurs booléennes. Les valeurs sont `true` ou `false`
  - Types d'enum
    - Types définis par l'utilisateur de la forme `enum E {...}` Un type `enum` est un type distinct avec des constantes nommées. Chaque type `enum` a un type sous-jacent qui doit être un des huit types intégraux. L'ensemble de valeurs d'un type `enum` est le même que l'ensemble de valeurs du type sous-jacent.
  - Types struct
    - Types définis par l'utilisateur de la forme `struct S {...}`
  - Types valeur Nullable
    - Extensions de tous les autres types de valeurs avec une valeur `null`

    - Types valeur tuple
        - Types définis par l'utilisateur de la forme `(T1, T2, ...)`
- Types référence
    - Types de classes
        - Classe de base fondamentale de tous les autres types : `object`
        - Chaînes Unicode : `string`, qui représente une séquence d'unités de code UTF-16
        - Types définis par l'utilisateur de la forme `class C {...}`
    - Types d'interface
        - Types définis par l'utilisateur de la forme `interface I {...}`
    - Types de tableaux
        - Unidimensionnel, multidimensionnel et en escalier. Par exemple : `int[]`, `int[,]` et `int[][]`
    - Types délégués
        - Types définis par l'utilisateur de la forme `delegate int D(...)`

Les programmes C# utilisent les *déclarations de type* pour créer de nouveaux types. Une déclaration de type spécifie le nom et les membres du nouveau type. Six catégories de types C# sont définissables par l'utilisateur : les types de classe, les types struct, les types d'interface, les types enum, les types de délégué et les types valeur tuple. Vous pouvez également déclarer des types `record`, soit `record struct`, soit `record class`. Les types d'enregistrement (record) ont des membres synthétisés par le compilateur. Vous utilisez les enregistrements principalement pour stocker des valeurs, avec un comportement associé minimal.

- Un type `class` définit une structure de données qui contient des données membres (champs) et des fonctions membres (méthodes, propriétés, etc.). Les types de classes prennent en charge l'héritage unique et le polymorphisme, des mécanismes par lesquels les classes dérivées peuvent étendre et spécialiser les classes de base.
- Un type `struct` est similaire à un type de classe dans la mesure où il représente une structure avec des membres de données et des membres de fonctions. Cependant, contrairement aux classes, les structs sont des types valeur et ne nécessitent généralement pas d'allocation de tas. Les types struct ne prennent pas en charge l'héritage spécifié par l'utilisateur, et tous les types struct héritent implicitement du type `object`.
- Un type `interface` définit un contrat en tant que jeu nommé de membres publics. Une `class` ou un `struct` qui implémente une `interface` doivent fournir les implémentations des membres de l'interface. Une `interface` peut hériter de

plusieurs interfaces de base, et une `class` ou `struct` peut implémenter plusieurs interfaces.
- Un type `delegate` représente des références aux méthodes avec une liste de paramètres et un type de retour particuliers. Les délégués permettent de traiter les méthodes en tant qu'entités qui peuvent être affectées à des variables et passées comme paramètres. Les délégués sont semblables aux types de fonction fournis par les langages fonctionnels. Ils sont aussi similaires au concept de pointeurs de fonction que l'on trouve dans d'autres langages. Contrairement aux pointeurs de fonction, les délégués sont orientés objet et de type sécurisé.

Les types `class`, `struct`, `interface` et `delegate` prennent tous en charge les génériques, ce qui leur permet d'être paramétrés avec d'autres types.

C# prend en charge les tableaux unidimensionnels et multidimensionnels de tout type. Contrairement aux types mentionnés ci-dessus, les types de tableau n'ont pas à être déclarés avant de pouvoir être utilisés. Au lieu de cela, les types de tableaux sont construits en ajoutant des crochets à un nom de type. Par exemple, `int[]` est un tableau unidimensionnel de `int`, `int[,]` est un tableau bidimensionnel de `int` et `int[][]` est un tableau unidimensionnel de tableaux unidimensionnels, ou un tableau « en escalier », de `int`.

Les types nullables n'ont pas besoin de définition distincte. Pour chaque type `T` non-nullable, il existe un type `T?` nullable correspondant, qui peut contenir une valeur supplémentaire, `null`. Par exemple, `int?` est un type qui peut contenir n'importe quel entier 32 bits ou la valeur `null`, tandis que `string?` est un type qui peut contenir n'importe quelle `string` ou la valeur `null`.

Le système de types de C# est unifié afin qu'une valeur de n'importe quel type puisse être traitée comme `object`. Chaque type dans C# dérive directement ou indirectement du type `object`, et `object` est la classe de base fondamentale de tous les types. Les valeurs des types référence sont considérées comme des objets simplement en affichant les valeurs en tant que type `object`. Les valeurs des types valeur sont considérées comme des objets en effectuant des opérations de *boxing* et *d'unboxing*. Dans l'exemple suivant, une valeur `int` est convertie en `object` et à nouveau en `int`.

C#

```
int i = 123;
object o = i;    // Boxing
int j = (int)o;  // Unboxing
```

Quand une valeur d'un type valeur est affectée à une référence `object`, une « box » est alloué pour contenir la valeur. Cette « box » est une instance d'un type référence et la valeur est copiée dans celle-ci. À l'inverse, lorsqu'une référence `object` est castée en type valeur, un contrôle est effectué pour s'assurer que l'`object` référencé est une « box » du type valeur correct. Si le contrôle réussit, la valeur dans la « box » est copiée dans le type valeur.

Le système de types unifié de *C#* signifie en effet que les types valeur sont traités comme références `object` « à la demande ». En raison de l'unification, les bibliothèques à usage général qui se servent du type `object` peuvent être utilisées avec tous les types qui dérivent de `object`, y compris les types référence et les types valeur.

Il existe plusieurs types de *variables* en C#, y compris les champs, les éléments de tableau, les variables locales et les paramètres. Les variables représentent des emplacements de stockage. Chaque variable a un type qui détermine quelles valeurs peuvent y être stockées, comme illustré ci-dessous.

- Type de valeur n'acceptant pas Null
  - Une valeur de ce type exact
- Types valeur Nullable
  - Une valeur `null` ou une valeur de ce type exact
- object
  - Une référence `null`, une référence à un objet de tout type référence ou une référence à une valeur boxed de n'importe quel type valeur
- Type de classe
  - Une référence `null`, une référence à une instance de ce type de classe ou une référence à une instance d'une classe dérivée de ce type de classe
- Type d'interface
  - Une référence `null`, une référence à une instance d'un type de classe qui implémente ce type d'interface ou une référence à une valeur boxed d'un type valeur qui implémente ce type d'interface
- Type tableau
  - Une référence `null`, une référence à une instance de ce type de tableau ou une instance d'un type de tableau compatible
- Type délégué
  - Une référence `null` ou une référence à une instance d'un type délégué compatible

Structure du programme

Les concepts organisationnels clés en C# sont les ***programmes***, les ***espaces de noms***, les ***types***, les ***membres*** et les ***assemblys***. Les programmes déclarent des types qui contiennent des membres et peuvent être organisés en espaces de noms. Les classes, les structs et les interfaces sont des exemples de types. Les champs, méthodes, propriétés et événements sont des exemples de membres. Lorsque les programmes C# sont compilés, ils sont physiquement empaquetés dans des assemblys. Les assemblys ont généralement l'extension de fichier `.exe` ou `.dll`, selon qu'elles implémentent des ***applications*** ou des ***bibliothèques***, respectivement.

Voici un petit exemple d'un assembly qui contient le code suivant :

C#

```
namespace Acme.Collections;

public class Stack<T>
{
    Entry _top;

    public void Push(T data)
    {
        _top = new Entry(_top, data);
    }

    public T Pop()
    {
        if (_top == null)
        {
            throw new InvalidOperationException();
        }
        T result = _top.Data;
        _top = _top.Next;

        return result;
    }

    class Entry
    {
        public Entry Next { get; set; }
        public T Data { get; set; }

        public Entry(Entry next, T data)
        {
            Next = next;
            Data = data;
        }
    }
}
```

Le nom qualifié complet de cette classe est `Acme.Collections.Stack`. La classe contient plusieurs membres : un champ nommé `_top`, deux méthodes nommées `Push` et `Pop`, et une classe imbriquée nommée `Entry`. La classe `Entry` contient trois membres en plus : une propriété nommée `Next`, une propriété nommée `Data` et un constructeur. `Stack` est une classe *générique*. Il a un paramètre de type, `T`, qui est remplacé par un type concret lorsqu'il est utilisé.

Un *stack* est une collection FILO (« first in - last out »). De nouveaux éléments sont ajoutés en haut du stack. Lorsqu'un élément est supprimé, il est supprimé du haut du stack. L'exemple précédent déclare le type `Stack` qui définit le stockage et le comportement d'un stack. Vous pouvez déclarer une variable qui fait référence à une instance du type `Stack` pour utiliser cette fonctionnalité.

Les assemblys contiennent le code exécutable sous forme d'instructions de langage intermédiaire (IL) et des informations symboliques sous la forme de métadonnées. Avant son exécution, le compilateur juste-à-temps (JIT) du Common Language Runtime .NET convertit le code IL d'un assembly en code spécifique au processeur.

Comme un assembly est une unité de fonctionnalité autodescriptive contenant du code et des métadonnées, les directives `#include` et les fichiers d'en-tête ne sont pas nécessaires en C#. Les membres et types publics contenus dans un assembly particulier sont disponibles dans un programme C# par simple référence à cet assembly lors de la compilation du programme. Par exemple, ce programme utilise la classe `Acme.Collections.Stack` à partir de l'assembly `acme.dll` :

C#

```
class Example
{
    public static void Main()
    {
        var s = new Acme.Collections.Stack<int>();
        s.Push(1); // stack contains 1
        s.Push(10); // stack contains 1, 10
        s.Push(100); // stack contains 1, 10, 100
        Console.WriteLine(s.Pop()); // stack contains 1, 10
        Console.WriteLine(s.Pop()); // stack contains 1
        Console.WriteLine(s.Pop()); // stack is empty
    }
}
```

Pour compiler ce programme, vous devez *référencer* l'assembly contenant la classe stack définie dans l'exemple précédent.

Les programmes C# peuvent être stockés dans plusieurs fichiers sources. Lorsqu'un programme C# est compilé, tous les fichiers sources sont traités ensemble et peuvent librement se référencer entre eux. Sur le plan conceptuel, c'est comme si tous les fichiers sources étaient concaténés dans un seul grand fichier avant d'être traités. Les déclarations anticipées ne sont jamais nécessaires en C#, car, à de rares exceptions près, l'ordre de déclaration n'a pas d'importance. C# ne limite pas un fichier source à la déclaration d'un seul type public et n'a pas besoin non plus que le nom du fichier source corresponde à un type déclaré dans ce fichier.

D'autres articles de cette visite guidée expliquent ces blocs organisationnels.

Next

Types C# et membres

Article • 01/06/2023

En tant que langage orienté objet, C# prend en charge les concepts d'encapsulation, d'héritage et de polymorphisme. Une classe peut hériter directement d'une classe parent et elle peut implémenter un nombre quelconque d'interfaces. Les méthodes qui substituent des méthodes virtuelles dans une classe parente requièrent le mot-clé `override` pour éviter toute redéfinition accidentelle. En C#, un struct est comme une classe légère : il s'agit d'un type alloué par la pile qui peut implémenter des interfaces mais ne prend pas en charge l'héritage. C# fournit des types `record class` et `record struct`, qui sont des types dont l'objectif principal est de stocker des valeurs de données.

Tous les types sont initialisés par le biais d'un *constructeur*, une méthode responsable de l'initialisation d'une instance. Deux déclarations de constructeur ont un comportement unique :

- *Constructeur sans paramètre*, qui initialise tous les champs à leur valeur par défaut.
- *Constructeur principal*, qui déclare les paramètres requis pour une instance de ce type.

Classes et objets

Les *classes* représentent le type le plus fondamental de C#. Une classe est une structure de données qui combine l'état (champs) et les actions (méthodes et autres fonctions membres) en une seule unité. Une classe fournit une définition pour les *instances* de la classe, également appelées *objets*. Les classes prennent en charge *l'héritage* et le *polymorphisme*, des mécanismes par lesquels les *classes dérivées* peuvent étendre et spécialiser les *classes de base*.

Les nouvelles classes sont créées à l'aide des déclarations de classe. Une déclaration de classe commence par un en-tête. L'en-tête spécifie :

- Les attributs et les modificateurs de la classe
- Le nom de la classe
- La classe de base (lors de l'héritage d'une classe de base)
- Les interfaces implémentées par la classe.

L'en-tête est suivi par le corps de la classe, qui se compose d'une liste de déclarations de membres écrites entre les délimiteurs `{` et `}`.

Le code suivant indique une déclaration d'une classe simple nommée `Point` :

```C#
public class Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y) => (X, Y) = (x, y);
}
```

Les instances de classes sont créées à l'aide de l'opérateur `new`, qui alloue la mémoire pour une nouvelle instance, appelle un constructeur pour initialiser l'instance et retourne une référence à l'instance. Les instructions suivantes créent deux objets `Point` et stockent les références à ces objets dans les deux variables :

```C#
var p1 = new Point(0, 0);
var p2 = new Point(10, 20);
```

La mémoire occupée par un objet est automatiquement libérée lorsque l'objet n'est plus accessible. Il n'est ni possible, ni nécessaire de libérer explicitement des objets dans C#.


```C#
var p1 = new Point(0, 0);
var p2 = new Point(10, 20);
```


Les applications ou les tests pour les algorithmes peuvent avoir besoin de créer plusieurs objets `Point`. La classe suivante génère une séquence de points aléatoires. Le nombre de points est défini par le paramètre du *constructeur principal*. Le paramètre `numPoints` du constructeur principal est dans l'étendue de tous les membres de la classe :

```C#
public class PointFactory(int numberOfPoints)
{
    public IEnumerable<Point> CreatePoints()
    {
        var generator = new Random();
        for (int i = 0; i < numberOfPoints; i++)
        {
            yield return new Point(generator.Next(), generator.Next());
```

```
        }
    }
}
```

Vous pouvez utiliser la classe comme indiqué dans le code suivant :

C#

```
var factory = new PointFactory(10);
foreach (var point in factory.CreatePoints())
{
    Console.WriteLine($"({point.X}, {point.Y})");
}
```

Paramètres de type

Les classes génériques définissent les ***paramètres de type***. Les paramètres de type sont une liste de noms de paramètres de type placés entre crochets. Les paramètres de type suivent le nom de la classe. Les paramètres de type peuvent ensuite être utilisés dans le corps des déclarations de classe pour définir les membres de la classe. Dans l'exemple suivant, les paramètres de type de `Pair` sont `TFirst` et `TSecond` :

C#

```
public class Pair<TFirst, TSecond>
{
    public TFirst First { get; }
    public TSecond Second { get; }

    public Pair(TFirst first, TSecond second) =>
        (First, Second) = (first, second);
}
```

Un type de classe déclaré pour prendre des paramètres de type est appelé un *type de classe générique*. Les types struct, interface et délégué peuvent également être génériques. Lorsque la classe générique est utilisée, des arguments de type doivent être fournis pour chacun des paramètres de type :

C#

```
var pair = new Pair<int, string>(1, "two");
int i = pair.First;     //TFirst int
string s = pair.Second; //TSecond string
```

Un type générique avec des arguments de type fournis, comme `Pair<int,string>` ci-dessus, est appelé un *type construit*.

Classes de base

Une déclaration de classe peut spécifier une classe de base. Suivez les paramètres nom et type de la classe avec un symbole deux-points et le nom de la classe de base. L'omission d'une spécification de classe de base revient à dériver du type `object`. Dans l'exemple suivant, la classe de base de `Point3D` est `Point`. Dans le premier exemple, la classe de base de `Point` est `object` :

C#

```
public class Point3D : Point
{
    public int Z { get; set; }

    public Point3D(int x, int y, int z) : base(x, y)
    {
        Z = z;
    }
}
```

Une classe hérite des membres de sa classe de base. L'héritage signifie qu'une classe contient implicitement tous les membres de sa classe de base. Une classe n'hérite pas de l'instance et des constructeurs statiques, ni du finaliseur. Une classe dérivée peut ajouter de nouveaux membres dont elle hérite, mais elle ne peut pas supprimer la définition d'un membre hérité. Dans l'exemple précédent, `Point3D` hérite des membres `X` et `Y` de `Point` et chaque instance de `Point3D` contient trois propriétés `X`, `Y` et `Z`.

Il existe une conversion implicite d'un type de classe vers un de ses types de classe de base. Une variable d'un type de classe peut référencer une instance de cette classe ou une instance d'une classe dérivée. Par exemple, pour les déclarations de classe précédentes, une variable de type `Point` peut faire référence à un objet `Point` ou `Point3D` :

C#

```
Point a = new(10, 20);
Point b = new Point3D(10, 20, 30);
```

Structures

Les classes définissent des types qui prennent en charge l'héritage et le polymorphisme. Ils vous permettent de créer des comportements sophistiqués basés sur des hiérarchies de classes dérivées. En revanche, les types de***struct*** sont des types plus simples dont l'objectif principal est de stocker des valeurs de données. Les structs ne peuvent pas déclarer un type de base ; ils dérivent implicitement de System.ValueType. Vous ne pouvez pas dériver d'autres `struct` types à partir d'un type `struct`. Ils sont implicitement scellés.

C#

```
public struct Point
{
    public double X { get; }
    public double Y { get; }

    public Point(double x, double y) => (X, Y) = (x, y);
}
```

Interfaces

Une ***interface*** définit un contrat qui peut être implémenté par des classes et structures. Vous définissez une *interface* pour déclarer des fonctionnalités partagées parmi des types distincts. Par exemple, l'interface System.Collections.Generic.IEnumerable<T> définit une façon cohérente de parcourir tous les éléments d'une collection, comme un tableau. Une interface peut contenir des méthodes, des propriétés, des événements et des indexeurs. En général, une interface ne fournit pas les implémentations des membres qu'elle définit, elle indique simplement les membres qui doivent être fournis par les classes ou les structs qui implémentent l'interface.

Les interfaces peuvent utiliser ***l'héritage multiple***. Dans l'exemple suivant, l'interface `IComboBox` hérite à la fois de `ITextBox` et `IListBox`.

C#

```
interface IControl
{
    void Paint();
}

interface ITextBox : IControl
{
    void SetText(string text);
}

interface IListBox : IControl
```

```
{
    void SetItems(string[] items);
}

interface IComboBox : ITextBox, IListBox { }
```

Les classes et les structs peuvent implémenter plusieurs interfaces. Dans l'exemple suivant, la classe `EditBox` implémente à la fois `IControl` et `IDataBound`.

C#

```
interface IDataBound
{
    void Bind(Binder b);
}

public class EditBox : IControl, IDataBound
{
    public void Paint() { }
    public void Bind(Binder b) { }
}
```

Lorsqu'une classe ou un struct implémente une interface spécifique, les instances de cette classe ou struct peuvent être converties implicitement en ce type d'interface. Par exemple

C#

```
EditBox editBox = new();
IControl control = editBox;
IDataBound dataBound = editBox;
```

Énumérations

Un type ***Enum*** définit un ensemble de valeurs constantes. Les éléments suivants `enum` déclarent des constantes qui définissent différents légumes racines :

C#

```
public enum SomeRootVegetable
{
    HorseRadish,
    Radish,
    Turnip
}
```

Vous pouvez également définir un `enum` à utiliser en combinaison comme indicateurs. La déclaration suivante déclare un ensemble d'indicateurs pour les quatre saisons. Toute combinaison de saisons peut être appliquée, y compris une valeur `All` qui inclut toutes les saisons :

C#

```
[Flags]
public enum Seasons
{
    None = 0,
    Summer = 1,
    Autumn = 2,
    Winter = 4,
    Spring = 8,
    All = Summer | Autumn | Winter | Spring
}
```

L'exemple suivant montre les déclarations des deux énumérations précédentes :

C#

```
var turnip = SomeRootVegetable.Turnip;

var spring = Seasons.Spring;
var startingOnEquinox = Seasons.Spring | Seasons.Autumn;
var theYear = Seasons.All;
```

Types Nullable

Les variables de tout type peuvent être ***déclarées comme non-nullables*** ou ***nullables***. Une variable nullable peut contenir une valeur supplémentaire `null`, indiquant aucune valeur. Les types valeur nullable (structs ou énumérations) sont représentés par System.Nullable<T>. Les types référence non nullable et référence nullable sont tous deux représentés par le type de référence sous-jacent. La distinction est représentée par les métadonnées lues par le compilateur et certaines bibliothèques. Le compilateur émet des avertissements lorsque les références nullables sont déréférencées sans avoir d'abord vérifié leur valeur par rapport à `null`. Le compilateur fournit également des avertissements lorsque des références non nullables se voient attribuer une valeur qui peut être `null`. L'exemple suivant déclare un ***int nullable*** et l'initialise à `null`. Ensuite, il définit la valeur sur `5`. Il démontre le même concept avec une ***chaîne nullable***. Pour plus d'informations, consultez types de valeur nullables et types de référence nullables.

C#

```
int? optionalInt = default;
optionalInt = 5;
string? optionalText = default;
optionalText = "Hello World.";
```

Tuples

C# prend en charge les ***tuples***, qui fournissent une syntaxe concise pour regrouper plusieurs éléments de données dans une structure de données légère. Vous instanciez un tuple en déclarant les types et les noms des membres entre `(` et `)`, comme illustré dans l'exemple suivant :

C#

```
(double Sum, int Count) t2 = (4.5, 3);
Console.WriteLine($"Sum of {t2.Count} elements is {t2.Sum}.");
//Output:
//Sum of 3 elements is 4.5.
```

Les tuples offrent une alternative à la structure de données avec plusieurs membres, sans utiliser les blocs de construction décrits dans l'article suivant.

Précédent Suivant

Principaux composants des programmes C#

Article • 07/04/2023

Les types décrits dans l'article précédent de cette série sur C# sont créés à l'aide de ces principaux composants :

- Membres, tels que les propriétés, les champs, les méthodes et les événements.
- Expressions
- Instructions

Membres

Les membres d'un `class` sont des ***membres statiques*** ou des ***membres d'instance***. Les membres statiques appartiennent à des classes, et les membres d'instance appartiennent à des objets (instances de classes).

La liste suivante fournit une vue d'ensemble des types de membres qu'une classe peut contenir.

- **Constantes** : valeurs constantes associées à la classe
- **Champs** : variables associées à la classe
- **Méthodes** : actions pouvant être effectuées par la classe
- **Propriétés** : actions associées à la lecture et à l'écriture de propriétés nommées de la classe
- **Indexeurs** : actions associées à l'indexation d'instances de la classe comme un tableau
- **Événements** : notifications pouvant être générées par la classe
- **Opérateurs** : conversions et opérateurs d'expression pris en charge par la classe
- **Constructeurs** : actions requises pour initialiser des instances de la classe ou la classe elle-même
- **Finaliseurs** : actions effectuées avant que les instances de la classe soient définitivement ignorées
- **Types** : types imbriqués déclarés par la classe

Accessibilité

Chaque membre d'une classe a une accessibilité associée, qui contrôle les régions du texte du programme qui sont en mesure d'accéder au membre. Il existe six formes

possibles d'accessibilité. Les modificateurs d'accès sont résumés ci-dessous.

- `public` : l'accès n'est pas limité.
- `private` : l'accès est limité à cette classe.
- `protected` : l'accès est limité à cette classe ou aux classes dérivées de cette classe.
- `internal` : l'accès est limité à l'assembly actuel (`.exe` ou `.dll`).
- `protected internal` : l'accès est limité à cette classe, aux classes dérivées de cette classe ou aux classes appartenant au même assembly.
- `private protected` : l'accès est limité à cette classe ou aux classes dérivées de ce type appartenant au même assembly.

Champs

Un *champ* est une variable qui est associée à une classe ou une instance d'une classe.

Un champ déclaré avec le modificateur static définit un champ statique. Un champ statique identifie exactement un seul emplacement de stockage. Peu importe le nombre d'instances d'une classe qui sont créées, il n'existe qu'une seule copie d'un champ statique.

Un champ déclaré sans le modificateur static définit un champ d'instance. Chaque instance d'une classe contient une copie distincte de tous les champs d'instance de cette classe.

Dans l'exemple suivant, chaque instance de la classe `Color` a une copie distincte des champs d'instance `R`, `G` et `B`, mais il n'existe qu'une seule copie des champs statiques `Black`, `White`, `Red`, `Green` et `Blue` :

C#

```
public class Color
{
    public static readonly Color Black = new(0, 0, 0);
    public static readonly Color White = new(255, 255, 255);
    public static readonly Color Red = new(255, 0, 0);
    public static readonly Color Green = new(0, 255, 0);
    public static readonly Color Blue = new(0, 0, 255);

    public byte R;
    public byte G;
    public byte B;

    public Color(byte r, byte g, byte b)
    {
        R = r;
        G = g;
```

```
            B = b;
        }
    }
```

Comme indiqué dans l'exemple précédent, les *champs en lecture seule* peuvent être déclarés avec un modificateur `readonly`. L'affectation à un champ en lecture seule peut survenir uniquement dans le cadre de la déclaration du champ ou dans un constructeur de la même classe.

Méthodes

Une *méthode* est un membre qui implémente un calcul ou une action qui peut être effectuée par un objet ou une classe. Les *méthodes statiques* sont accessibles à travers la classe. Les *méthodes d'instance* sont accessibles via des instances de la classe.

Les méthodes peuvent avoir une liste de *paramètres*, qui représentent des valeurs ou des références de variables transmises à la méthode. Les méthodes ont un *type de retour*, qui spécifie le type de la valeur calculée et retournée par la méthode. Le type de retour d'une méthode est `void` si elle ne retourne pas de valeur.

Comme les types, les méthodes peuvent également être un jeu de paramètres de type pour lesquels les arguments de type doivent être spécifiés lorsque la méthode est appelée. Contrairement aux types, les arguments de type peuvent souvent être déduits à partir des arguments d'un appel de méthode et n'ont pas à être fournis explicitement.

La *signature* d'une méthode doit être unique dans la classe dans laquelle la méthode est déclarée. La signature d'une méthode se compose du nom de la méthode, du nombre de paramètres de type et du nombre, des modificateurs et des types de ses paramètres. La signature d'une méthode n'inclut pas le type de retour.

Lorsqu'un corps de méthode est une expression unique, la méthode peut être définie à l'aide d'un format d'expression compact, comme illustré dans l'exemple suivant :

C#

```
public override string ToString() => "This is an object";
```

Paramètres

Les paramètres sont utilisés pour passer des valeurs ou des références variables aux méthodes. Les paramètres d'une méthode obtiennent leurs valeurs réelles à partir des *arguments* qui sont spécifiés lorsque la méthode est appelée. Il existe quatre types de

paramètres : les paramètres de valeur, les paramètres de référence, les paramètres de sortie et les tableaux de paramètres.

Un *paramètre de valeur* est utilisé pour passer des arguments en entrée. Un paramètre de valeur correspond à une variable locale qui obtient sa valeur initiale de l'argument qui a été transmis pour le paramètre. Les modifications apportées à un paramètre de valeur n'affectent pas l'argument qui a été transmis pour le paramètre.

Les paramètres de valeur peuvent être facultatifs, en spécifiant une valeur par défaut afin que les arguments correspondants puissent être omis.

Un *paramètre de référence* est utilisé pour passer des arguments par référence. L'argument transmis pour un paramètre de référence doit être une variable avec une valeur définie. Pendant l'exécution de la méthode, le paramètre de référence représente le même emplacement de stockage que la variable d'argument. Un paramètre de référence est déclaré avec le modificateur `ref`. L'exemple suivant illustre l'utilisation des paramètres `ref`.

C#

```
static void Swap(ref int x, ref int y)
{
    int temp = x;
    x = y;
    y = temp;
}

public static void SwapExample()
{
    int i = 1, j = 2;
    Swap(ref i, ref j);
    Console.WriteLine($"{i} {j}");    // "2 1"
}
```

Un *paramètre de sortie* est utilisé pour passer des arguments par référence. Il est similaire à un paramètre de référence, sauf qu'il ne nécessite pas d'affecter explicitement une valeur à l'argument fourni par l'appelant. Un paramètre de sortie est déclaré avec le modificateur `out`. L'exemple suivant illustre l'utilisation des paramètres `out`.

C#

```
static void Divide(int x, int y, out int quotient, out int remainder)
{
    quotient = x / y;
    remainder = x % y;
}
```

```
public static void OutUsage()
{
    Divide(10, 3, out int quo, out int rem);
    Console.WriteLine($"{quo} {rem}");  // "3 1"
}
```

Un *tableau de paramètres* autorise le passage d'un nombre variable d'arguments à une méthode. Un tableau de paramètres est déclaré avec le modificateur `params`. Seul le dernier paramètre d'une méthode peut être un tableau de paramètres, et le type d'un tableau de paramètres doit être un type tableau unidimensionnel. Les méthodes `Write` et `WriteLine` de la classe System.Console sont de bons exemples d'utilisation des tableaux de paramètres. Elles sont déclarées comme suit.

C#

```
public class Console
{
    public static void Write(string fmt, params object[] args) { }
    public static void WriteLine(string fmt, params object[] args) { }
    // ...
}
```

Dans une méthode qui utilise un tableau de paramètres, le tableau de paramètres se comporte exactement comme un paramètre ordinaire de type tableau. Toutefois, dans un appel à une méthode avec un tableau de paramètres, il est possible de passer un argument unique de type tableau de paramètres ou n'importe quel nombre d'arguments du type d'élément du tableau de paramètres. Dans ce cas, une instance de tableau est automatiquement créée et initialisée avec les arguments donnés. Cet exemple

C#

```
int x, y, z;
x = 3;
y = 4;
z = 5;
Console.WriteLine("x={0} y={1} z={2}", x, y, z);
```

revient à écrire ce qui suit.

C#

```
int x = 3, y = 4, z = 5;

string s = "x={0} y={1} z={2}";
object[] args = new object[3];
```

```
args[0] = x;
args[1] = y;
args[2] = z;
Console.WriteLine(s, args);
```

Corps de la méthode et variables locales

Le corps d'une méthode spécifie les instructions à exécuter lorsque la méthode est appelée.

Un corps de méthode peut déclarer des variables qui sont spécifiques à l'appel de la méthode. Ces variables sont appelées *variables locales*. Une déclaration de variable locale spécifie un nom de type, un nom de variable et éventuellement une valeur initiale. L'exemple suivant déclare une variable locale `i` avec une valeur initiale de zéro et une variable locale `j` sans valeur initiale.

C#

```
class Squares
{
    public static void WriteSquares()
    {
        int i = 0;
        int j;
        while (i < 10)
        {
            j = i * i;
            Console.WriteLine($"{i} x {i} = {j}");
            i++;
        }
    }
}
```

C# requiert qu'une variable locale soit *assignée de manière définitive* avant de pouvoir obtenir sa valeur. Par exemple, si la déclaration du `i` précédent n'inclut pas de valeur initiale, le compilateur signale une erreur pour les utilisations ultérieures de `i`, car `i` ne serait pas assigné de manière définitive à ces points dans le programme.

Une méthode peut utiliser les instructions `return` pour retourner le contrôle à son appelant. Dans une méthode retournant `void`, les instructions `return` ne peuvent pas spécifier une expression. Dans une méthode avec un type de retour non void, les instructions `return` doivent inclure une expression qui calcule la valeur de retour.

Méthodes statiques et d'instance

Une méthode déclarée avec un modificateur `static` est une *méthode statique*. Une méthode statique n'opère pas sur une instance spécifique et permet uniquement d'accéder directement à des membres statiques.

Une méthode déclarée sans modificateur `static` est une *méthode d'instance*. Une méthode d'instance opère sur une instance spécifique et peut accéder aux membres statiques et d'instance. L'instance sur laquelle une méthode d'instance a été appelée est explicitement accessible en tant que `this`. Une erreur consiste à faire référence à `this` dans une méthode statique.

La classe `Entity` suivante a à la fois des statiques et des membres d'instance.

C#

```
class Entity
{
    static int s_nextSerialNo;
    int _serialNo;

    public Entity()
    {
        _serialNo = s_nextSerialNo++;
    }

    public int GetSerialNo()
    {
        return _serialNo;
    }

    public static int GetNextSerialNo()
    {
        return s_nextSerialNo;
    }

    public static void SetNextSerialNo(int value)
    {
        s_nextSerialNo = value;
    }
}
```

Chaque instance `Entity` contient un numéro de série (et probablement d'autres informations qui ne sont pas indiquées ici). Le constructeur `Entity` (qui est similaire à une méthode d'instance) initialise la nouvelle instance avec le numéro de série suivant. Étant donné que le constructeur est un membre d'instance, il est autorisé à accéder à la fois au champ d'instance `_serialNo` et au champ statique `s_nextSerialNo`.

Les méthodes statiques `GetNextSerialNo` et `SetNextSerialNo` peuvent accéder au champ statique `s_nextSerialNo`, mais ce serait une erreur pour eux d'accéder directement au champ d'instance `_serialNo`.

L'exemple suivant montre l'utilisation de la classe `Entity`.

```
C#

Entity.SetNextSerialNo(1000);
Entity e1 = new();
Entity e2 = new();
Console.WriteLine(e1.GetSerialNo());          // Outputs "1000"
Console.WriteLine(e2.GetSerialNo());          // Outputs "1001"
Console.WriteLine(Entity.GetNextSerialNo());  // Outputs "1002"
```

Les méthodes statiques `SetNextSerialNo` et `GetNextSerialNo` sont appelées sur la classe alors que la méthode d'instance `GetSerialNo` est appelée sur les instances de la classe.

Méthodes virtuelles, de substitution et abstraites

Vous utilisez des méthodes virtuelles, de substitution et abstraites pour définir le comportement d'une hiérarchie de types de classes. Comme une classe peut dériver d'une classe de base, ces classes dérivées peuvent avoir besoin de modifier le comportement implémenté dans la classe de base. Une méthode ***virtuelle*** est déclarée et implémentée dans une classe de base où toute classe dérivée peut fournir une implémentation plus spécifique. Une méthode ***de substitution*** est une méthode implémentée dans une classe dérivée qui modifie le comportement de l'implémentation de la classe de base. Une méthode ***abstraite*** est une méthode déclarée dans une classe de base qui *doit* être substituée dans toutes les classes dérivées. En fait, les méthodes abstraites ne définissent pas d'implémentation dans la classe de base.

Les appels de méthode aux méthodes d'instance peuvent être résolus en implémentations de classe de base ou de classe dérivée. Le type d'une variable détermine son *type au moment de la compilation*. Le *type au moment de la compilation* est le type utilisé par le compilateur pour déterminer ses membres. Toutefois, une variable peut être affectée à une instance de n'importe quel type dérivé de son *type au moment de la compilation*. Le *type au moment de l'exécution* est le type de l'instance réelle à laquelle une variable fait référence.

Lorsqu'une méthode virtuelle est appelée, le *type au moment de l'exécution* de l'instance pour laquelle cet appel prend place détermine l'implémentation de méthode à appeler. Dans un appel de méthode non virtuelle, le *type lors de la compilation* de l'instance est le facteur déterminant.

Une méthode virtuelle peut être *substituée* dans une classe dérivée. Lorsqu'une déclaration de méthode d'instance comprend un modificateur override, la méthode substitue une méthode virtuelle héritée ayant la même signature. Une déclaration de méthode virtuelle introduit une nouvelle méthode. Une déclaration de méthode de substitution spécialise une méthode virtuelle héritée existante en fournissant une nouvelle implémentation de cette méthode.

Une *méthode abstraite* est une méthode virtuelle sans implémentation. Une méthode abstraite est déclarée avec le modificateur `abstract` et n'est autorisée que dans une classe abstraite. Une méthode abstraite doit être remplacée dans chaque classe dérivée non abstraite.

L'exemple suivant déclare une classe abstraite, `Expression`, qui représente un nœud d'arborescence de l'expression et trois classes dérivées, `Constant`, `VariableReference` et `Operation`, qui implémentent des nœuds d'arborescence de l'expression pour les références variables, les constantes et les opérations arithmétiques. (Cet exemple est similaire aux types arborescence de l'expression, mais ne les confondez pas).

C#

```
public abstract class Expression
{
    public abstract double Evaluate(Dictionary<string, object> vars);
}

public class Constant : Expression
{
    double _value;

    public Constant(double value)
    {
        _value = value;
    }

    public override double Evaluate(Dictionary<string, object> vars)
    {
        return _value;
    }
}

public class VariableReference : Expression
{
    string _name;

    public VariableReference(string name)
    {
        _name = name;
    }
```

```
    public override double Evaluate(Dictionary<string, object> vars)
    {
        object value = vars[_name] ?? throw new Exception($"Unknown
variable: {_name}");
        return Convert.ToDouble(value);
    }
}

public class Operation : Expression
{
    Expression _left;
    char _op;
    Expression _right;

    public Operation(Expression left, char op, Expression right)
    {
        _left = left;
        _op = op;
        _right = right;
    }

    public override double Evaluate(Dictionary<string, object> vars)
    {
        double x = _left.Evaluate(vars);
        double y = _right.Evaluate(vars);
        switch (_op)
        {
            case '+': return x + y;
            case '-': return x - y;
            case '*': return x * y;
            case '/': return x / y;

            default: throw new Exception("Unknown operator");
        }
    }
}
```

Les quatre classes précédentes permet de modéliser des expressions arithmétiques. Par exemple, en utilisant des instances de ces classes, l'expression `x + 3` peut être représentée comme suit.

C#

```
Expression e = new Operation(
    new VariableReference("x"),
    '+',
    new Constant(3));
```

La méthode `Evaluate` d'une instance `Expression` est appelée pour évaluer l'expression donnée et produire une valeur `double`. La méthode prend un argument `Dictionary` qui

contient des noms de variables (comme clés des entrées) et des valeurs (comme valeurs des entrées). Comme `Evaluate` est une méthode abstraite, les classes non abstraites dérivées de `Expression` doivent remplacer `Evaluate`.

Une implémentation de `Constant` de `Evaluate` renvoie simplement la constante stockée. Une implémentation de `VariableReference` recherche le nom de variable dans le dictionnaire et renvoie la valeur résultante. Une implémentation de `Operation` évalue d'abord les opérandes de gauche et de droite (en appelant de manière récursive leurs méthodes `Evaluate`), puis effectue l'opération arithmétique donnée.

Le programme suivant utilise les classes `Expression` pour évaluer l'expression `x * (y + 2)` pour différentes valeurs de `x` et `y`.

C#

```
Expression e = new Operation(
    new VariableReference("x"),
    '*',
    new Operation(
        new VariableReference("y"),
        '+',
        new Constant(2)
    )
);
Dictionary<string, object> vars = new();
vars["x"] = 3;
vars["y"] = 5;
Console.WriteLine(e.Evaluate(vars)); // "21"
vars["x"] = 1.5;
vars["y"] = 9;
Console.WriteLine(e.Evaluate(vars)); // "16.5"
```

Surcharge de méthode

La *surcharge* de méthode permet d'avoir plusieurs méthodes dans la même classe avec le même nom, tant qu'elles ont des signatures uniques. Lors de la compilation d'un appel à une méthode surchargée, le compilateur utilise *la résolution de surcharge* pour déterminer la méthode spécifique à appeler. La résolution de surcharge recherche la méthode qui correspond le mieux aux arguments. Si aucune correspondance unique n'est trouvée, une erreur est signalée. L'exemple suivant montre la résolution de surcharge en action. Le commentaire pour chaque appel de la méthode `UsageExample` affiche quelle méthode est appelée.

C#

```
class OverloadingExample
{
    static void F() => Console.WriteLine("F()");
    static void F(object x) => Console.WriteLine("F(object)");
    static void F(int x) => Console.WriteLine("F(int)");
    static void F(double x) => Console.WriteLine("F(double)");
    static void F<T>(T x) => Console.WriteLine($"F<T>(T), T is
{typeof(T)}");
    static void F(double x, double y) => Console.WriteLine("F(double,
double)");

    public static void UsageExample()
    {
        F();            // Invokes F()
        F(1);           // Invokes F(int)
        F(1.0);         // Invokes F(double)
        F("abc");       // Invokes F<T>(T), T is System.String
        F((double)1);   // Invokes F(double)
        F((object)1);   // Invokes F(object)
        F<int>(1);      // Invokes F<T>(T), T is System.Int32
        F(1, 1);        // Invokes F(double, double)
    }
}
```

Comme le montre l'exemple, une méthode particulière peut toujours être sélectionnée en effectuant un cast explicite des arguments aux types de paramètres et arguments de type exacts.

Autres fonctions membres

Les membres qui contiennent du code exécutable sont collectivement regroupés sous les *membres de fonction* d'une classe. La section précédente décrit les méthodes qui sont des types principaux des fonctions membres. Cette section décrit les autres types de fonctions membres pris en charge par C# : constructeurs, propriétés, indexeurs, événements, opérateurs et finaliseurs.

L'exemple suivant représente une classe générique nommée `MyList<T>`, qui implémente une liste d'objets évolutive. La classe contient plusieurs exemples des types les plus courants de membres de fonction.

C#

```
public class MyList<T>
{
    const int DefaultCapacity = 4;

    T[] _items;
```

```
    int _count;

    public MyList(int capacity = DefaultCapacity)
    {
        _items = new T[capacity];
    }

    public int Count => _count;

    public int Capacity
    {
        get =>  _items.Length;
        set
        {
            if (value < _count) value = _count;
            if (value != _items.Length)
            {
                T[] newItems = new T[value];
                Array.Copy(_items, 0, newItems, 0, _count);
                _items = newItems;
            }
        }
    }

    public T this[int index]
    {
        get => _items[index];
        set
        {
            if (!object.Equals(_items[index], value)) {
                _items[index] = value;
                OnChanged();
            }
        }
    }

    public void Add(T item)
    {
        if (_count == Capacity) Capacity = _count * 2;
        _items[_count] = item;
        _count++;
        OnChanged();
    }
    protected virtual void OnChanged() =>
        Changed?.Invoke(this, EventArgs.Empty);

    public override bool Equals(object other) =>
        Equals(this, other as MyList<T>);

    static bool Equals(MyList<T> a, MyList<T> b)
    {
        if (Object.ReferenceEquals(a, null)) return
Object.ReferenceEquals(b, null);
        if (Object.ReferenceEquals(b, null) || a._count != b._count)
            return false;
```

```
        for (int i = 0; i < a._count; i++)
        {
            if (!object.Equals(a._items[i], b._items[i]))
            {
                return false;
            }
        }
        return true;
    }

    public event EventHandler Changed;

    public static bool operator ==(MyList<T> a, MyList<T> b) =>
        Equals(a, b);

    public static bool operator !=(MyList<T> a, MyList<T> b) =>
        !Equals(a, b);
}
```

Constructeurs

C# prend en charge les constructeurs statiques et d'instance. Un *constructeur d'instance* est un membre qui implémente les actions requises pour initialiser une instance d'une classe. Un *constructeur statique* est un membre qui implémente les actions requises pour initialiser une classe elle-même lors de son chargement.

Un constructeur est déclaré comme une méthode sans aucun type de retour et le même nom que la classe contenante. Si une déclaration de constructeur inclut un modificateur `static`, elle déclare un constructeur statique. Dans le cas contraire, elle déclare un constructeur d'instance.

Les constructeurs d'instance peuvent être surchargés et avoir des paramètres facultatifs. Par exemple, la classe `MyList<T>` déclare un constructeur d'instance avec un seul paramètre `int` facultatif. Les constructeurs d'instance sont appelés en utilisant l'opérateur `new`. Les instructions suivantes allouent deux instances `MyList<string>` à l'aide du constructeur de la classe `MyList` avec et sans l'argument facultatif.

C#

```
MyList<string> list1 = new();
MyList<string> list2 = new(10);
```

Contrairement à d'autres membres, les constructeurs d'instance ne sont pas hérités. Une classe n'a aucun constructeur d'instance autre que les constructeurs réellement déclarés

dans la classe. Si aucun constructeur d'instance n'est fourni pour une classe, un constructeur vide sans paramètre est fourni automatiquement.

Propriétés

Les *propriétés* sont une extension naturelle des champs. Les deux sont des membres nommés avec des types associés, et la syntaxe pour accéder aux champs et propriétés est la même. Toutefois, contrairement aux champs, les propriétés ne désignent pas des emplacements de stockage. Au lieu de cela, les propriétés ont des *accesseurs* qui spécifient les instructions à exécuter lorsque les valeurs sont lues ou écrites. Un *accesseur get* lit la valeur. Un *accesseur set* écrit la valeur.

Une propriété est déclarée comme un champ, sauf que la déclaration se termine par un accesseur get ou un accesseur set écrit entre les délimiteurs `{` et `}` au lieu de se terminer par un point-virgule. Une propriété qui a à la fois un accesseur get et un accesseur set est une *propriété en lecture-écriture*. Une propriété qui n'a qu'un accesseur get est une *propriété en lecture seule*. Une propriété qui n'a qu'un accesseur set est une *propriété en écriture seule*.

Un accesseur get correspond à une méthode sans paramètre avec une valeur de retour du type de la propriété. Un accesseur set correspond à une méthode avec un paramètre unique nommé valeur et aucun type de retour. L'accesseur get calcule la valeur de la propriété. L'accesseur set fournit une nouvelle valeur pour la propriété. Lorsque la propriété est la cible d'une affectation, ou l'opérande de `++` ou `--`, l'accesseur set est appelé. Dans d'autres cas où la propriété est référencée, l'accesseur get est appelé.

La classe `MyList<T>` déclare deux propriétés, `Count` et `Capacity`, qui sont respectivement en lecture seule et en lecture-écriture. Le code suivant est un exemple d'utilisation de ces propriétés :

C#

```
MyList<string> names = new();
names.Capacity = 100;   // Invokes set accessor
int i = names.Count;    // Invokes get accessor
int j = names.Capacity; // Invokes get accessor
```

C# prend en charge les propriétés d'instance et les propriétés statiques, similaires aux champs et aux méthodes. Les propriétés statiques sont déclarées avec le modificateur static, et les propriétés d'instance sont déclarées sans.

Le ou les accesseurs d'une propriété peuvent être virtuels. Lorsqu'une déclaration de propriété inclut un modificateur `virtual`, `abstract` ou `override`, elle s'applique aux

accesseurs de la propriété.

Indexeurs

Un *indexeur* est un membre qui permet l'indexation des objets de la même façon en tant que tableau. Un indexeur est déclaré comme une propriété, sauf que le nom du membre est `this`, suivi d'une liste de paramètres écrits entre les délimiteurs `[` et `]`. Les paramètres sont disponibles dans le ou les accesseurs de l'indexeur. Similaires aux propriétés, les indexeurs peuvent être en lecture-écriture, en lecture seule et en écriture seule, et les accesseurs d'un indexeur peuvent être virtuels.

La classe `MyList<T>` déclare un indexeur en lecture-écriture unique qui prend un paramètre `int`. L'indexeur rend possible l'indexation des instances `MyList<T>` avec des valeurs `int`. Par exemple :

C#

```
MyList<string> names = new();
names.Add("Liz");
names.Add("Martha");
names.Add("Beth");
for (int i = 0; i < names.Count; i++)
{
    string s = names[i];
    names[i] = s.ToUpper();
}
```

Les indexeurs peuvent être surchargés. Une classe peut déclarer plusieurs indexeurs tant que le nombre ou les types de leurs paramètres diffèrent.

Événements

Un *événement* est un membre qui permet à une classe ou un objet de fournir des notifications. Un événement est déclaré comme un champ, sauf que la déclaration inclut un mot-clé `event` et que le type doit être un type délégué.

Dans une classe qui déclare un membre d'événement, l'événement se comporte comme un champ d'un type délégué (à condition que l'événement ne soit pas abstrait et ne déclare pas d'accesseurs). Le champ stocke une référence à un délégué qui représente les gestionnaires d'événements qui ont été ajoutés à l'événement. Si aucun gestionnaire d'événements n'est présent, le champ est `null`.

La classe `MyList<T>` déclare un seul membre d'événement appelé `Changed`, qui indique qu'un nouvel élément a été ajouté à la liste ou qu'un élément de liste a été modifié à l'aide de l'accesseur set de l'indexeur. L'événement Changed est déclenché par la méthode virtuelle `OnChanged`, qui vérifie si l'événement est `null` (ce qui signifie qu'aucun gestionnaire n'est présent). La notion de déclenchement d'un événement équivaut précisément à appeler le délégué représenté par l'événement. Il n'existe aucune construction de langage spéciale pour déclencher des événements.

Les clients réagissent aux événements via les *gestionnaires d'événements*. Les gestionnaires d'événements sont joints à l'aide de l'opérateur `+=` et supprimés à l'aide de l'opérateur `-=`. L'exemple suivant joint un gestionnaire d'événements à l'événement `Changed` d'un `MyList<string>`.

C#

```
class EventExample
{
    static int s_changeCount;

    static void ListChanged(object sender, EventArgs e)
    {
        s_changeCount++;
    }

    public static void Usage()
    {
        var names = new MyList<string>();
        names.Changed += new EventHandler(ListChanged);
        names.Add("Liz");
        names.Add("Martha");
        names.Add("Beth");
        Console.WriteLine(s_changeCount); // "3"
    }
}
```

Pour les scénarios avancés où le contrôle du stockage sous-jacent d'un événement est souhaité, une déclaration d'événement peut fournir explicitement des accesseurs `add` et `remove`, qui sont similaires à l'accesseur `set` d'une propriété.

Opérateurs

Un *opérateur* est un membre qui définit la signification de l'application d'un opérateur d'expression particulière aux instances d'une classe. Trois types d'opérateurs peuvent être définis : les opérateurs unaires, les opérateurs binaires et les opérateurs de conversion. Tous les opérateurs doivent être déclarés comme `public` et `static`.

La classe `MyList<T>` déclare deux opérateurs : `operator ==` et `operator !=`. Ces opérateurs substitués donnent une nouvelle signification aux expressions qui appliquent ces opérateurs aux instances `MyList`. Plus précisément, les opérateurs définissent l'égalité de deux instances `MyList<T>` comme la comparaison de chacun des objets contenus à l'aide de leurs méthodes `Equals`. L'exemple suivant utilise l'opérateur `==` pour comparer deux instances de `MyList<int>`.

C#

```
MyList<int> a = new();
a.Add(1);
a.Add(2);
MyList<int> b = new();
b.Add(1);
b.Add(2);
Console.WriteLine(a == b);  // Outputs "True"
b.Add(3);
Console.WriteLine(a == b);  // Outputs "False"
```

La première `Console.WriteLine` génère `True`, car les deux listes contiennent le même nombre d'objets avec les mêmes valeurs dans le même ordre. Si `MyList<T>` n'avait pas défini `operator ==`, la première `Console.WriteLine` aurait affiché `False`, car `a` et `b` référencent des instances `MyList<int>` différentes.

Finaliseurs

Un *finaliseur* est un membre qui implémente les actions requises pour finaliser une instance d'une classe. En règle générale, un finaliseur est nécessaire pour libérer des ressources non managées. Les finaliseurs ne peuvent pas avoir de paramètres, ils ne peuvent pas avoir de modificateurs d'accessibilité et ils ne peuvent pas être appelés explicitement. Le finaliseur pour une instance est appelé automatiquement lors du nettoyage de la mémoire (garbage collection). Pour plus d'informations, consultez l'article sur les finaliseurs.

Le récupérateur de mémoire est libre de déterminer quand collecter des objets et exécuter des finaliseurs. Plus précisément, le timing des appels du finaliseur n'est pas déterministe, et les finaliseurs ne peuvent être exécutés sur n'importe quel thread. Pour ces raisons et d'autres, les classes doivent implémenter des finaliseurs uniquement lorsqu'aucune des autres solutions n'est envisageable.

L'instruction `using` fournit une meilleure approche pour la destruction d'objets.

Expressions

Les *expressions* sont construites à partir de *d'opérandes* et *d'opérateurs*. Les opérateurs d'une expression indiquent les opérations à appliquer aux opérandes. Parmi les exemples d'opérateurs figurent `+`, `-`, `*`, `/` et `new`. Les littéraux, les champs, les variables locales et les expressions sont des exemples d'opérandes.

Quand une expression contient plusieurs opérateurs, la *priorité* des opérateurs contrôle l'ordre dans lequel ils sont évalués. Par exemple, l'expression `x + y * z` est évaluée comme `x + (y * z)`, car l'opérateur `*` a une priorité plus élevée que `+`.

Lorsqu'un opérande se produit entre deux opérateurs de même priorité, *l'associativité* des opérateurs détermine l'ordre dans lequel les opérations sont effectuées :

- À l'exception des opérateurs d'assignation et de coalescence nulle, tous les opérateurs binaires sont *associatifs sur leur gauche*, ce qui signifie que les opérations sont effectuées de gauche à droite. Par exemple, `x + y + z` est évalué comme étant `(x + y) + z`.
- Les opérateurs d'assignation, les opérateurs `??` et `??=` de coalescence nulle et l'opérateur conditionnel `?:` sont *associatifs sur leur droite*, ce qui signifie que les opérations sont effectuées de droite à gauche. Par exemple, `x = y = z` est évalué comme étant `x = (y = z)`.

La priorité et l'associativité peuvent être contrôlées à l'aide de parenthèses. Par exemple, `x + y * z` multiplie d'abord `y` par `z`, puis ajoute le résultat à `x`, mais `(x + y) * z` ajoute d'abord `x` et `y`, puis multiplie le résultat par `z`.

La plupart des opérateurs peuvent être *surchargés*. La surcharge d'opérateur autorise la spécification d'implémentations d'opérateurs définies par l'utilisateur pour les opérations où l'un des deux opérandes ou les deux sont d'un type classe ou struct défini par l'utilisateur.

C# met à votre disposition des opérateurs pour effectuer des opérations arithmétiques, logiques, au niveau du bit et de décalage et des comparaisons d'égalité et d'ordre .

Pour obtenir la liste complète des opérateurs C# classés par niveau de priorité, consultez Opérateurs C#.

Instructions

Les actions d'un programme sont exprimées à l'aide *d'instructions*. C# prend en charge plusieurs types d'instructions, dont plusieurs sont définies en termes d'instructions

incorporées.

- Un *bloc* autorise l'écriture de plusieurs instructions dans les contextes où une seule instruction est autorisée. Un bloc se compose d'une liste d'instructions écrites entre les délimiteurs `{` et `}`.
- Les *instructions de déclaration* sont utilisées pour déclarer des variables locales et des constantes.
- Les *instructions d'expression* sont utilisées pour évaluer des expressions. Les expressions qui peuvent être utilisées en tant qu'instructions comprennent les appels de méthode, les allocations d'objet avec l'opérateur `new`, les affectations avec `=` et les opérateurs d'assignation composée, les opérations d'incrémentation et de décrémentation à l'aide des opérateurs `++` et `--` et les expressions `await`.
- Les *instructions de sélection* sont utilisées pour sélectionner un nombre d'instructions possibles pour l'exécution en fonction de la valeur d'une expression. Ce groupe contient les instructions `if` et `switch`.
- Les *instructions d'itération* servent à exécuter à plusieurs reprises une instruction incorporée. Ce groupe contient les instructions `while`, `do`, `for` et `foreach`.
- Les *instructions de saut* sont utilisées pour transférer le contrôle. Ce groupe contient les instructions `break`, `continue`, `goto`, `throw`, `return` et `yield`.
- L'instruction `try`…`catch` est utilisée pour intercepter les exceptions qui se produisent pendant l'exécution d'un bloc, et l'instruction `try`…`finally` permet de spécifier que le code de finalisation est toujours exécuté, qu'une exception se soit produite ou non.
- Les instructions `checked` et `unchecked` permettent de contrôler le contexte de dépassement de capacité pour les conversions et les opérations arithmétiques de type intégral.
- L'instruction `lock` est utilisée pour obtenir le verrou d'exclusion mutuelle pour un objet donné, exécuter une instruction, puis libérer le verrou.
- L'instruction `using` est utilisée pour obtenir une ressource, exécuter une instruction puis supprimer cette ressource.

Les listes suivantes répertorient les types d'instructions qui peuvent être utilisés :

- Déclaration de variable locale.
- Déclaration de constante locale.
- Instruction d'expression.
- Instruction `if`.
- Instruction `switch`.
- Instruction `while`.
- Instruction `do`.

- Instruction `for`.
- Instruction `foreach`.
- Instruction `break`.
- Instruction `continue`.
- Instruction `goto`.
- Instruction `return`.
- Instruction `yield`.
- Instructions `throw` et `try`.
- Instructions `checked` et `unchecked`.
- Instruction `lock`.
- Instruction `using`.

Précédent Suivant

Domaines principaux du langage C#

Article • 16/03/2023

Cet article présente les principales fonctionnalités du langage C#.

Tableaux, collections et LINQ

C# et .NET fournissent de nombreux types de collection différents. Les tableaux ont une syntaxe définie par le langage. Les types de collection génériques sont répertoriés dans l'espace de noms System.Collections.Generic. Les collections spécialisées incluent System.Span<T> pour l'accès à la mémoire continue sur la frame de pile et System.Memory<T> pour l'accès à la mémoire continue sur le tas managé. Toutes les collections, y compris les tableaux, Span<T> et Memory<T> partagent un principe unifié pour l'itération. Vous utilisez l'interface System.Collections.Generic.IEnumerable<T>. Ce principe d'unification signifie que tous les types de collection peuvent être utilisés avec des requêtes LINQ ou d'autres algorithmes. Vous écrivez des méthodes à l'aide de IEnumerable<T> et ces algorithmes utilisent n'importe quelle collection.

Tableaux

Un ***tableau*** est une structure de données qui contient un certain nombre de variables qui sont accessibles par des indices calculés. Les variables contenues dans un tableau, également appelées les ***éléments*** du tableau, sont du même type. Ce type est appelé le ***type d'élément*** du tableau.

Les types tableau sont des types référence, et la déclaration d'une variable tableau réserve simplement un espace pour une référence à une instance de tableau. Les instances de tableau réelles sont créées dynamiquement au moment de l'exécution à l'aide de l'opérateur `new`. L'opération `new` spécifie la ***longueur*** de la nouvelle instance de tableau, qui reste ensuite fixe pour la durée de vie de l'instance. Les indices des éléments d'un tableau vont de `0` à `Length - 1`. L'opérateur `new` initialise automatiquement les éléments d'un tableau à leur valeur par défaut, c'est-à-dire, par exemple, zéro pour tous les types numériques et `null` pour tous les types référence.

L'exemple suivant crée un tableau d'éléments `int`, initialise le tableau et imprime le contenu du tableau.

```
C#
```

```
int[] a = new int[10];
for (int i = 0; i < a.Length; i++)
{
    a[i] = i * i;
}
for (int i = 0; i < a.Length; i++)
{
    Console.WriteLine($"a[{i}] = {a[i]}");
}
```

Cet exemple crée et utilise un ***tableau unidimensionnel***. C# prend également en charge les ***tableaux multidimensionnels***. Le nombre de dimensions d'un type tableau, également appelé ***rang*** du type tableau, est de un plus le nombre de virgules entre les crochets du type tableau. L'exemple suivant alloue respectivement des tableaux à une seule, deux et trois dimensions, respectivement.

C#

```
int[] a1 = new int[10];
int[,] a2 = new int[10, 5];
int[,,] a3 = new int[10, 5, 2];
```

Le tableau `a1` contient 10 éléments, le tableau `a2` en contient 50 (10 x 5) et le tableau `a3` en contient 100 (10 × 5 × 2). Le type d'élément d'un tableau peut être de n'importe quel type, y compris un type tableau. Un tableau avec des éléments d'un type tableau est parfois appelé un ***tableau en escalier***, car les longueurs des tableaux d'éléments ne sont pas nécessairement les mêmes. L'exemple suivant alloue un tableau de tableaux de `int` :

C#

```
int[][] a = new int[3][];
a[0] = new int[10];
a[1] = new int[5];
a[2] = new int[20];
```

La première ligne crée un tableau avec trois éléments, chacun de type `int[]` et chacun avec une valeur initiale de `null`. Les lignes suivantes initialisent ensuite les trois éléments avec des références aux instances individuelles de tableau de longueurs variables.

L'opérateur `new` autorise la spécification des valeurs initiales des éléments du tableau en utilisant un ***initialiseur de tableau***, qui est une liste d'expressions écrites entre les délimiteurs `{` et `}`. L'exemple suivant alloue et initialise un `int[]` avec trois éléments.

C#

```
int[] a = new int[] { 1, 2, 3 };
```

La longueur du tableau est déduite du nombre d'expressions entre `{` et `}`.
L'initialisation du tableau peut être abrégée davantage afin de ne pas avoir à redéclarer le type tableau.

C#

```
int[] a = { 1, 2, 3 };
```

Les deux exemples précédents sont équivalents au code suivant :

C#

```
int[] t = new int[3];
t[0] = 1;
t[1] = 2;
t[2] = 3;
int[] a = t;
```

L'instruction `foreach` peut être utilisée pour énumérer les éléments de n'importe quelle collection. Le code suivant énumère le tableau de l'exemple précédent :

C#

```
foreach (int item in a)
{
    Console.WriteLine(item);
}
```

L'instruction `foreach` utilise l'interface IEnumerable<T>, afin qu'elle puisse fonctionner avec n'importe quelle collection.

Interpolation de chaîne

L'interpolation de chaîne C# vous permet de mettre en forme des chaînes en définissant des expressions dont les résultats sont placés dans une chaîne de format. Par exemple, l'exemple suivant imprime la température sur un jour donné à partir d'un ensemble de données météorologiques :

C#

```
Console.WriteLine($"The low and high temperature on {weatherData.Date:MM-dd-
yyyy}");
Console.WriteLine($"    was {weatherData.LowTemp} and
{weatherData.HighTemp}.");
// Output (similar to):
// The low and high temperature on 08-11-2020
//     was 5 and 30.
```

Une chaîne interpolée est déclarée à l'aide du jeton `$`. L'interpolation de chaîne évalue les expressions entre `{` et `}`, puis convertit le résultat en une `string` et remplace le texte entre les crochets par le résultat de la chaîne de l'expression. `:` dans la première expression, `{weatherData.Date:MM-dd-yyyy}` spécifie la *chaîne de format*. Dans l'exemple précédent, il spécifie que la date doit être imprimée au format « MM-jj-aaaa ».

Critères spéciaux

Le langage C# fournit des expressions de ***critères spéciaux*** pour interroger l'état d'un objet et exécuter du code en fonction de cet état. Vous pouvez inspecter les types et les valeurs des propriétés et des champs pour déterminer quelle action effectuer. Vous pouvez également inspecter les éléments d'une liste ou d'un tableau. L'expression `switch` est l'expression principale pour les critères spéciaux.

Délégués et expressions lambda

Un ***type délégué*** représente des références aux méthodes avec une liste de paramètres et un type de retour particuliers. Les délégués permettent de traiter les méthodes en tant qu'entités qui peuvent être affectées à des variables et passées comme paramètres. Les délégués sont similaires au concept de pointeurs de fonction trouvés dans d'autres langages. Contrairement aux pointeurs de fonction, les délégués sont orientés objet et de type sécurisé.

L'exemple suivant déclare et utilise un type délégué nommé `Function`.

C#

```
delegate double Function(double x);

class Multiplier
{
    double _factor;

    public Multiplier(double factor) => _factor = factor;

    public double Multiply(double x) => x * _factor;
```

```
}

class DelegateExample
{
    static double[] Apply(double[] a, Function f)
    {
        var result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }

    public static void Main()
    {
        double[] a = { 0.0, 0.5, 1.0 };
        double[] squares = Apply(a, (x) => x * x);
        double[] sines = Apply(a, Math.Sin);
        Multiplier m = new(2.0);
        double[] doubles = Apply(a, m.Multiply);
    }
}
```

Une instance du type délégué `Function` peut faire référence à toute méthode qui accepte un argument `double` et retourne une valeur `double`. La méthode `Apply` applique une `Function` donnée aux éléments d'un `double[]`, en retournant un `double[]` avec les résultats. Dans la méthode `Main`, `Apply` sert à appliquer trois fonctions différentes à un `double[]`.

Un délégué peut référencer une expression lambda pour créer une fonction anonyme (par exemple `(x) => x * x` dans l'exemple précédent), une méthode statique (par exemple `Math.Sin` dans l'exemple précédent) ou une méthode d'instance (comme `m.Multiply` dans l'exemple précédent). Un délégué qui référence une méthode d'instance référence aussi un objet particulier, et quand la méthode d'instance est appelée via le délégué, cet objet devient `this` dans l'appel.

Les délégués peuvent également être créés à l'aide de fonctions anonymes ou d'expressions lambda, qui sont des « méthodes inline » créées lors de la déclaration. Les fonctions anonymes peuvent voir les variables locales des méthodes qui l'entourent. L'exemple suivant ne crée pas de classe :

C#

```
double[] doubles = Apply(a, (double x) => x * 2.0);
```

Un délégué ne sait pas ou ne s'intéresse pas à la classe de la méthode qu'il référence. La méthode référencée doit avoir les mêmes paramètres et le même type de retour que le délégué.

async / await

C# prend en charge les programmes asynchrones avec deux mots clés : `async` et `await`. Vous ajoutez le modificateur `async` à une déclaration de méthode pour déclarer que la méthode est asynchrone. L'opérateur `await` indique au compilateur d'attendre de façon asynchrone qu'un résultat se termine. Le contrôle est retourné à l'appelant et la méthode retourne une structure qui gère l'état du travail asynchrone. La structure est généralement un System.Threading.Tasks.Task<TResult>, mais peut être n'importe quel type qui prend en charge le modèle awaiter. Ces fonctionnalités vous permettent d'écrire du code qui lit comme son équivalent synchrone, mais s'exécute de manière asynchrone. Par exemple, le code suivant télécharge la page d'accueil de Microsoft docs :

C#

```
public async Task<int> RetrieveDocsHomePage()
{
    var client = new HttpClient();
    byte[] content = await
client.GetByteArrayAsync("https://learn.microsoft.com/");

    Console.WriteLine($"{nameof(RetrieveDocsHomePage)}: Finished
downloading.");
    return content.Length;
}
```

Ce petit exemple montre les principales fonctionnalités de la programmation asynchrone :

- La déclaration de méthode inclut le modificateur `async`.
- Le corps de la méthode `await`s est le retour de la méthode `GetByteArrayAsync`.
- Le type spécifié dans l'instruction `return` correspond à l'argument de type dans la déclaration `Task<T>` de la méthode. (Une méthode qui retourne une `Task` utiliserait des instructions `return` sans argument).

Attributs

Les types, membres et autres entités d'un programme C# prennent en charge les modificateurs qui contrôlent certains aspects de leur comportement. Par exemple, l'accessibilité d'une méthode est contrôlée à l'aide des modificateurs `public`, `protected`, `internal` et `private`. C# généralise cette fonctionnalité pour que les types d'informations déclaratives définis par l'utilisateur puissent être associés aux entités de

programme et récupérés au moment de l'exécution. Les programmes spécifient ces informations déclaratives en définissant et en utilisant des ***attributs***.

L'exemple suivant déclare un attribut `HelpAttribute` qui peut être placé sur des entités de programme pour fournir des liens vers leur documentation associée.

C#

```
public class HelpAttribute : Attribute
{
    string _url;
    string _topic;

    public HelpAttribute(string url) => _url = url;

    public string Url => _url;

    public string Topic
    {
        get => _topic;
        set => _topic = value;
    }
}
```

Toutes les classes d'attributs dérivent de la classe de base Attribute fournie par la bibliothèque .NET. Les attributs peuvent être appliqués en donnant leur nom, ainsi que les éventuels arguments, à l'intérieur de crochets juste avant la déclaration associée. Si le nom d'un attribut se termine en `Attribute`, cette partie du nom peut être omise lorsque l'attribut est référencé. Par exemple, `HelpAttribute` peut être utilisé comme suit.

C#

```
[Help("https://learn.microsoft.com/dotnet/csharp/tour-of-csharp/features")]
public class Widget
{
    [Help("https://learn.microsoft.com/dotnet/csharp/tour-of-
csharp/features",
    Topic = "Display")]
    public void Display(string text) { }
}
```

Cet exemple joint un `HelpAttribute` à la classe `Widget`. Il ajoute un autre `HelpAttribute` à la méthode `Display` dans la classe. Les constructeurs publics d'une classe d'attributs contrôlent les informations qui doivent être fournies lorsque l'attribut est joint à une entité de programme. Des informations supplémentaires peuvent être fournies en référençant les propriétés en lecture-écriture publiques de la classe d'attributs (comme la référence à la propriété `Topic`, précédemment).

Les métadonnées définies par les attributs peuvent être lues et manipulées par réflexion lors de l'exécution. Lorsqu'un attribut particulier est demandé à l'aide de cette technique, le constructeur de la classe d'attributs est appelé avec les informations fournies dans la source du programme. L'instance d'attribut résultante est retournée. Si des informations supplémentaires ont été fournies via les propriétés, ces propriétés sont définies sur les valeurs données avant que le renvoi de l'instance de l'attribut.

L'exemple de code suivant montre comment obtenir les `HelpAttribute` instances associées à la classe `Widget` et sa méthode `Display`.

C#

```
Type widgetType = typeof(Widget);

object[] widgetClassAttributes =
widgetType.GetCustomAttributes(typeof(HelpAttribute), false);

if (widgetClassAttributes.Length > 0)
{
    HelpAttribute attr = (HelpAttribute)widgetClassAttributes[0];
    Console.WriteLine($"Widget class help URL : {attr.Url} - Related topic :
{attr.Topic}");
}

System.Reflection.MethodInfo displayMethod =
widgetType.GetMethod(nameof(Widget.Display));

object[] displayMethodAttributes =
displayMethod.GetCustomAttributes(typeof(HelpAttribute), false);

if (displayMethodAttributes.Length > 0)
{
    HelpAttribute attr = (HelpAttribute)displayMethodAttributes[0];
    Console.WriteLine($"Display method help URL : {attr.Url} - Related topic
: {attr.Topic}");
}
```

En savoir plus

Vous pouvez en savoir plus sur C# en essayant l'un de nos didacticiels.

Précédent

Présentation de C#

Article • 25/02/2023

Bienvenue dans les tutoriels de présentation de C#. Ces leçons commencent par du code interactif que vous pouvez exécuter dans votre navigateur. Vous pouvez apprendre les principes de base de C# à partir de la série vidéo C# 101 avant de commencer ces leçons interactives.
https://docs.microsoft.com/shows/CSharp-101/What-is-C/player

Les premières leçons expliquent les concepts de C# à l'aide de petits extraits de code. Vous apprendrez les bases de la syntaxe C# et comment utiliser des types de données tels que des chaînes, des nombres et des valeurs booléennes. Tout est interactif, vous allez écrire et exécuter du code en quelques minutes. Ces premières leçons prennent pour acquis que vous n'avez aucune connaissance préalable de la programmation ou du langage C#.

Vous pouvez essayer ces tutoriels dans différents environnements. Les concepts que vous allez apprendre sont les mêmes. La différence réside dans l'expérience que vous préférez :

- Dans votre navigateur, sur la plateforme des documents : cette expérience incorpore une fenêtre de code C# exécutable dans des pages de documents. Vous écrivez et exécutez du code C# dans le navigateur.
- Dans l'expérience Microsoft Learn. Ce parcours d'apprentissage contient plusieurs modules qui enseignent les principes de base de C#.
- Dans Jupyter sur Binder. Vous pouvez tester le code C# dans un notebook Jupyter sur Binder.
- Sur votre ordinateur local. Une fois que vous avez exploré en ligne, vous pouvez télécharger le SDK .NET et générer des programmes sur votre ordinateur.

Tous les tutoriels de présentation proposés après la leçon Hello World sont disponibles en ligne dans le navigateur ou dans votre propre environnement de développement local. À la fin de chaque tutoriel, vous décidez si vous souhaitez effectuer la leçon suivante en ligne ou sur votre propre ordinateur. Vous trouverez également des liens pour vous aider à configurer votre environnement et passer au tutoriel suivant sur votre ordinateur.

Hello world

Dans le tutoriel Hello world, vous allez créer le programme le plus basique en C#. Vous explorerez le type `string` et l'utilisation de texte. Vous pouvez également utiliser le

chemin sur Microsoft Learn ou Jupyter sur Binder.

Nombres en C#

Dans le tutoriel Nombres en C#, vous allez apprendre comment les ordinateurs stockent les nombres et comment effectuer des calculs avec différents types numériques. Vous apprendrez les bases de l'arrondi et comment effectuer des calculs mathématiques à l'aide de C#. Vous pouvez également exécuter ce tutoriel localement sur votre ordinateur.

Ce tutoriel part du principe que vous avez terminé la leçon Hello World.

Branches et boucles

Le tutoriel Branches et boucles explique les bases de la sélection de différents chemins d'exécution de code en fonction des valeurs stockées dans des variables. Vous allez apprendre les principes fondamentaux du flux de contrôle, sur la base duquel les programmes prennent des décisions et choisissent différentes actions. Vous pouvez également exécuter ce tutoriel localement sur votre ordinateur.

Ce tutoriel part du principe que vous avez terminé les leçons « Hello World » et « Nombres in C#.

Collection de listes

La leçon Collection de listes vous présente le type Collection de listes qui stocke les séquences de données. Vous apprendrez à ajouter et à supprimer des éléments, à rechercher des éléments et à trier des listes. Vous découvrirez différents types de listes. Vous pouvez également exécuter ce tutoriel localement sur votre ordinateur.

Ce didacticiel part du principe que vous avez terminé les leçons mentionnées ci-dessus.

101 échantillons LINQ

Cet exemple nécessite l'outil global dotnet-try. Une fois que vous avez installé l'outil et cloné le référentiel try-samples, vous pouvez découvrir la requête LINQ (Language Integrated Query) à travers un ensemble de 101 échantillons que vous pouvez exécuter de manière interactive. Vous pouvez explorer différentes façons d'interroger, d'explorer et de transformer des séquences de données.

Configurer votre environnement local

Article • 10/05/2023

La première étape pour exécuter un tutoriel sur votre ordinateur consiste à configurer un environnement de développement.

- Nous vous recommandons Visual Studio pour Windows ou Mac. Vous pouvez télécharger une version gratuite à partir de la page téléchargements de Visual Studio. Visual Studio inclut le Kit de développement logiciel (SDK) .NET.
- Vous pouvez également utiliser l'éditeur Visual Studio Code. Vous devez installer le dernier SDK .NET séparément.
- Si vous préférez un autre éditeur, vous devez installer le dernier SDK .NET.

Flux de développement d'une application de base

Les instructions de ces didacticiels supposent que vous utilisez l'interface CLI .NET pour créer, générer et exécuter des applications. Vous utiliserez la commande suivante :

- dotnet new crée une application. Cette commande génère les fichiers et ressources nécessaires pour votre application. Les tutoriels de présentation de C# utilisent tous le type d'application `console`. Une fois que vous avez les bases, vous pouvez développer d'autres types d'applications.
- dotnet build génère l'exécutable.
- dotnet run exécute l'exécutable.

Si vous utilisez Visual Studio 2019 pour ces didacticiels, vous choisirez une sélection de menu Visual Studio lorsqu'un didacticiel vous indique d'exécuter l'une de ces commandes CLI :

- **Fichier**>**Nouveau**>**Project** crée une application.
  - Le modèle de projet `Console Application` est recommandé.
  - Vous aurez la possibilité de spécifier une version cible de .Net Framework. Les didacticiels ci-dessous fonctionnent mieux lorsque vous ciblez .NET 5 ou une version ultérieure.
- **Générer**>**Générer la solution** génère l'exécutable.
- **Déboguer**>**Démarrer sans débogage** exécute l'exécutable.

Choisir un tutoriel

Vous pouvez commencer par l'un des tutoriels suivants :

Nombres en C#

Dans le tutoriel Nombres en C#, vous allez apprendre comment les ordinateurs stockent les nombres et comment effectuer des calculs avec différents types numériques. Vous apprendrez les bases de l'arrondi et comment effectuer des calculs mathématiques à l'aide de C#.

Ce tutoriel part du principe que vous avez terminé la leçon Hello world.

Branches et boucles

Le tutoriel Branches et boucles explique les bases de la sélection de différents chemins d'exécution de code en fonction des valeurs stockées dans des variables. Vous allez apprendre les principes fondamentaux du flux de contrôle, sur la base duquel les programmes prennent des décisions et choisissent différentes actions.

Ce tutoriel part du principe que vous avez fini les leçons Hello world et Nombres en C#.

Collection de listes

La leçon Collection de listes vous présente le type Collection de listes qui stocke les séquences de données. Vous apprendrez à ajouter et à supprimer des éléments, à rechercher des éléments et à trier des listes. Vous découvrirez différents types de listes.

Ce tutoriel part du principe que vous avez fini les leçons listées ci-dessus.

Comment utiliser des nombres entiers et à virgule flottante en langage C#

Article • 10/05/2023

Ce tutoriel permet de découvrir les types numériques en C#. Vous allez écrire de petites quantités de code, puis vous compilerez et exécuterez ce code. Ce tutoriel comporte une série de leçons visant à explorer les nombres et les opérations mathématiques en C#. Ces leçons présentent les concepts de base du langage C#.

> **Conseil**
>
> Pour coller un extrait de code dans le **mode focus**, vous devez utiliser le raccourci clavier (`Ctrl` + `v` ou `cmd` + `v`).

Prérequis

Le tutoriel suppose que vous avez un ordinateur configuré pour le développement local. Consultez Configurer votre environnement local pour obtenir des instructions sur l'installation et une vue d'ensemble du développement d'applications dans .NET.

Si vous ne souhaitez pas configurer un environnement local, consultez la version interactive dans le navigateur de ce tutoriel.

Explorer les mathématiques avec des entiers

Créez un répertoire nommé *numbers-quickstart*. Définissez le répertoire actif et exécutez la commande suivante :

CLI .NET

```
dotnet new console -n NumbersInCSharp -o .
```

> **Important**
>
> Les modèles C# pour .NET 6 utilisent des *instructions de niveau supérieur*. Votre application ne correspond peut-être pas au code de cet article, si vous avez déjà effectué une mise à niveau vers .NET 6. Pour plus d'informations, consultez l'article **Les nouveaux modèles C# génèrent des instructions de niveau supérieur**.

Le SDK .NET 6 ajoute également un ensemble de directives *implicites* `global using` pour les projets qui utilisent les kits SDK suivants :

- Microsoft.NET.Sdk
- Microsoft.NET.Sdk.Web
- Microsoft.NET.Sdk.Worker

Ces directives `global using` implicites incluent les espaces de noms les plus courants pour le type de projet.

Ouvrez *Program.cs* dans votre éditeur favori et remplacez le contenu du fichier par le code suivant :

C#

```
int a = 18;
int b = 6;
int c = a + b;
Console.WriteLine(c);
```

Exécutez ce code en tapant `dotnet run` dans votre fenêtre de commande.

Vous avez vu l'une des opérations mathématiques fondamentales avec des nombres entiers. Le type `int` représente un **nombre entier**, un zéro, ou un nombre positif ou négatif entier. Vous utilisez le symbole `+` pour effectuer une addition. Les autres opérations mathématiques courantes avec des entiers sont les suivantes :

- `-` pour la soustraction
- `*` pour la multiplication
- `/` pour la division

Commencez par explorer ces différentes opérations. Ajoutez ces lignes après la ligne qui écrit la valeur de `c` :

C#

```
// subtraction
c = a - b;
Console.WriteLine(c);

// multiplication
c = a * b;
Console.WriteLine(c);

// division
```

```
c = a / b;
Console.WriteLine(c);
```

Exécutez ce code en tapant `dotnet run` dans votre fenêtre de commande.

Vous pouvez également expérimenter en écrivant plusieurs opérations mathématiques dans la même ligne si vous le souhaitez. Essayez `c = a + b - 12 * 17;` par exemple. La combinaison de variables et de nombres constants est autorisée.

> **Conseil**
>
> Durant votre exploration de C# (ou de tout autre langage de programmation), vous commettrez des erreurs d'écriture du code. Le **compilateur** détectera ces erreurs et vous les signalera. Si la sortie contient des messages d'erreur, vérifiez attentivement l'exemple de code ainsi que le code dans votre fenêtre pour identifier les corrections à apporter. Cet exercice vous aidera à mieux comprendre la structure du code C#.

Vous avez terminé la première étape. Avant de commencer la section suivante, nous allons déplacer le code actuel dans une *méthode* distincte. Une méthode est une série d'instructions regroupées et nommées. Vous nommez une méthode en écrivant le nom de la méthode suivi de `()`. L'organisation de votre code en méthodes facilite l'utilisation d'un nouvel exemple. Lorsque vous avez terminé, votre code doit ressembler à ceci :

C#

```
WorkWithIntegers();

void WorkWithIntegers()
{
    int a = 18;
    int b = 6;
    int c = a + b;
    Console.WriteLine(c);


    // subtraction
    c = a - b;
    Console.WriteLine(c);

    // multiplication
    c = a * b;
    Console.WriteLine(c);

    // division
    c = a / b;
```

```
    Console.WriteLine(c);
}
```

La ligne `WorkWithIntegers();` appelle la méthode. Le code suivant déclare la méthode et la définit.

Explorer l'ordre des opérations

Commentez l'appel à `WorkingWithIntegers()`. La sortie est ainsi moins encombrée lorsque vous travaillez dans cette section :

C#

```
//WorkWithIntegers();
```

`//` démarre un **commentaire** dans C#. Les commentaires correspondent à du texte que vous voulez conserver dans votre code source sans l'exécuter en tant que code. Le compilateur ne génère aucun code exécutable à partir des commentaires. Étant donné que `WorkWithIntegers()` est d'une méthode, vous devez uniquement commenter une seule ligne.

Le langage C# définit la priorité des différentes opérations mathématiques avec à l'aide de règles cohérentes avec les règles mathématiques que vous avez apprises. La multiplication et la division ont priorité sur l'addition et la soustraction. Explorez cela en ajoutant le code suivant après l'appel et `WorkWithIntegers()`en exécutant `dotnet run`:

C#

```
int a = 5;
int b = 4;
int c = 2;
int d = a + b * c;
Console.WriteLine(d);
```

La sortie montre que la multiplication est effectuée avant l'addition.

Vous pouvez appliquer un ordre de calcul différent en mettant entre parenthèses la ou les opérations à exécuter en premier. Ajoutez les lignes suivantes et exécutez à nouveau :

C#

```
d = (a + b) * c;
Console.WriteLine(d);
```

Pratiquez en combinant plusieurs opérations différentes. Ajoutez quelque chose comme les lignes suivantes. Essayez `dotnet run` à nouveau.

C#

```
d = (a + b) - 6 * c + (12 * 4) / 3 + 12;
Console.WriteLine(d);
```

Vous avez peut-être observé un comportement intéressant des entiers. La division d'entiers produit toujours un résultat entier, même quand vous pensez que le résultat devrait inclure une partie décimale ou fractionnaire.

Si vous n'avez pas vu ce comportement, essayez le code suivant :

C#

```
int e = 7;
int f = 4;
int g = 3;
int h = (e + f) / g;
Console.WriteLine(h);
```

Tapez `dotnet run` à nouveau pour afficher les résultats.

Avant de continuer, prenez tout le code que vous avez écrit dans cette section et placez-le dans une nouvelle méthode. Nommez cette nouvelle méthode `OrderPrecedence`. Votre code doit ressembler à ceci :

C#

```
// WorkWithIntegers();
OrderPrecedence();

void WorkWithIntegers()
{
    int a = 18;
    int b = 6;
    int c = a + b;
    Console.WriteLine(c);


    // subtraction
    c = a - b;
    Console.WriteLine(c);
```

```
    // multiplication
    c = a * b;
    Console.WriteLine(c);

    // division
    c = a / b;
    Console.WriteLine(c);
}

void OrderPrecedence()
{
    int a = 5;
    int b = 4;
    int c = 2;
    int d = a + b * c;
    Console.WriteLine(d);

    d = (a + b) * c;
    Console.WriteLine(d);

    d = (a + b) - 6 * c + (12 * 4) / 3 + 12;
    Console.WriteLine(d);

    int e = 7;
    int f = 4;
    int g = 3;
    int h = (e + f) / g;
    Console.WriteLine(h);
}
```

Explorer la précision et les limites des entiers

Ce dernier exemple a montré que la division d'entiers tronque le résultat. Vous pouvez obtenir le **reste** à l'aide de l'opérateur **modulo**, à savoir le caractère `%`. Essayez le code suivant après l'appel de méthode à `OrderPrecedence()`:

C#

```
int a = 7;
int b = 4;
int c = 3;
int d = (a + b) / c;
int e = (a + b) % c;
Console.WriteLine($"quotient: {d}");
Console.WriteLine($"remainder: {e}");
```

Le type d'entier C# diffère des entiers mathématiques d'une autre manière : le type `int` a des limites minimale et maximale. Ajoutez ce code pour observer les limites :

C#

```
int max = int.MaxValue;
int min = int.MinValue;
Console.WriteLine($"The range of integers is {min} to {max}");
```

Si un calcul produit une valeur qui dépasse ces limites, vous obtenez une condition de **dépassement négatif** ou **dépassement positif**. La réponse affichée indique la plage (d'une limite à l'autre). Ajoutez ces deux lignes pour voir un exemple :

C#

```
int what = max + 3;
Console.WriteLine($"An example of overflow: {what}");
```

Notez que la réponse est très proche de l'entier minimal (négatif). Il en est de même pour `min + 2`. L'addition a produit un **dépassement négatif** des valeurs autorisées pour les entiers. La réponse est un très grand nombre négatif, car un dépassement négatif « inclut » de la plus grande valeur d'entier possible à la plus petite.

Il existe d'autres types numériques avec une précision et des limites différentes que vous pouvez utiliser quand le type `int` ne répond pas à vos besoins. Examinons ensuite ces autres types. Avant de commencer la section suivante, déplacez le code que vous avez écrit dans cette section dans une méthode distincte. Nommez-le `TestLimits`.

Utiliser le type double

Le type numérique `double` représente un nombre à virgule flottante double précision. Ces termes vous sont peut-être inconnus. Un nombre **à virgule flottante** est utile pour représenter de très grands ou petits nombres non intégraux. **Double précision** est un terme relatif qui décrit le nombre de chiffres binaires utilisés pour stocker la valeur. Les nombres de **double précision** ont deux fois le nombre de chiffres binaires d'une **simple précision**. Sur les ordinateurs modernes, il est plus courant d'utiliser une double précision que des nombres de simple précision. Les nombres de **simple précision** sont déclarés à l'aide du mot clé `float`. Explorons ce type double. Ajoutez le code suivant et affichez le résultat :

C#

```
double a = 5;
double b = 4;
double c = 2;
```

```
double d = (a + b) / c;
Console.WriteLine(d);
```

Notez que la réponse inclut la partie décimale du quotient. Essayez avec une expression légèrement plus complexe utilisant des doubles :

C#

```
double e = 19;
double f = 23;
double g = 8;
double h = (e + f) / g;
Console.WriteLine(h);
```

La plage d'une valeur double est nettement supérieure à celle de valeurs entières. Essayez le code suivant sous ce que vous avez écrit jusque-là :

C#

```
double max = double.MaxValue;
double min = double.MinValue;
Console.WriteLine($"The range of double is {min} to {max}");
```

Ces valeurs sont imprimées en notation scientifique. Le nombre à gauche du `E` est le nombre significatif. Le nombre à droite est l'exposant, en puissance de 10. Tout comme les nombres décimaux en mathématiques, les doubles en C# peuvent présenter des erreurs d'arrondi. Exécutez le code suivant :

C#

```
double third = 1.0 / 3.0;
Console.WriteLine(third);
```

Vous savez que `0.3` le nombre fini répété de fois n'est pas exactement la même chose que `1/3`.

Problème

Effectuez d'autres calculs avec des grands nombres, des petits nombres, des multiplications et des divisions à l'aide du type `double`. Effectuez des calculs plus complexes. Après avoir consacré plus de temps au test, placez le code que vous avez écrit dans une nouvelle méthode. Nommez cette nouvelle méthode `WorkWithDoubles`.

Utiliser des types décimaux

Vous avez vu les types numériques de base en C#, à savoir les entiers et les doubles. Il existe un autre type à apprendre : le type `decimal`. Le type `decimal` a une plage plus petite, mais une précision supérieure à celle du type `double`. Voyons voir :

C#

```
decimal min = decimal.MinValue;
decimal max = decimal.MaxValue;
Console.WriteLine($"The range of the decimal type is {min} to {max}");
```

Notez que la plage est plus petite que celle du type `double`. Vous pouvez observer la plus haute précision du type décimal en exécutant le code suivant :

C#

```
double a = 1.0;
double b = 3.0;
Console.WriteLine(a / b);

decimal c = 1.0M;
decimal d = 3.0M;
Console.WriteLine(c / d);
```

Le suffixe `M` des nombres permet d'indiquer comment une constante doit utiliser le type `decimal`. Sinon, le compilateur utilise le type `double`.

> ⓘ **Notes**
>
> La lettre `M` a été choisie comme lettre la plus distincte visuellement entre les mots clés et les mots clés `double` et `decimal`.

Notez que le calcul utilisant le type décimal a plus de chiffres à droite de la virgule décimale.

Problème

Maintenant que vous avez vu les différents types numériques, écrivez du code qui calcule la surface d'un cercle avec un rayon de 2,5 centimètres. Rappelez-vous que la surface d'un cercle est le rayon au carré multiplié par PI. Conseil : .NET contient une constante pour PI, à savoir Math.PI, que vous pouvez utiliser pour cette valeur. Math.PI, comme toutes les constantes déclarées dans l'espace de noms `System.Math`, est une

valeur `double`. Pour cette raison, vous devez utiliser `double` au lieu de `decimal` valeurs pour ce défi.

Vous devriez obtenir une réponse comprise entre 19 et 20. Vous pouvez vérifier votre réponse en consultant l'exemple de code terminé sur GitHub.

Si vous le voulez, essayez d'autres formules.

Vous avez terminé le guide de démarrage rapide « Nombres en C# ». Vous pouvez passer au guide de démarrage rapide Branches et boucles dans votre propre environnement de développement.

Pour plus d'informations sur les nombres en C#, consultez les article suivants :

- Types numériques intégraux
- Types numériques à virgule flottante
- Conversions numériques intégrées

Instructions et boucles `if` C# : didacticiel de logique conditionnelle

Article • 10/05/2023

Ce didacticiel explique comment écrire du code C# qui examine des variables et modifie le chemin d'exécution en fonction de ces variables. Vous allez écrire un code en C# et afficher les résultats de la compilation et de l'exécution du code. Ce tutoriel comporte une série de leçons visant à explorer les constructions de type branches et boucles en C#. Ces leçons présentent les concepts de base du langage C#.

> **Conseil**
>
> Pour coller un extrait de code dans le **mode focus**, vous devez utiliser le raccourci clavier (`Ctrl` + `v` ou `cmd` + `v`).

Prérequis

Le tutoriel suppose que vous avez un ordinateur configuré pour le développement local. Consultez Configurer votre environnement local pour obtenir des instructions d'installation et une vue d'ensemble du développement d'applications dans .NET.

Si vous préférez exécuter le code sans avoir à configurer un environnement local, consultez la version interactive de ce didacticiel dans le navigateur.

Prendre des décisions à l'aide de l'instruction `if`

Créez un répertoire nommé *branches-tutorial*. Indiquez le répertoire actuel et exécutez la commande suivante :

CLI .NET

```
dotnet new console -n BranchesAndLoops -o .
```

> **Important**

Les modèles C# pour .NET 6 utilisent des *instructions de niveau supérieur*. Votre application ne correspond peut-être pas au code de cet article, si vous avez déjà effectué une mise à niveau vers .NET 6. Pour plus d'informations, consultez l'article **Les nouveaux modèles C# génèrent des instructions de niveau supérieur**.

Le SDK .NET 6 ajoute également un ensemble de directives *implicites* `global using` pour les projets qui utilisent les kits SDK suivants :

- Microsoft.NET.Sdk
- Microsoft.NET.Sdk.Web
- Microsoft.NET.Sdk.Worker

Ces directives implicites `global using` incluent les espaces de noms les plus courants pour le type de projet.

Cette commande crée une nouvelle application console .NET dans le répertoire actuel. Ouvrez *Program.cs* dans votre éditeur favori, puis remplacez les contenus par le code qui suit :

C#

```
int a = 5;
int b = 6;
if (a + b > 10)
    Console.WriteLine("The answer is greater than 10.");
```

Essayez ce code en tapant `dotnet run` dans la fenêtre de console. Le message « la réponse est supérieure à 10 » doit s'afficher sur votre console. Modifiez la déclaration de `b` pour que la somme soit inférieure à 10 :

C#

```
int b = 3;
```

Tapez `dotnet run` à nouveau. La réponse étant inférieure à 10, rien ne s'affiche. La **condition** que vous testez a une valeur false. Vous n'avez pas de code à exécuter, car vous avez uniquement écrit l'une des branches possibles pour une instruction `if` : la branche true.

Conseil

Durant votre exploration de C# (ou de tout autre langage de programmation), vous commettrez des erreurs d'écriture du code. Le compilateur détectera et signalera les erreurs. Observez attentivement la sortie d'erreur et le code ayant généré l'erreur. L'erreur du compilateur peut en général vous aider à trouver le problème.

Le premier exemple montre la puissance de l'instruction `if` et des types booléens. Un *booléen* est une variable qui peut avoir l'une des deux valeurs suivantes : `true` ou `false`. C# définit un type spécial, `bool`, pour les variables booléennes. L'instruction `if` vérifie la valeur d'un `bool`. Quand la valeur est `true`, l'instruction qui suit `if` s'exécute. Autrement, il est ignoré. Ce processus de vérification des conditions et d'exécution des instructions en fonction de ces conditions est performant.

Utiliser if et else ensemble

Pour exécuter un code différent dans les branches true et false, vous créez une branche `else` qui s'exécute quand la condition a une valeur false. Essayez une branche `else`. Ajoutez les deux dernières lignes dans le code ci-dessous (vous devriez déjà avoir les quatre premières lignes) :

C#

```
int a = 5;
int b = 3;
if (a + b > 10)
    Console.WriteLine("The answer is greater than 10");
else
    Console.WriteLine("The answer is not greater than 10");
```

L'instruction qui suit le mot clé `else` s'exécute uniquement quand la condition testée a une valeur `false`. La combinaison de `if` et `else` avec des conditions booléennes offre les hautes performances dont vous avez besoin pour traiter une condition `true` et une condition `false`.

ⓘ Important

La mise en retrait sous les instructions `if` et `else` a simplement pour but de faciliter la lecture. Le langage C# ne considère pas la mise en retrait ou les espaces blancs comme des éléments significatifs. L'instruction qui suit le mot clé `if` ou `else` sera exécutée en fonction de la condition. Tous les exemples de ce tutoriel suivent une pratique courante qui consiste à mettre en retrait les lignes en fonction du flux de contrôle des instructions.

Étant donné que la mise en retrait n'est pas significative, vous devez utiliser `{` et `}` pour indiquer quand vous souhaitez inclure plus d'une instruction dans le bloc qui s'exécute de manière conditionnelle. Les programmeurs C# utilisent généralement les accolades pour toutes les clauses `if` et `else`. L'exemple suivant est identique à celui que vous avez créé. Modifiez votre code ci-dessus pour qu'il corresponde au code suivant :

C#

```
int a = 5;
int b = 3;
if (a + b > 10)
{
    Console.WriteLine("The answer is greater than 10");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
}
```

Conseil

Dans le reste de ce didacticiel, tous les exemples de code incluent les accolades, conformément aux pratiques acceptées.

Vous pouvez tester des conditions plus complexes. Ajoutez le code suivant après le code que avez écrit jusque-là :

C#

```
int c = 4;
if ((a + b + c > 10) && (a == b))
{
    Console.WriteLine("The answer is greater than 10");
    Console.WriteLine("And the first number is equal to the second");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
    Console.WriteLine("Or the first number is not equal to the second");
}
```

Le symbole `==` teste *l'égalité*. `==` se distingue du test d'égalité d'attribution, que nous avons vu avec `a = 5`.

`&&` représente « et ». Cela signifie que les deux conditions doivent être true pour que l'instruction s'exécute dans la branche true. Ces exemples montrent également que vous pouvez avoir plusieurs instructions dans chaque branche conditionnelle, à condition de les mettre entre `{` et `}`. Vous pouvez également utiliser `||` pour représenter « ou ». Ajoutez le code suivant après ce que vous avez écrit jusque-là :

C#

```
if ((a + b + c > 10) || (a == b))
{
    Console.WriteLine("The answer is greater than 10");
    Console.WriteLine("Or the first number is equal to the second");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
    Console.WriteLine("And the first number is not equal to the second");
}
```

Modifiez les valeurs de `a`, `b` et `c` et passez de `&&` à `||` et inversement. Vous comprendrez mieux comment fonctionnent les opérateurs `&&` et `||`.

Vous avez terminé la première étape. Avant de passer à la section suivante, déplaçons le code actuel dans une méthode distincte. Cela nous permettra de travailler plus facilement avec un nouvel exemple. Placez le code existant dans une méthode appelée `ExploreIf()`. Appelez-le à partir du haut de votre programme. Une fois ces modifications terminées, votre code doit ressembler à ce qui suit :

C#

```
ExploreIf();

void ExploreIf()
{
    int a = 5;
    int b = 3;
    if (a + b > 10)
    {
        Console.WriteLine("The answer is greater than 10");
    }
    else
    {
        Console.WriteLine("The answer is not greater than 10");
    }

    int c = 4;
    if ((a + b + c > 10) && (a > b))
    {
        Console.WriteLine("The answer is greater than 10");
```

```
        Console.WriteLine("And the first number is greater than the
second");
    }
    else
    {
        Console.WriteLine("The answer is not greater than 10");
        Console.WriteLine("Or the first number is not greater than the
second");
    }

    if ((a + b + c > 10) || (a > b))
    {
        Console.WriteLine("The answer is greater than 10");
        Console.WriteLine("Or the first number is greater than the second");
    }
    else
    {
        Console.WriteLine("The answer is not greater than 10");
        Console.WriteLine("And the first number is not greater than the
second");
    }
}
```

Commentez l'appel à `ExploreIf()`. La sortie est ainsi moins encombrée lorsque vous travaillez dans cette section :

C#

```
//ExploreIf();
```

`//` démarre un **commentaire** dans C#. Les commentaires correspondent à du texte que vous voulez conserver dans votre code source sans l'exécuter en tant que code. Le compilateur ne génère aucun code exécutable à partir des commentaires.

Utiliser des boucles pour répéter des opérations

Dans cette section, vous utilisez des **boucles** pour répéter des instructions. Ajoutez ce code après l'appel à `ExploreIf` :

C#

```
int counter = 0;
while (counter < 10)
{
    Console.WriteLine($"Hello World! The counter is {counter}");
```

```
    counter++;
}
```

L'instruction `while` vérifie une condition et exécute l'instruction ou le bloc d'instructions qui suit `while`. Elle répète la vérification de la condition, exécutant ces instructions jusqu'à ce que la condition soit false.

Cet exemple contient un nouvel opérateur. `++` après la variable `counter` est l'opérateur d'**incrémentation**. Il ajoute 1 à la valeur de `counter` et stocke cette valeur dans la variable de `counter`.

> ⓘ **Important**
>
> Assurez-vous que la condition de boucle `while` devient false quand vous exécutez le code. Dans le cas contraire, vous allez créer une **boucle infinie** dans laquelle votre programme ne se terminera jamais. Ceci n'est pas démontré dans cet exemple, car vous devez forcer la fermeture de votre programme avec **CTRL-C** ou à l'aide d'autres moyens.

La boucle `while` teste la condition avant d'exécuter le code qui suit `while`. La boucle `do ... while` exécute d'abord le code, puis vérifie la condition. La boucle *do while* est illustrée dans l'exemple de code suivant :

C#

```
int counter = 0;
do
{
    Console.WriteLine($"Hello World! The counter is {counter}");
    counter++;
} while (counter < 10);
```

Cette boucle `do` et la boucle antérieure `while` produisent la même sortie.

Utiliser la boucle for

La boucle **for** est couramment utilisée dans C#. Exécutez le code suivant :

C#

```
for (int index = 0; index < 10; index++)
{
```

```
    Console.WriteLine($"Hello World! The index is {index}");
}
```

Ce code précédent fonctionne de manière identique à la `while` boucle et à la `do` boucle que vous avez déjà utilisées. L'instruction `for` comprend trois parties qui contrôlent son fonctionnement.

La première partie est l'**initialiseur for** : `int index = 0;` déclare que `index` est la variable de boucle et définit sa valeur initiale sur `0`.

La partie centrale est la **condition for** : `index < 10` déclare que cette boucle `for` continue à s'exécuter tant que la valeur de compteur est inférieure à 10.

La dernière partie est l'**itérateur for** : `index++` indique comment modifier la variable de boucle après l'exécution du bloc qui suit l'instruction `for`. Il spécifie ici que `index` doit être incrémenté de 1 chaque fois que le bloc s'exécute.

Essayez vous-même. Essayez chacune des variantes suivantes :

- Modifiez l'initialiseur pour définir le démarrage à une valeur différente.
- Modifiez la condition pour définir l'arrêt à une valeur différente.

Une fois terminé, vous allez vous-même écrire des codes pour mettre en pratique ce que vous avez appris.

Il existe une autre instruction de bouclage qui n'est pas abordée dans ce tutoriel : l'instruction `foreach`. L'instruction `foreach` répète son instruction pour chaque élément d'une séquence d'éléments. Il est le plus souvent utilisé avec des *collections*, c'est pourquoi il est abordé dans le didacticiel suivant.

Création de boucles imbriquées

Une boucle `while`, `do` ou `for` peut être imbriquée dans une autre boucle pour créer une matrice à l'aide de la combinaison de chaque élément de la boucle externe avec chaque élément de la boucle interne. Faisons cela pour créer un ensemble de paires alphanumériques afin de représenter des lignes et des colonnes.

Une boucle `for` peut générer les lignes :

C#

```
for (int row = 1; row < 11; row++)
{
```

```
    Console.WriteLine($"The row is {row}");
}
```

Une autre boucle peut générer les colonnes :

C#

```
for (char column = 'a'; column < 'k'; column++)
{
    Console.WriteLine($"The column is {column}");
}
```

Vous pouvez imbriquer une boucle à l'intérieur de l'autre pour former des paires :

C#

```
for (int row = 1; row < 11; row++)
{
    for (char column = 'a'; column < 'k'; column++)
    {
        Console.WriteLine($"The cell is ({row}, {column})");
    }
}
```

Vous pouvez voir que la boucle externe s'incrémente une fois pour chaque exécution complète de la boucle interne. Inversez l'imbrication des lignes et des colonnes et observez les modifications. Lorsque vous avez terminé, placez le code de cette section dans une méthode appelée `ExploreLoops()`.

Combiner des branches et des boucles

Maintenant que vous avez vu l'instruction `if` et la création de boucles en langage C#, vérifiez si vous pouvez écrire un code C# pour obtenir la somme de tous les entiers de 1 à 20 divisibles par 3. Voici quelques conseils :

- L'opérateur `%` vous donne le reste d'une opération de division.
- L'instruction `if` vous donne la condition pour vérifier si un nombre doit être inclus dans la somme.
- La boucle `for` peut vous aider à répéter une série d'étapes pour tous les nombres de 1 à 20.

Essayez par vous-même et vérifiez le résultat. Vous devriez obtenir 63 comme réponse. Vous pouvez voir une réponse possible en consultant le code terminé sur GitHub.

Vous avez terminé le didacticiel « Branches et boucles ».

Vous pouvez passer au tutoriel Tableaux et collections dans votre propre environnement de développement.

Pour en savoir plus sur ces concepts, consultez les articles :

- Instructions de sélection
- Instructions d'itération

Découvrir comment gérer les collections de données avec List<T> dans C#

Article • 10/05/2023

Ce didacticiel propose une introduction au langage C# et présente les concepts de base de la classe List<T>.

Prérequis

Le tutoriel suppose que vous avez un ordinateur configuré pour le développement local. Consultez Configurer votre environnement local pour obtenir des instructions d'installation et une vue d'ensemble du développement d'applications dans .NET.

Si vous préférez exécuter le code sans avoir à configurer un environnement local, consultez la version navigateur interactive de ce tutoriel.

Exemple de liste de base

Créez un répertoire nommé *list-tutorial*. Faites-en le répertoire actuel et exécutez `dotnet new console`.

> **ⓘ Important**
>
> Les modèles C# pour .NET 6 utilisent des *instructions de niveau supérieur*. Votre application ne correspond peut-être pas au code de cet article, si vous avez déjà effectué une mise à niveau vers .NET 6. Pour plus d'informations, consultez l'article **Les nouveaux modèles C# génèrent des instructions de niveau supérieur**.
>
> Le SDK .NET 6 ajoute également un ensemble de directives *implicites* `global using` pour les projets qui utilisent les kits SDK suivants :
>
> - Microsoft.NET.Sdk
> - Microsoft.NET.Sdk.Web
> - Microsoft.NET.Sdk.Worker
>
> Ces directives `global using` implicites comprennent les espaces de noms les plus courants du type de projet.

Ouvrez *Program.cs* dans votre éditeur favori, puis remplacez le code existant par le suivant :

```
C#
```

```
var names = new List<string> { "<name>", "Ana", "Felipe" };
foreach (var name in names)
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}
```

Remplacez `<name>` par votre nom. Enregistrez *Program.cs*. Tapez `dotnet run` dans votre fenêtre de console pour effectuer un essai.

Vous avez créé une liste de chaînes, ajouté trois noms à cette liste et affiché les noms en majuscules. Vous utilisez des concepts que vous avez appris dans les tutoriels précédents pour lire la liste en boucle.

Le code permettant d'afficher les noms utilise la fonctionnalité d'interpolation de chaîne. Quand vous faites précéder une `string` du caractère `$`, vous pouvez incorporer le code C# dans la déclaration de chaîne. La chaîne réelle remplace ce code C# par la valeur qu'elle génère. Dans cet exemple, elle remplace `{name.ToUpper()}` par chaque nom, converti en majuscules, car vous avez appelé la méthode ToUpper.

Continuons notre exploration.

Modifier le contenu de la liste

La collection que vous avez créée utilise le type List<T>. Ce type stocke les séquences d'éléments. Vous spécifiez le type des éléments entre crochets angulaires.

Un aspect important du type List<T> est qu'il peut augmenter ou diminuer, ce qui vous permet d'ajouter ou de supprimer des éléments. Ajoutez ce code à la fin de votre programme :

```
C#
```

```
Console.WriteLine();
names.Add("Maria");
names.Add("Bill");
names.Remove("Ana");
foreach (var name in names)
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}
```

Vous avez ajouté deux noms supplémentaires à la fin de la liste. Vous en avez également supprimé un. Enregistrez le fichier et tapez `dotnet run` pour effectuer un essai.

La List<T> vous permet en outre de faire référence à des éléments individuels par **index**. Vous placez l'index entre les jetons `[` et `]` après le nom de la liste. C# utilise la valeur 0 pour le premier index. Ajoutez le code directement sous le code que vous venez d'ajouter et effectuez un essai :

C#

```
Console.WriteLine($"My name is {names[0]}");
Console.WriteLine($"I've added {names[2]} and {names[3]} to the list");
```

Vous ne pouvez pas accéder à un index au-delà de la fin de la liste. Rappelez-vous que les index commencent à partir de 0, de sorte que le plus grand index valide correspond au nombre d'éléments de la liste moins un. Vous pouvez vérifier la longueur de la liste à l'aide de la propriété Count. Ajoutez le code suivant à la fin de votre programme :

C#

```
Console.WriteLine($"The list has {names.Count} people in it");
```

Enregistrez le fichier et tapez de nouveau `dotnet run` pour afficher les résultats.

Trier les listes et y effectuer des recherches

Nos exemples utilisent des listes relativement petites, mais vos applications peuvent souvent générer des listes contenant beaucoup plus d'éléments, qui se comptent parfois même en milliers. Pour rechercher des éléments dans ces collections plus volumineuses, vous devez rechercher différents éléments dans la liste. La méthode IndexOf recherche un élément et retourne l'index de ce dernier. Si l'élément ne figure pas dans la liste, `IndexOf` retourne `-1`. Ajoutez ce code en bas de votre programme :

C#

```
var index = names.IndexOf("Felipe");
if (index == -1)
{
    Console.WriteLine($"When an item is not found, IndexOf returns
{index}");
}
else
{
    Console.WriteLine($"The name {names[index]} is at index {index}");
```

```
}

index = names.IndexOf("Not Found");
if (index == -1)
{
    Console.WriteLine($"When an item is not found, IndexOf returns
{index}");
}
else
{
    Console.WriteLine($"The name {names[index]} is at index {index}");

}
```

Il est également possible de trier les éléments de la liste. La méthode Sort trie tous les éléments de la liste dans l'ordre normal (par ordre alphabétique dans le cas de chaînes). Ajoutez ce code en bas de votre programme :

C#

```
names.Sort();
foreach (var name in names)
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}
```

Enregistrez le fichier et tapez `dotnet run` pour tester cette dernière version.

Avant de passer à la section suivante, déplaçons le code actuel dans une méthode distincte. Cela nous permettra de travailler plus facilement avec un nouvel exemple. Placez tout le code que vous avez écrit dans une nouvelle méthode appelée `WorkWithStrings()`. Appelez cette méthode en haut de votre programme. Lorsque vous avez terminé, votre code devrait ressembler à ceci :

C#

```
WorkWithStrings();

void WorkWithStrings()
{
    var names = new List<string> { "<name>", "Ana", "Felipe" };
    foreach (var name in names)
    {
        Console.WriteLine($"Hello {name.ToUpper()}!");
    }

    Console.WriteLine();
    names.Add("Maria");
    names.Add("Bill");
```

```
    names.Remove("Ana");
    foreach (var name in names)
    {
        Console.WriteLine($"Hello {name.ToUpper()}!");
    }

    Console.WriteLine($"My name is {names[0]}");
    Console.WriteLine($"I've added {names[2]} and {names[3]} to the list");

    Console.WriteLine($"The list has {names.Count} people in it");

    var index = names.IndexOf("Felipe");
    if (index == -1)
    {
        Console.WriteLine($"When an item is not found, IndexOf returns
{index}");
    }
    else
    {
        Console.WriteLine($"The name {names[index]} is at index {index}");
    }

    index = names.IndexOf("Not Found");
    if (index == -1)
    {
        Console.WriteLine($"When an item is not found, IndexOf returns
{index}");
    }
    else
    {
        Console.WriteLine($"The name {names[index]} is at index {index}");

    }

    names.Sort();
    foreach (var name in names)
    {
        Console.WriteLine($"Hello {name.ToUpper()}!");
    }
}
```

Listes d'autres types

Vous avez jusqu'à présent utilisé le type `string` dans les listes. Nous allons maintenant créer une List<T> en utilisant un autre type. Commençons par créer un ensemble de nombres.

Ajoutez ce qui suit à votre programme après avoir appelé `WorkWithStrings()` :

```
C#
```

```
var fibonacciNumbers = new List<int> {1, 1};
```

Cela crée une liste d'entiers et définit les deux premiers entiers sur la valeur 1. Il s'agit des deux premières valeurs d'une *séquence Fibonacci* (séquence de nombres). Chaque nombre Fibonacci suivant correspond à la somme des deux nombres précédents. Ajoutez le code suivant :

C#

```
var previous = fibonacciNumbers[fibonacciNumbers.Count - 1];
var previous2 = fibonacciNumbers[fibonacciNumbers.Count - 2];

fibonacciNumbers.Add(previous + previous2);

foreach (var item in fibonacciNumbers)
{
    Console.WriteLine(item);
}
```

Enregistrez le fichier et tapez `dotnet run` pour afficher les résultats.

> **Conseil**
>
> Pour vous concentrer uniquement sur cette section, vous pouvez commenter le code qui appelle `WorkWithStrings();`. Placez juste deux caractères `/` devant l'appel comme ceci : `// WorkWithStrings();`.

Problème

Vérifiez si vous pouvez mettre en pratique certains des concepts appris ici et dans d'autres leçons antérieures. Approfondissez à partir de ce que vous avez créé jusqu'à présent avec les nombres de Fibonacci. Essayez d'écrire le code pour générer les 20 premiers nombres de la séquence. (Astuce, le 20ème nombre Fibonacci est 6765.)

Terminer le test

Vous pouvez voir un exemple de solution en consultant l'exemple de code terminé sur GitHub.

À chaque itération de la boucle, vous sélectionnez les deux derniers entiers de la liste, les additionner et ajoutez la valeur obtenue à la liste. La boucle se répète jusqu'à ce que vous ayez ajouté 20 éléments à la liste.

Félicitations, vous avez terminé ce didacticiel sur les listes. Vous pouvez suivre d'autres tutoriels dans votre propre environnement de développement.

Vous pouvez en savoir plus sur l'utilisation du type `List` dans l'article des notions fondamentales de .NET sur les collections. Vous allez également en découvrir plus sur de nombreux autres types de collection.

Structure générale d'un programme C#

Article • 07/04/2023

Les programmes C# se composent d'un ou plusieurs fichiers. Chaque fichier contient zéro ou plusieurs espaces de noms. Un espace de noms contient des types tels que des classes, des structs, des interfaces, des énumérations et des délégués, ou d'autres espaces de noms. L'exemple suivant est le squelette d'un programme C# qui contient tous ces éléments.

C#

```
// A skeleton of a C# program
using System;

// Your program starts here:
Console.WriteLine("Hello world!");

namespace YourNamespace
{
    class YourClass
    {
    }

    struct YourStruct
    {
    }

    interface IYourInterface
    {
    }

    delegate int YourDelegate();

    enum YourEnum
    {
    }

    namespace YourNestedNamespace
    {
        struct YourStruct
        {
        }
    }
}
```

L'exemple précédent utilise des *instructions de niveau supérieur* pour le point d'entrée du programme. Cette fonctionnalité a été ajoutée dans C# 9. Avant C# 9, le point

d'entrée était une méthode statique nommée `Main`, comme illustré dans l'exemple suivant :

C#

```
// A skeleton of a C# program
using System;
namespace YourNamespace
{
    class YourClass
    {
    }

    struct YourStruct
    {
    }

    interface IYourInterface
    {
    }

    delegate int YourDelegate();

    enum YourEnum
    {
    }

    namespace YourNestedNamespace
    {
        struct YourStruct
        {
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            //Your program starts here...
            Console.WriteLine("Hello world!");
        }
    }
}
```

Sections connexes

Pour en savoir plus sur ces éléments de programme, consultez la section sur les types du guide Notions de base :

- Classes

- Structures
- Espaces de noms
- Interfaces
- Énumérations
- Délégués

Spécification du langage C#

Pour plus d'informations, consultez Concepts de base dans la Spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Main() et arguments de ligne de commande

Article • 10/05/2023

La méthode `Main` est le point d'entrée d'une application C#. (Les bibliothèques et les services ne nécessitent pas de méthode `Main` comme point d'entrée.) Lorsque l'application est démarrée, la méthode `Main` est la première méthode appelée.

Il ne peut y avoir qu'un seul point d'entrée dans un programme C#. Si plusieurs classes ont une méthode `Main`, vous devez compiler votre programme avec l'option de compilateur **StartupObject** pour spécifier quelle méthode `Main` utiliser comme point d'entrée. Pour plus d'informations, consultez **StartupObject** (Options du compilateur C#).

C#

```
class TestClass
{
    static void Main(string[] args)
    {
        // Display the number of command line arguments.
        Console.WriteLine(args.Length);
    }
}
```

À partir de C# 9, vous pouvez omettre la méthode `Main` et écrire des instructions C# comme si elles se trouvaient dans la méthode `Main`, comme dans l'exemple suivant :

C#

```
using System.Text;

StringBuilder builder = new();
builder.AppendLine("Hello");
builder.AppendLine("World!");

Console.WriteLine(builder.ToString());
```

Pour plus d'informations sur l'écriture du code d'application avec une méthode de point d'entrée implicite, consultez Instructions de niveau supérieur.

Vue d'ensemble

- La méthode `Main` est le point d'entrée d'un programme exécutable ; c'est l'endroit où le contrôle du programme commence et se termine.
- `Main` est déclaré à l'intérieur d'une classe ou d'un struct. `Main` doit être static et ne doit pas être public. (Dans l'exemple précédent, il reçoit l'accès par défaut de private.) La classe ou le struct englobant n'a pas besoin d'être statique.
- `Main` peut avoir le type de retour `void`, `int`, `Task`, ou `Task<int>`.
- Si et seulement si `Main` retourne un `Task` ou `Task<int>`, la déclaration de `Main` peut inclure le modificateur async. Cela exclut spécifiquement une méthode `async void Main`.
- La méthode `Main` peut être déclarée avec ou sans paramètre `string[]`, qui contient des arguments de ligne de commande. Lorsque vous utilisez Visual Studio pour créer des applications Windows, vous pouvez ajouter le paramètre manuellement ou bien utiliser la méthode GetCommandLineArgs() pour obtenir les arguments de ligne de commande. Les paramètres sont lus comme des arguments de ligne de commande avec index de base zéro. Contrairement à C et C++, le nom du programme n'est pas traité comme le premier argument de ligne de commande du tableau `args`, mais il s'agit du premier élément de la méthode GetCommandLineArgs().

La liste suivante affiche les signatures `Main` valides :

C#

```
public static void Main() { }
public static int Main() { }
public static void Main(string[] args) { }
public static int Main(string[] args) { }
public static async Task Main() { }
public static async Task<int> Main() { }
public static async Task Main(string[] args) { }
public static async Task<int> Main(string[] args) { }
```

Les exemples précédents utilisent tous le modificateur d'accesseur `public`. C'est commun, mais pas obligatoire.

L'ajout des types de retour `async` et `Task`, `Task<int>` simplifie le code de programme lorsque les applications de console doivent démarrer et `await` des opérations asynchrones dans `Main`.

Valeurs de retour Main()

Vous pouvez renvoyer un `int` à la méthode `Main` en définissant la méthode de l'une des manières suivantes :

Code de la méthode `Main`	**Signature `Main`**
Aucune utilisation de `args` ou `await`	`static int Main()`
Utilise `args`, aucune utilisation de `await`	`static int Main(string[] args)`
Aucune utilisation de `args`, utilise `await`	`static async Task<int> Main()`
Utilise `args` et `await`	`static async Task<int> Main(string[] args)`

Si la valeur de retour de `Main` n'est pas utilisée, retourner `void` ou `Task` permet d'avoir un code un peu plus simple.

Code de la méthode `Main`	**Signature `Main`**
Aucune utilisation de `args` ou `await`	`static void Main()`
Utilise `args`, aucune utilisation de `await`	`static void Main(string[] args)`
Aucune utilisation de `args`, utilise `await`	`static async Task Main()`
Utilise `args` et `await`	`static async Task Main(string[] args)`

Cependant, retourner `int` ou `Task<int>` permet au programme de communiquer des informations d'état à d'autres programmes ou scripts qui appellent le fichier exécutable.

L'exemple suivant montre comment accéder au code de sortie du processus.

Cet exemple utilise les outils de ligne de commande .NET Core. Si vous ne connaissez pas les outils de ligne de commande .NET Core, vous pouvez les découvrir dans cet article Bien démarrer.

Créez une application en exécutant `dotnet new console`. Modifiez la méthode `Main` dans *Program.cs* comme suit :

C#

```
// Save this program as MainReturnValTest.cs.
class MainReturnValTest
{
    static int Main()
    {
        //...
        return 0;
```

```
    }
}
```

Quand un programme est exécuté dans Windows, toute valeur retournée par la fonction `Main` est stockée dans une variable d'environnement. Cette variable d'environnement peut être récupérée à l'aide de `ERRORLEVEL` dans un fichier de commandes ou de `$LastExitCode` dans PowerShell.

Vous pouvez générer l'application à l'aide de la commande dotnet CLI `dotnet build`.

Ensuite, créez un script PowerShell pour exécuter l'application et afficher le résultat. Collez le code suivant dans un fichier texte et enregistrez-le sous `test.ps1` dans le dossier qui contient le projet. Exécutez le script PowerShell en tapant `test.ps1` à l'invite de PowerShell.

Comme le code retourne zéro, le fichier de commandes indique la réussite. Toutefois, si vous modifiez MainReturnValTest.cs pour qu'il retourne une valeur différente de zéro, puis que vous recompilez le programme, l'exécution suivante du script PowerShell indique un échec.

PowerShell

```
dotnet run
if ($LastExitCode -eq 0) {
    Write-Host "Execution succeeded"
} else
{
    Write-Host "Execution Failed"
}
Write-Host "Return value = " $LastExitCode
```

Sortie

```
Execution succeeded
Return value = 0
```

Valeurs de retour d'Async Main

Lorsque vous déclarez une `async` valeur renvoyée pour `Main`, le compilateur génère le code réutilisable pour appeler des méthodes asynchrones dans `Main`. Si vous ne spécifiez pas le mot clé `async`, vous devez écrire ce code vous-même, comme illustré dans l'exemple suivant. Le code de l'exemple garantit que votre programme s'exécute jusqu'à ce que l'opération asynchrone soit terminée :

C#

```
public static void Main()
{
    AsyncConsoleWork().GetAwaiter().GetResult();
}

private static async Task<int> AsyncConsoleWork()
{
    // Main body here
    return 0;
}
```

Ce code réutilisable peut être remplacé par :

C#

```
static async Task<int> Main(string[] args)
{
    return await AsyncConsoleWork();
}
```

L'avantage de déclarer `Main` comme `async` est que le compilateur génère toujours un code correct.

Quand le point d'entrée de l'application retourne `Task` ou `Task<int>`, le compilateur génère un nouveau point d'entrée qui appelle la méthode de point d'entrée déclarée dans le code d'application. En supposant que ce point d'entrée est appelé `$GeneratedMain`, le compilateur génère le code suivant pour ces points d'entrée :

- `static Task Main()` permet au compilateur d'émettre l'équivalent de `private static void $GeneratedMain() => Main().GetAwaiter().GetResult();`
- `static Task Main(string[])` permet au compilateur d'émettre l'équivalent de `private static void $GeneratedMain(string[] args) => Main(args).GetAwaiter().GetResult();`
- `static Task<int> Main()` permet au compilateur d'émettre l'équivalent de `private static int $GeneratedMain() => Main().GetAwaiter().GetResult();`
- `static Task<int> Main(string[])` permet au compilateur d'émettre l'équivalent de `private static int $GeneratedMain(string[] args) => Main(args).GetAwaiter().GetResult();`

ⓘ Notes

Si les exemples ont utilisé le modificateur `async` sur la méthode `Main`, le compilateur génère le même code.

Arguments de ligne de commande

Vous pouvez envoyer des arguments à la méthode `Main` en définissant la méthode de l'une des manières suivantes :

Code de la méthode `Main`	**Signature `Main`**
Aucune valeur renvoyée, aucune utilisation de `await`	`static void Main(string[] args)`
Valeur renvoyée, aucune utilisation de `await`	`static int Main(string[] args)`
Aucune valeur renvoyée, utilise `await`	`static async Task Main(string[] args)`
Valeur renvoyée, utilise `await`	`static async Task<int> Main(string[] args)`

Si les arguments ne sont pas utilisés, vous pouvez omettre `args` de la signature de méthode pour du code légèrement plus simple :

Code de la méthode `Main`	**Signature `Main`**
Aucune valeur renvoyée, aucune utilisation de `await`	`static void Main()`
Valeur renvoyée, aucune utilisation de `await`	`static int Main()`
Aucune valeur renvoyée, utilise `await`	`static async Task Main()`
Valeur renvoyée, utilise `await`	`static async Task<int> Main()`

ⓘ Notes

Vous pouvez également utiliser **Environment.CommandLine** ou **Environment.GetCommandLineArgs** pour accéder aux arguments de ligne de commande à partir de n'importe quel emplacement d'une application console ou Windows Forms. Pour activer des arguments de ligne de commande dans la méthode de signature `Main` dans une application Windows Forms, vous devez modifier manuellement la signature de `Main`. Le code généré par Windows Forms crée `Main` sans paramètre d'entrée.

Le paramètre de la méthode `Main` est un tableau String qui représente les arguments de ligne de commande. En général, vous déterminez s'il existe des arguments en testant la propriété `Length`, par exemple :

```
C#

if (args.Length == 0)
{
    System.Console.WriteLine("Please enter a numeric argument.");
    return 1;
}
```

> **Conseil**
>
> Le tableau `args` ne peut pas être null. Par conséquent, il est sûr d'accéder à la propriété `Length` sans vérification de la valeur Null.

Vous pouvez également convertir les arguments de chaîne en types numériques à l'aide de la classe Convert ou de la méthode `Parse`. Par exemple, l'instruction suivante convertit `string` en nombre `long` en utilisant la méthode Parse :

```
C#

long num = Int64.Parse(args[0]);
```

Il est également possible d'utiliser le type C# `long`, qui sert d'alias à `Int64` :

```
C#

long num = long.Parse(args[0]);
```

Vous pouvez également utiliser la méthode `ToInt64` de la classe `Convert` pour obtenir le même résultat :

```
C#

long num = Convert.ToInt64(s);
```

Pour plus d'informations, consultez Parse et Convert.

L'exemple suivant montre comment utiliser des arguments de ligne de commande dans une application console. L'application prend un argument au moment de l'exécution, le

convertit en entier, puis calcule la factorielle du nombre. Si aucun argument n'est fourni, l'application affiche un message pour expliquer comment le programme doit être utilisé.

Pour compiler et exécuter l'application à partir d'une invite de commandes, procédez comme suit :

1. Collez le code suivant dans un éditeur de texte, puis enregistrez le fichier en tant que fichier texte sous le nom *Factorial.cs*.

C#

```
public class Functions
{
    public static long Factorial(int n)
    {
        // Test for invalid input.
        if ((n < 0) || (n > 20))
        {
            return -1;
        }

        // Calculate the factorial iteratively rather than recursively.
        long tempResult = 1;
        for (int i = 1; i <= n; i++)
        {
            tempResult *= i;
        }
        return tempResult;
    }
}

class MainClass
{
    static int Main(string[] args)
    {
        // Test if input arguments were supplied.
        if (args.Length == 0)
        {
            Console.WriteLine("Please enter a numeric argument.");
            Console.WriteLine("Usage: Factorial <num>");
            return 1;
        }

        // Try to convert the input arguments to numbers. This will
throw
        // an exception if the argument is not a number.
        // num = int.Parse(args[0]);
        int num;
        bool test = int.TryParse(args[0], out num);
        if (!test)
        {
            Console.WriteLine("Please enter a numeric argument.");
            Console.WriteLine("Usage: Factorial <num>");
```

```
            return 1;
        }

        // Calculate factorial.
        long result = Functions.Factorial(num);

        // Print result.
        if (result == -1)
            Console.WriteLine("Input must be >= 0 and <= 20.");
        else
            Console.WriteLine($"The Factorial of {num} is {result}.");

        return 0;
    }
}
// If 3 is entered on command line, the
// output reads: The factorial of 3 is 6.
```

2. Dans l'écran **Démarrer** ou le menu **Démarrer**, ouvrez une fenêtre **Invite de commandes développeur** Visual Studio, puis accédez au dossier qui contient le fichier que vous avez créé.

3. Entrez la commande suivante pour compiler l'application.

   `dotnet build`

   Si votre application ne contient pas d'erreurs de compilation, un fichier exécutable nommé *Factorial.exe* est créé.

4. Entrez la commande suivante pour calculer la factorielle de 3 :

   `dotnet run -- 3`

5. Cette commande génère la sortie suivante : `The factorial of 3 is 6.`

> ⓘ Notes
>
> Quand vous exécutez une application dans Visual Studio, vous pouvez spécifier des arguments de ligne de commande dans la **page Déboguer du Concepteur de projet**.

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- System.Environment
- Comment afficher les arguments de ligne de commande

Instructions de niveau supérieur – programmes sans méthodes `Main`

Article • 07/04/2023

À compter de C# 9, vous n'avez pas besoin d'inclure explicitement une méthode `Main` dans un projet d'application console. Au lieu de cela, vous pouvez utiliser la fonctionnalité d'*instructions de niveau supérieur* pour réduire le volume de code que vous devez écrire. Dans ce cas, le compilateur génère une classe un point d'entrée de méthode `Main` pour l'application.

Voici un *fichier Program.cs* qui est un programme C# complet en C# 10 :

```
C#

Console.WriteLine("Hello World!");
```

Les instructions de niveau supérieur vous permettent d'écrire des programmes simples pour de petits utilitaires tels qu'Azure Functions et GitHub Actions. Elles simplifient également l'apprentissage et l'écriture du code pour les nouveaux programmeurs C#.

Les sections suivantes expliquent les règles concernant ce que vous pouvez et ne pouvez pas faire avec les instructions de niveau supérieur.

Un seul fichier de niveau supérieur

Une application doit avoir un seul point d'entrée. Un projet ne peut avoir qu'un fichier avec des instructions de niveau supérieur. Le fait de placer des instructions de niveau supérieur dans plusieurs fichiers d'un projet entraîne l'erreur du compilateur suivante :

> CS8802 Une seule unité de compilation peut avoir des instructions de niveau supérieur.

Un projet peut comporter un nombre quelconque de fichiers de code source supplémentaires qui ne possèdent pas de déclarations de niveau supérieur.

Aucun autre point d'entrée

Vous pouvez écrire une méthode `Main` explicitement, mais elle ne peut pas fonctionner comme point d'entrée. Le compilateur émet l'avertissement suivant :

> CS7022 Le point d'entrée du programme est le code global, en ignorant le point d'entrée « Main() ».

Dans un projet comportant des instructions de niveau supérieur, vous ne pouvez pas utiliser l'option de compilateur -main pour sélectionner le point d'entrée, même si le projet comporte une ou plusieurs méthodes `Main`.

Directives `using`

Si vous incluez des directives using, elles doivent figurer en premier dans le fichier, comme dans cet exemple :

C#

```
using System.Text;

StringBuilder builder = new();
builder.AppendLine("Hello");
builder.AppendLine("World!");

Console.WriteLine(builder.ToString());
```

Espace de noms global

Les instructions de niveau supérieur figurent implicitement dans l'espace de noms global.

Espaces de noms et définitions de type

Un fichier avec des instructions de niveau supérieur peut également contenir des espaces de noms et des définitions de type, mais, dans ce cas, ceux-ci doivent figurer après les instructions de niveau supérieur. Par exemple :

C#

```
MyClass.TestMethod();
MyNamespace.MyClass.MyMethod();

public class MyClass
{
    public static void TestMethod()
    {
```

```
        Console.WriteLine("Hello World!");
    }

}

namespace MyNamespace
{
    class MyClass
    {
        public static void MyMethod()
        {
            Console.WriteLine("Hello World from
MyNamespace.MyClass.MyMethod!");
        }
    }
}
```

`args`

Les instructions de niveau supérieur peuvent référencer la variable `args` afin d'accéder à tous les arguments de ligne de commande entrés. La variable `args` n'est jamais null, mais son argument `Length` est égal à zéro si aucun argument de ligne de commande n'a été fourni. Par exemple :

C#

```
if (args.Length > 0)
{
    foreach (var arg in args)
    {
        Console.WriteLine($"Argument={arg}");
    }
}
else
{
    Console.WriteLine("No arguments");
}
```

`await`

Vous pouvez appeler une méthode asynchrone à l'aide de `await`. Par exemple :

C#

```
Console.Write("Hello ");
await Task.Delay(5000);
Console.WriteLine("World!");
```

Code de sortie du processus

Pour retourner une valeur `int` lorsque l'application s'achève, utilisez l'instruction `return` comme vous le feriez dans une méthode `Main` qui retourne un `int`. Par exemple :

C#

```
string? s = Console.ReadLine();

int returnValue = int.Parse(s ?? "-1");
return returnValue;
```

Méthode de point d'entrée implicite

Le compilateur génère une méthode pour servir de point d'entrée de programme pour un projet comportant des instructions de niveau supérieur. Le nom de cette méthode n'est pas réellement `Main`, il s'agit d'un détail d'implémentation que votre code ne peut pas référencer directement. La signature de la méthode varie selon que les instructions de niveau supérieur contiennent le mot clé `await` ou l'instruction `return`. Le tableau suivant montre à quoi ressemblerait la signature de méthode, en utilisant le nom `Main` de la méthode dans la table pour des raisons pratiques.

Le code de niveau supérieur contient	**Signature implicite `Main`**
`await` et `return`	`static async Task<int> Main(string[] args)`
`await`	`static async Task Main(string[] args)`
`return`	`static int Main(string[] args)`
Aucun `await` ou `return`	`static void Main(string[] args)`

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Spécification des fonctionnalités – Instructions de niveau supérieur

Système de types C#

Article • 25/02/2023

C# est un langage fortement typé. Chaque variable et chaque constante ont un type, tout comme chaque expression qui fournit une valeur. Chaque déclaration de méthode spécifie un nom, le type et le genre (valeur, référence ou sortie) pour chaque paramètre d'entrée et pour la valeur de retour. La bibliothèque de classes .NET définit des types numériques intégrés et des types complexes qui représentent un large éventail de constructions. Il s'agit notamment du système de fichiers, des connexions réseau, des collections et des tableaux d'objets et des dates. Un programme C# classique utilise des types de la bibliothèque de classes et des types définis par l'utilisateur qui modélisent les concepts propres au domaine du problème du programme.

Les informations stockées dans un type peuvent inclure les éléments suivants :

- Espace de stockage nécessaire pour une variable du type.
- Valeurs minimale et maximale que le type peut représenter.
- Membres (méthodes, champs, événements, etc.) que le type contient.
- Type de base dont le type est hérité.
- Interface(s) qu'il implémente.
- Sortes d'opérations autorisées.

Le compilateur utilise les informations de type pour s'assurer que toutes les opérations qui sont effectuées dans votre code sont de *type safe*. Par exemple, si vous déclarez une variable de type int, le compilateur vous permet d'utiliser la variable dans des opérations d'addition et de soustraction. Si vous essayez d'effectuer ces mêmes opérations avec une variable de type bool, le compilateur génère une erreur, comme illustré dans l'exemple suivant :

C#

```
int a = 5;
int b = a + 2; //OK

bool test = true;

// Error. Operator '+' cannot be applied to operands of type 'int' and
'bool'.
int c = a + test;
```

ⓘ Notes

Développeurs C et C++ : notez que dans C#, `bool` n'est pas convertible en `int`.

Le compilateur incorpore les informations de type dans le fichier exécutable sous forme de métadonnées. Le common language runtime (CLR) utilise ces métadonnées au moment de l'exécution pour garantir que le type est sécurisé lorsqu'il alloue et libère de la mémoire.

Spécification de types dans les déclarations de variable

Lorsque vous déclarez une variable ou une constante dans un programme, vous devez spécifier son type ou utiliser le mot clé var pour permettre au compilateur de déduire le type. L'exemple suivant montre des déclarations de variable qui utilisent des types numériques intégrés et des types complexes définis par l'utilisateur :

C#

```
// Declaration only:
float temperature;
string name;
MyClass myClass;

// Declaration with initializers (four examples):
char firstLetter = 'C';
var limit = 3;
int[] source = { 0, 1, 2, 3, 4, 5 };
var query = from item in source
            where item <= limit
            select item;
```

Les types de paramètres de méthode et de valeurs renvoyée sont spécifiés dans la déclaration de méthode. La signature suivante présente une méthode qui requiert `int` comme argument d'entrée et retourne une chaîne :

C#

```
public string GetName(int ID)
{
    if (ID < names.Length)
        return names[ID];
    else
        return String.Empty;
}
private string[] names = { "Spencer", "Sally", "Doug" };
```

Après avoir déclaré une variable, vous ne pouvez pas la redéclarer avec un nouveau type et vous ne pouvez pas affecter une valeur non compatible avec son type déclaré. Par exemple, vous ne pouvez pas déclarer une variable de type `int` et lui assigner la valeur booléenne `true`. Toutefois, les valeurs peuvent être converties en d'autres types, par exemple quand elles sont assignées à de nouvelles variables ou passées en tant qu'arguments de méthode. Une *conversion de type* qui ne cause pas de perte de données est effectuée automatiquement par le compilateur. Une conversion susceptible de causer la perte de données exige une variable *cast* dans le code source.

Pour plus d'informations, consultez Cast et conversions de types.

Types intégrés

C# fournit un ensemble standard de types intégrés. Ils représentent des entiers, des valeurs à virgule flottante, des expressions booléennes, des caractères de texte, des valeurs décimales et d'autres types de données. Il existe également des types `string` et `object` intégrés. Ces types peuvent être utilisés dans n'importe quel programme C#. Pour obtenir la liste complète des types intégrés, consultez Types intégrés.

Types personnalisés

Vous utilisez les constructions struct, class, interface, enum, et record pour créer vos propres types personnalisés. La bibliothèque de classes .NET est une collection de types personnalisés que vous pouvez utiliser dans vos propres applications. Par défaut, les types les plus fréquemment utilisés dans la bibliothèque de classes sont disponibles dans tous les programmes C#. Les autres types sont disponibles uniquement si vous ajoutez explicitement une référence de projet à l'assembly qui les définit. Une fois que le compilateur a une référence à l'assembly, vous pouvez déclarer des variables (et des constantes) des types déclarés dans cet assembly dans le code source. Pour plus d'informations, consultez Bibliothèque de classes .NET.

Système de type commun

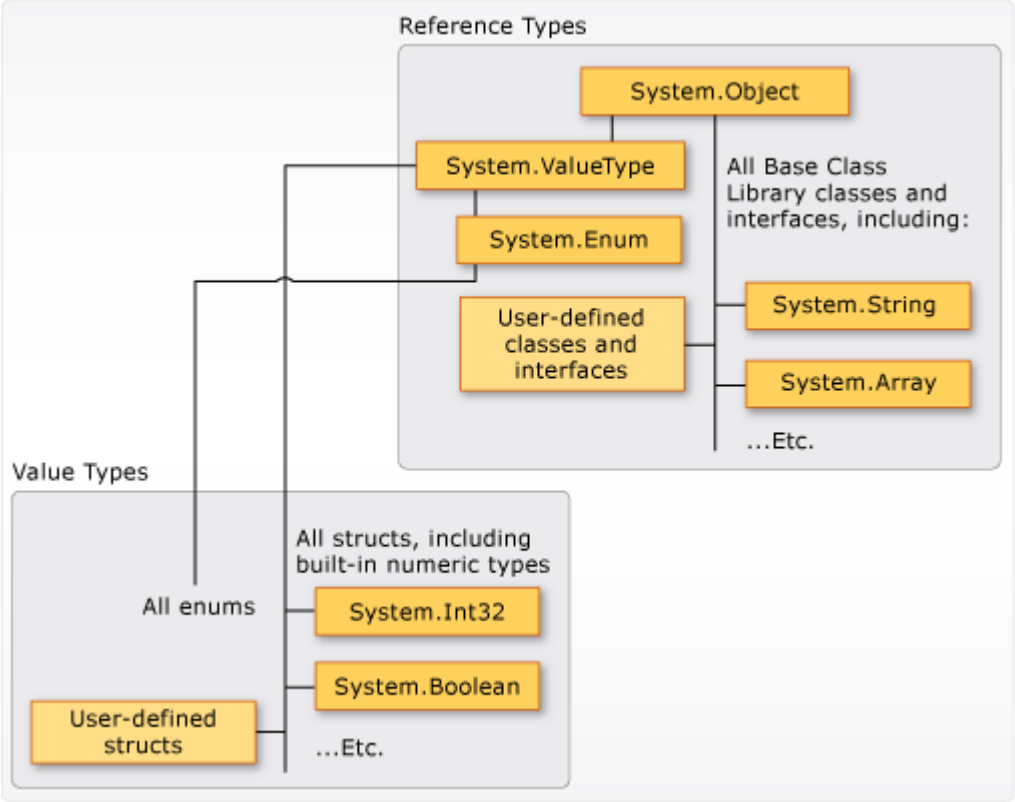
Il est important de comprendre deux points fondamentaux concernant le système de type dans .NET :

- Il prend en charge le principe d'héritage. Les types peuvent dériver d'autres types, appelés *types de base*. Le type dérivé hérite (avec certaines restrictions) des méthodes, des propriétés et des autres membres du type de base. Le type de base peut à son tour dériver d'un autre type, auquel cas le type dérivé hérite des

membres des deux types de base dans sa hiérarchie d'héritage. Tous les types, y compris les types numériques intégrés tels que System.Int32 (mot clé C# : `int`), dérivent au final d'un seul type de base, qui est System.Object (mot clé C# : object). Cette hiérarchie de types unifiée est appelée Système de type commun (CTS). Pour plus d'informations sur l'héritage dans C#, consultez Héritage.

- Chaque type du CTS est défini comme *type valeur* ou *type référence*. Cela inclut tous les types personnalisés dans la bibliothèque de classes .NET, ainsi que les types définis par l'utilisateur. Les types que vous définissez à l'aide du mot clé `struct` sont des types valeur ; tous les types numériques intégrés sont `structs`. Les types que vous définissez à l'aide du mot clé `class` ou `record` sont des types référence. Les types référence et les types valeur ne suivent pas les mêmes règles de compilation et ont un comportement différent au moment de l'exécution.

L'illustration suivante montre la relation entre les types valeur et les types référence dans le CTS.



> ⓘ **Notes**
>
> Vous pouvez voir que les types couramment utilisés sont tous organisés dans l'espace de noms **System**. Toutefois, l'espace de noms qui contient un type n'a aucune incidence sur le fait qu'il s'agisse d'un type valeur ou d'un type référence.

Les classes et les structs sont deux des constructions de base du système de type commun dans .NET. C# 9 ajoute des enregistrements, qui sont une sorte de classe.

Chacun est en substance une structure de données qui encapsule un ensemble de données et de comportements constituant une unité logique. Les données et les comportements sont les *membres* de la classe, du struct ou de l'enregistrement. Les membres incluent ses méthodes, propriétés, événements, etc., comme indiqué plus loin dans cet article.

Une déclaration de classe, de structure ou d'enregistrement est semblable à un plan utilisé pour créer des instances ou des objets au moment de l'exécution. Si vous définissez une classe, un struct ou un enregistrement nommé `Person`, `Person` est le nom du type. Si vous déclarez et initialisez une variable `p` de type `Person`, `p` est dit objet ou instance de `Person`. Plusieurs instances du même type `Person` peuvent être créées, et chaque instance peut avoir des valeurs différentes dans ses propriétés et champs.

Une classe est un type de référence. Lorsqu'un objet du type est créé, la variable à laquelle l'objet est affecté conserve uniquement une référence à cette mémoire. Lorsque la référence d'objet est affectée à une variable, la nouvelle variable fait référence à l'objet d'origine. Les modifications apportées à une variable sont répercutées sur l'autre variable, car toutes deux font référence aux mêmes données.

Un struct est un type valeur. Lorsqu'un struct est créé, la variable à laquelle le struct est assigné contient les données réelles du struct. Lorsque le struct est affecté à une nouvelle variable, il est copié. La nouvelle variable et la variable d'origine contiennent par conséquent deux copies distinctes des mêmes données. Les modifications apportées à une copie n'affectent pas l'autre copie.

Les types d'enregistrements peuvent être des types référence (`record class`) ou des types valeur (`record struct`).

En général, les classes sont utilisées pour modéliser un comportement plus complexe. Les classes stockent généralement les données destinées à être modifiées après la création d'un objet de classe. Les structs sont mieux adaptés aux petites structures de données. Les structs stockent généralement des données qui ne sont pas destinées à être modifiées après la création du struct. Les types d'enregistrements sont des structures de données avec des membres synthétisés supplémentaires du compilateur. Les enregistrements stockent généralement des données qui ne sont pas destinées à être modifiées après la création de l'objet.

Types de valeur

Les types valeur dérivent de System.ValueType, qui dérive de System.Object. Les types qui dérivent de System.ValueType ont un comportement spécial dans le CLR. Les variables de type valeur contiennent directement leurs valeurs. La mémoire d'un struct

est allouée inline dans n'importe quel contexte où la variable est déclarée. Il n'existe pas d'allocation de tas ou de surcharge de garbage collection pour les variables de type valeur. Vous pouvez déclarer des types `record struct` qui sont des types valeur et inclure les membres synthétisés pour les enregistrements.

Il existe deux catégories de types valeur : `struct` et `enum`.

Les types numériques intégrés sont des structs et ont des champs et des méthodes auxquels vous pouvez accéder :

C#

```
// constant field on type byte.
byte b = byte.MaxValue;
```

Mais vous les déclarez et leur assignez des valeurs comme s'il s'agissait de types non agrégés simples :

C#

```
byte num = 0xA;
int i = 5;
char c = 'Z';
```

Les types valeur sont *sealed*. Vous ne pouvez pas dériver un type à partir d'un type valeur, par exemple System.Int32. Vous ne pouvez pas définir un struct pour hériter d'une classe ou d'un struct défini par l'utilisateur, car un struct peut uniquement hériter de System.ValueType. Toutefois, un struct peut implémenter une ou plusieurs interfaces. Vous pouvez convertir un type de struct en n'importe quel type d'interface qu'il implémente. Ce cast entraîne une opération de *boxing* pour envelopper le struct dans un objet de type référence sur le tas managé. Les opérations de boxing se produisent quand vous passez un type valeur à une méthode qui prend un System.Object ou tout type d'interface comme paramètre d'entrée. Pour plus d'informations, consultez Conversion boxing et unboxing.

Vous utilisez le mot clé struct pour créer vos propres types valeur personnalisés. En règle générale, un struct est utilisé comme conteneur pour un petit jeu de variables connexes, comme le montre l'exemple suivant :

C#

```
public struct Coords
{
    public int x, y;
```

```
    public Coords(int p1, int p2)
    {
        x = p1;
        y = p2;
    }
}
```

Pour plus d'informations sur les structs, consultez Types de structure. Pour plus d'informations sur les types valeur, consultez Types valeur.

L'autre catégorie de types valeur est `enum`. Un enum définit un jeu de constantes intégrales nommées. Par exemple l'énumération System.IO.FileMode dans la bibliothèque de classes du .NET contient un ensemble d'entiers constants nommés qui spécifient comment un fichier doit être ouvert. Il est défini comme indiqué dans l'exemple suivant :

C#

```
public enum FileMode
{
    CreateNew = 1,
    Create = 2,
    Open = 3,
    OpenOrCreate = 4,
    Truncate = 5,
    Append = 6,
}
```

La constante System.IO.FileMode.Create a la valeur 2. Toutefois, le nom est bien plus explicite pour les utilisateurs qui lisent le code source. pour cette raison, il est préférable d'utiliser des énumérations au lieu de nombres littéraux constants. Pour plus d'informations, consultez System.IO.FileMode.

Toutes les énumérations héritent de System.Enum, qui hérite de System.ValueType. Toutes les règles qui s'appliquent aux structs s'appliquent également aux enums. Pour plus d'informations sur les enums, consultez Types énumération.

Types référence

Type défini sous la forme `class`, `record`, delegate, d'un tableau, ou interface est un reference type.

Lors de la déclaration d'une variable de reference type, elle contient la valeur null jusqu'à ce que vous l'affectiez avec une instance de ce type ou en créiez une à l'aide de

l'opérateur new. La création et l'affectation d'une classe sont illustrées dans l'exemple suivant :

C#

```
MyClass myClass = new MyClass();
MyClass myClass2 = myClass;
```

interface ne peut pas être instancié directement à l'aide de l'opérateur new. Au lieu de cela, créez et affectez une instance d'une classe qui implémente l'interface. Prenons l'exemple suivant :

C#

```
MyClass myClass = new MyClass();

// Declare and assign using an existing value.
IMyInterface myInterface = myClass;

// Or create and assign a value in a single statement.
IMyInterface myInterface2 = new MyClass();
```

Lorsque l'objet est créé, la mémoire est allouée sur le tas managé. La variable contient uniquement une référence à l'emplacement de l'objet. Les types sur le tas managé nécessitent une surcharge à la fois quand ils sont alloués et quand ils sont récupérés. *Garbage collection* est la fonctionnalité de gestion automatique de la mémoire du CLR, qui effectue la récupération. Toutefois, garbage collection est également hautement optimisé et, dans la plupart des scénarios, il ne crée pas de problème de performances. Pour plus d'informations sur le garbage collection, consultez Gestion automatique de la mémoire.

Tous les tableaux sont des types référence, même si leurs éléments sont des types valeur. Les tableaux dérivent implicitement de la classe System.Array. Vous les déclarez et les utilisez avec la syntaxe simplifiée fournie par C#, comme illustré dans l'exemple suivant :

C#

```
// Declare and initialize an array of integers.
int[] nums = { 1, 2, 3, 4, 5 };

// Access an instance property of System.Array.
int len = nums.Length;
```

Les types référence prennent en charge l'héritage. Lorsque vous créez une classe, vous pouvez hériter de toute autre interface ou classe qui n'est pas définie comme sealed. D'autres classes peuvent hériter de votre classe et remplacer vos méthodes virtuelles. Pour plus d'informations sur la création de vos propres classes, consultez Classes, structs et enregistrements. Pour plus d'informations sur l'héritage et les méthodes virtuelles, consultez Héritage.

Types de valeurs littérales

Dans C#, les valeurs littérales reçoivent un type du compilateur. Vous pouvez spécifier la façon dont un littéral numérique doit être typé en ajoutant une lettre à la fin du nombre. Par exemple, pour spécifier que la valeur `4.56` doit être traitée comme une valeur `float`, ajoutez « f » ou « F » après le nombre : `4.56f`. Si aucune lettre n'est ajoutée, le compilateur déduit un type pour le littéral. Pour plus d'informations sur les types qui peuvent être spécifiés avec des suffixes de lettre, consultez Types numériques intégraux et Types numériques à virgule flottante.

Comme les littéraux sont typés et que tous les types dérivent en fin de compte de System.Object, vous pouvez écrire et compiler du code, comme le code suivant :

C#

```
string s = "The answer is " + 5.ToString();
// Outputs: "The answer is 5"
Console.WriteLine(s);

Type type = 12345.GetType();
// Outputs: "System.Int32"
Console.WriteLine(type);
```

Types génériques

Un type peut être déclaré avec un ou plusieurs *paramètres de type* qui servent d'espace réservé pour le type réel (le *type concret*). Le code client fournit le type concret lorsqu'il crée une instance du type. Ces types sont appelés *types génériques*. Par exemple, le type .NET System.Collections.Generic.List<T> a un paramètre de type qui, par convention, porte le nom `T`. Lorsque vous créez une instance du type, vous spécifiez le type des objets contenus dans la liste, par exemple, `string` :

C#

```
List<string> stringList = new List<string>();
stringList.Add("String example");
// compile time error adding a type other than a string:
stringList.Add(4);
```

L'utilisation du paramètre de type rend possible la réutilisation de la même classe pour contenir tout type d'élément, sans avoir à convertir chaque élément en object. Les classes de collections génériques sont appelées *collections fortement typées*, car le compilateur connaît le type spécifique des éléments de chaque collection et il peut déclencher une erreur au moment de la compilation. C'est le cas, par exemple, si vous essayez d'ajouter un entier à l'objet `stringList` dans l'exemple précédent. Pour plus d'informations, consultez Génériques.

Types implicites, types anonymes et types valeur pouvant accepter la valeur Null

Vous pouvez attribuer implicitement un type à une variable locale (mais pas les membres de la classe) à l'aide du mot clé var. La variable reçoit toujours un type au moment de la compilation, mais le type est fourni par le compilateur. Pour plus d'informations, consultez Variables locales implicitement typées.

Il peut être difficile de créer un type nommé pour des ensembles simples de valeurs associées que vous ne souhaitez pas stocker ou transférer en dehors des limites de la méthode. Vous pouvez alors créer des *types anonymes*. Pour plus d'informations, consultez Types anonymes.

Les types valeur ordinaires ne peuvent pas avoir la valeur null. Toutefois, vous pouvez créer des *types valeur pouvant accepter la valeur Null* en apposant un `?` après le type. Par exemple, `int?` est un type `int` qui peut également avoir la valeur null. Les types valeur pouvant accepter la valeur Null sont des instances du type struct générique System.Nullable<T>. Les types valeur pouvant accepter la valeur Null sont particulièrement utiles lorsque vous transmettez des données vers et à partir de bases de données dans lesquelles les valeurs numériques peuvent être `null`. Pour plus d'informations, consultez Types valeur pouvant accepter la valeur Null.

Type de compilation et type d'exécution

Une variable peut avoir différents types de compilation et d'exécution. Le *type au moment de la compilation* est le type déclaré ou différé de la variable dans le code

source. Le *type d'exécution* est le type de l'instance référencée par cette variable. Souvent, ces deux types sont les mêmes, comme dans l'exemple suivant :

```
C#
```

```
string message = "This is a string of characters";
```

Dans d'autres cas, le type de compilation est différent, comme illustré dans les deux exemples suivants :

```
C#
```

```
object anotherMessage = "This is another string of characters";
IEnumerable<char> someCharacters = "abcdefghijklmnopqrstuvwxyz";
```

Dans les deux exemples précédents, le type d'exécution est un `string`. Le type au moment de la compilation est `object` dans la première ligne et `IEnumerable<char>` dans la deuxième.

Si les deux types sont différents pour une variable, il est important de comprendre quand le type de compilation et le type d'exécution s'appliquent. Le type de compilation détermine toutes les actions effectuées par le compilateur. Ces actions du compilateur incluent la résolution des appels de méthode, la résolution de surcharge et les casts implicites et explicites disponibles. Le type d'exécution détermine toutes les actions qui sont résolues au moment de l'exécution. Ces actions d'exécution incluent la distribution d'appels de méthode virtuelle, l'évaluation des expressions `is` et `switch`, ainsi que d'autres API de test de type. Pour mieux comprendre comment votre code interagit avec les types, identifiez l'action qui s'applique à quel type.

Sections connexes

Pour plus d'informations, consultez les articles suivants :

- Types intégrés
- Types valeur
- Types référence

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Déclarez des espaces de noms pour organiser les types

Article • 13/07/2023

Les espaces de noms sont largement utilisés de deux façons dans la programmation avec C#. Tout d'abord, .NET utilise des espaces de noms pour organiser ses nombreuses classes, comme suit :

```
C#

System.Console.WriteLine("Hello World!");
```

System est un espace de noms et Console est une classe de cet espace de noms. Le mot clé `using` peut être utilisé afin que le nom complet ne soit pas obligatoire, comme dans l'exemple suivant :

```
C#

using System;
```

```
C#

Console.WriteLine("Hello World!");
```

Pour plus d'informations, consultez Directive using.

> **ⓘ Important**
>
> Les modèles C# pour .NET 6 utilisent des *instructions de niveau supérieur*. Votre application ne correspond peut-être pas au code de cet article, si vous avez déjà effectué une mise à niveau vers .NET 6. Pour plus d'informations, consultez l'article **Les nouveaux modèles C# génèrent des instructions de niveau supérieur**.
>
> Le SDK .NET 6 ajoute également un ensemble de directives *implicites* `global using` pour les projets qui utilisent les kits SDK suivants :
>
> - Microsoft.NET.Sdk
> - Microsoft.NET.Sdk.Web
> - Microsoft.NET.Sdk.Worker

Ces directives implicites `global using` incluent les espaces de noms les plus courants pour le type de projet.

Pour plus d'informations, consultez l'article sur les **directives d'utilisation implicite**

Deuxièmement, la déclaration de vos propres espaces de noms peut vous aider à contrôler l'étendue des noms de classes et de méthodes dans les projets de programmation plus vastes. Utilisez le mot clé namespace pour déclarer un espace de noms, comme dans l'exemple suivant :

C#

```
namespace SampleNamespace
{
    class SampleClass
    {
        public void SampleMethod()
        {
            System.Console.WriteLine(
                "SampleMethod inside SampleNamespace");
        }
    }
}
```

Le nom de l'espace de noms doit être un nom d'identificateur C# valide.

À compter de C# 10, vous pouvez déclarer un espace de noms pour tous les types définis dans ce fichier, comme illustré dans l'exemple suivant :

C#

```
namespace SampleNamespace;

class AnotherSampleClass
{
    public void AnotherSampleMethod()
    {
        System.Console.WriteLine(
            "SampleMethod inside SampleNamespace");
    }
}
```

L'avantage de cette nouvelle syntaxe est qu'elle est plus simple, ce qui permet d'économiser de l'espace horizontal et des accolades. Cela facilite la lecture de votre code.

Vue d'ensemble des espaces de noms

Les espaces de noms ont les propriétés suivantes :

- Ils organisent les projets de code de taille importante.
- Ils sont délimités à l'aide de l'opérateur `.`.
- La directive `using` évite de devoir spécifier le nom de l'espace de noms pour chaque classe.
- L'espace de noms `global` est l'espace de noms « racine » : `global::System` fait toujours référence à l'espace de noms System .NET.

spécification du langage C#

Pour plus d'informations, voir la section Espace de noms de la spécification du langage C#.

Introduction aux classes

Article • 01/06/2023

Types référence

Un type défini comme classest un *type référence*. Au moment de l'exécution, quand vous déclarez une variable de type référence, celle-ci contient la valeur null tant que vous n'avez pas explicitement créé une instance de la classe à l'aide de l'opérateur new, ou que vous ne lui avez pas assigné un objet existant d'un type compatible, comme indiqué dans l'exemple suivant :

C#

```
//Declaring an object of type MyClass.
MyClass mc = new MyClass();

//Declaring another object of the same type, assigning it the value of the
first object.
MyClass mc2 = mc;
```

Quand l'objet est créé, une quantité de mémoire suffisante est allouée sur le tas managé de l'objet spécifié, et la variable contient uniquement une référence à l'emplacement de cet objet. La mémoire utilisée par un objet est récupérée par la fonctionnalité de gestion automatique de la mémoire du CLR, appelée *garbage collection*. Pour plus d'informations sur le garbage collection, consultez Gestion automatique de la mémoire et garbage collection.

Déclaration de classes

Les classes sont déclarées à l'aide du mot clé `class` suivi d'un identificateur unique, comme l'illustre l'exemple suivant :

C#

```
//[access modifier] - [class] - [identifier]
public class Customer
{
   // Fields, properties, methods and events go here...
}
```

Un modificateur d'accès facultatif précède le mot clé `class`. Comme public est utilisé dans ce cas, n'importe qui peut créer des instances de cette classe. Le nom de la classe suit le mot clé `class`. Le nom de la classe doit être un nom d'identificateur C# valide. Le reste de la définition est le corps de la classe, où le comportement et les données sont définis. Les champs, propriétés, méthodes et événements d'une classe sont désignés collectivement par le terme « *membres de classe* ».

Création d'objets

Bien qu'ils soient parfois employés indifféremment, une classe et un objet sont deux choses différentes. Une classe définit un type d'objet, mais il ne s'agit pas d'un objet en soi. Un objet, qui est une entité concrète basée sur une classe, est parfois désigné par le terme « instance de classe ».

Les objets peuvent être créés à l'aide du mot clé `new` suivi du nom de la classe, comme suit :

```
C#

Customer object1 = new Customer();
```

Quand une instance d'une classe est créée, une référence à l'objet est repassée au programmeur. Dans l'exemple précédent, `object1` est une référence à un objet basé sur `Customer`. Cette référence fait référence au nouvel objet, mais elle ne contient pas ses données. En fait, vous pouvez créer une référence d'objet sans créer d'objet :

```
C#

Customer object2;
```

Nous vous déconseillons de créer des références d'objet sans référence à un objet, car toute tentative d'accès à un objet à l'aide d'une telle référence échoue au moment de l'exécution. Vous pouvez créer une telle référence pour faire référence à un objet : soit en créant un objet, soit en l'assignant à un objet existant, comme suit :

```
C#

Customer object3 = new Customer();
Customer object4 = object3;
```

Ce code crée deux références d'objet qui font toutes deux référence au même objet. Toute modification apportée à l'objet par le biais de `object3` est donc reflétée dans les utilisations suivantes de `object4`. Les objets qui sont basés sur des classes étant désignés par référence, les classes sont appelées des « types référence ».

Constructeurs et initialisation

Les sections précédentes ont introduit la syntaxe pour déclarer un type de classe et créer une instance de ce type. Lorsque vous créez une instance d'un type, vous souhaitez vous assurer que ses champs et ses propriétés sont initialisés en valeurs utiles. Il existe plusieurs façons d'initialiser des valeurs :

- Accepter les valeurs par défaut
- Initialiseurs de champs
- Paramètres du constructeur
- Initialiseurs d'objet

Chaque type .NET a une valeur par défaut. En règle générale, cette valeur est 0 pour les types numériques et `null` pour tous les types de référence. Vous pouvez vous appuyer sur cette valeur par défaut lorsqu'elle est raisonnable dans votre application.

Lorsque la valeur par défaut de .NET n'est pas la bonne, vous pouvez définir une valeur initiale à l'aide d'un *initialiseur de champ* :

C#

```
public class Container
{
    // Initialize capacity field to a default value of 10:
    private int _capacity = 10;
}
```

Vous pouvez exiger que les appelants fournissent une valeur initiale en définissant un *constructeur* responsable de la définition de cette valeur initiale :

C#

```
public class Container
{
    private int _capacity;

    public Container(int capacity) => _capacity = capacity;
}
```

À compter de C# 12, vous pouvez définir un *constructeur principal* dans le cadre de la déclaration de classe :

```
C#

public class Container(int capacity)
{
    private int _capacity = capacity;
}
```

L'ajout de paramètres au nom de classe définit le *constructeur principal*. Ces paramètres sont disponibles dans le corps de la classe, qui inclut ses membres. Vous pouvez les utiliser pour initialiser des champs ou n'importe où où ils sont nécessaires.

Vous pouvez également utiliser le modificateur `required` sur une propriété et autoriser les appelants à utiliser un *initialiseur d'objet* pour définir la valeur initiale de la propriété :

```
C#

public class Person
{
    public required string LastName { get; set; }
    public required string FirstName { get; set; }
}
```

L'ajout des mandats de mot clé `required` que les appelants doivent définir ces propriétés dans le cadre d'une expression `new` :

```
C#

var p1 = new Person(); // Error! Required properties not set
var p2 = new Person() { FirstName = "Grace", LastName = "Hopper" };
```

Héritage de classe

Les classes prennent entièrement en charge l'*héritage*, caractéristique fondamentale de la programmation orientée objet. Lorsque vous créez une classe, vous pouvez hériter de toute autre classe qui n'est pas définie comme sealed. D'autres classes peuvent hériter de votre classe et remplacer vos méthodes virtuelles de classe. En outre, vous pouvez implémenter une ou plusieurs interfaces.

L'héritage se fait par le biais d'une *dérivation*, ce qui signifie qu'une classe est déclarée à l'aide d'une *classe de base* dont elle hérite les données et le comportement. Pour

spécifier une classe de base, ajoutez deux-points et le nom de la classe de base après le nom de la classe dérivée, comme suit :

```
C#
```

```
public class Manager : Employee
{
    // Employee fields, properties, methods and events are inherited
    // New Manager fields, properties, methods and events go here...
}
```

Quand une déclaration de classe inclut une classe de base, elle hérite de tous les membres de la classe de base à l'exception des constructeurs. Pour plus d'informations, consultez Héritage.

Une classe en C# ne peut hériter directement que d'une seule classe de base. Toutefois, une classe de base pouvant elle-même hériter d'une autre classe, une classe peut hériter indirectement de plusieurs classes de base. En outre, une classe peut implémenter directement une ou plusieurs interfaces. Pour plus d'informations, consultez Interfaces.

Une classe peut être déclarée abstract. Une classe abstraite contient des méthodes abstraites qui ont une définition de signature, mais aucune implémentation. Les classes abstraites ne peuvent pas être instanciées. Elles peuvent être utilisées uniquement à travers des classes dérivées qui implémentent les méthodes abstraites. En revanche, une classe sealed ne permet pas à d'autres classes de dériver d'elle. Pour plus d'informations, consultez Classes abstract et sealed, et membres de classe.

Les définitions de classe peuvent être fractionnées entre différents fichiers sources. Pour plus d'informations, consultez la page Classes et méthodes partielles.

Spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Introduction aux types d'enregistrements en C#

Article • 01/06/2023

Un enregistrement en C# est une classe ou un struct qui fournit une syntaxe et un comportement spéciaux pour l'utilisation de modèles de données. Le modificateur `record` indique au compilateur de synthétiser des membres utiles pour les types dont le rôle principal stocke des données. Ces membres incluent une surcharge et des membres qui prennent en charge l'égalité des valeurs ToString().

Quand utiliser des enregistrements

Envisagez d'utiliser un enregistrement à la place d'une classe ou d'un struct dans les scénarios suivants :

- Vous souhaitez définir un modèle de données qui dépend de l'égalité des valeurs.
- Vous souhaitez définir un type pour lequel les objets sont immuables.

Égalité des valeurs

Pour les enregistrements, l'égalité des valeurs signifie que deux variables d'un type d'enregistrement sont égales si les types correspondent et que toutes les valeurs de propriété et de champ correspondent. Pour d'autres types de référence tels que les classes, l'égalité signifie l'égalité des références. Autrement dit, deux variables d'un type de classe sont égales si elles font référence au même objet. Les méthodes et les opérateurs qui déterminent l'égalité de deux instances d'enregistrement utilisent l'égalité des valeurs.

Tous les modèles de données ne fonctionnent pas correctement avec l'égalité des valeurs. Par exemple, Entity Framework Core dépend de l'égalité des références pour s'assurer qu'il n'utilise qu'une seule instance d'un type d'entité pour ce qui est conceptuellement une seule entité. Pour cette raison, les types d'enregistrements ne sont pas appropriés pour une utilisation en tant que types d'entités dans Entity Framework Core.

Immuabilité

Un type immuable est un type qui vous empêche de modifier les valeurs de propriété ou de champ d'un objet après son instanciation. L'immuabilité peut être utile lorsque vous

avez besoin d'un type thread-safe ou que vous dépendez d'un code de hachage restant le même dans une table de hachage. Les enregistrements fournissent une syntaxe concise pour la création et l'utilisation de types immuables.

L'immuabilité n'est pas appropriée pour tous les scénarios de données. Entity Framework Core, par exemple, ne prend pas en charge la mise à jour avec des types d'entités immuables.

Différences entre les enregistrements et les classes et les structs

La même syntaxe qui déclare et instancie des classes ou des structs peut être utilisée avec des enregistrements. Remplacez simplement le mot clé `class` par `record`, ou utilisez `record struct` au lieu de `struct`. De même, la syntaxe utilisée pour exprimer les relations d'héritage est prise en charge par les classes d'enregistrement. Les enregistrements diffèrent des classes de la manière suivante :

- Vous pouvez utiliser des paramètres positionnels dans un constructeur principal pour créer et instancier un type avec des propriétés immuables.
- Les mêmes méthodes et opérateurs qui indiquent l'égalité de référence ou l'inégalité dans les classes (par exemple Object.Equals(Object) et `==`), indiquent l'égalité ou l'inégalité de valeur dans les enregistrements.
- Vous pouvez utiliser une expressionwith pour créer une copie d'un objet immuable avec de nouvelles valeurs dans les propriétés sélectionnées.
- La méthode `ToString` d'un enregistrement crée une chaîne mise en forme qui affiche le nom de type d'un objet et les noms et valeurs de toutes ses propriétés publiques.
- Un enregistrement peut hériter d'un autre enregistrement. Un enregistrement ne peut pas hériter d'une classe et une classe ne peut pas hériter d'un enregistrement.

Les structs d'enregistrement diffèrent des structs : le compilateur synthétise les méthodes d'égalité et `ToString`. Le compilateur synthétise une méthode `Deconstruct` pour les structs d'enregistrement positionnel.

Le compilateur synthétise une propriété init seule publique pour chaque paramètre de constructeur principal dans un `record class`. Dans un `record struct`, le compilateur synthétise une propriété en lecture-écriture publique. Le compilateur ne crée pas de propriétés pour les paramètres de constructeur principal dans les types `class` et `struct` qui n'incluent pas de modificateur `record`.

Exemples

L'exemple suivant définit un enregistrement public qui utilise des paramètres positionnels pour déclarer et instancier un enregistrement. Il affiche ensuite le nom de type et les valeurs de propriété :

C#

```
public record Person(string FirstName, string LastName);

public static void Main()
{
    Person person = new("Nancy", "Davolio");
    Console.WriteLine(person);
    // output: Person { FirstName = Nancy, LastName = Davolio }
}
```

L'exemple suivant illustre l'égalité des valeurs dans les enregistrements :

C#

```
public record Person(string FirstName, string LastName, string[]
PhoneNumbers);

public static void Main()
{
    var phoneNumbers = new string[2];
    Person person1 = new("Nancy", "Davolio", phoneNumbers);
    Person person2 = new("Nancy", "Davolio", phoneNumbers);
    Console.WriteLine(person1 == person2); // output: True

    person1.PhoneNumbers[0] = "555-1234";
    Console.WriteLine(person1 == person2); // output: True

    Console.WriteLine(ReferenceEquals(person1, person2)); // output: False
}
```

L'exemple suivant illustre l'utilisation d'une expression `with` pour copier un objet immuable et modifier l'une des propriétés :

C#

```
public record Person(string FirstName, string LastName)
{
    public string[] PhoneNumbers { get; init; }
}

public static void Main()
{
```

```
    Person person1 = new("Nancy", "Davolio") { PhoneNumbers = new string[1]
};
    Console.WriteLine(person1);
    // output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers
= System.String[] }

    Person person2 = person1 with { FirstName = "John" };
    Console.WriteLine(person2);
    // output: Person { FirstName = John, LastName = Davolio, PhoneNumbers =
System.String[] }
    Console.WriteLine(person1 == person2); // output: False

    person2 = person1 with { PhoneNumbers = new string[1] };
    Console.WriteLine(person2);
    // output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers
= System.String[] }
    Console.WriteLine(person1 == person2); // output: False

    person2 = person1 with { };
    Console.WriteLine(person1 == person2); // output: True
}
```

Pour plus d'informations, consultez Enregistrements (référence C#).

Spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Interfaces : définir le comportement pour plusieurs types

Article • 18/03/2023

Une interface contient des définitions pour un groupe de fonctionnalités connexes qu'une class non abstraite ou un struct doivent implémenter. Une interface peut définir des méthodes `static`, qui doivent avoir une implémentation. Une interface peut définir une implémentation par défaut pour les membres. Une interface peut ne pas déclarer de données d'instance telles que des champs, des propriétés implémentées automatiquement ou des événements similaires à une propriété.

En utilisant des interfaces, vous pouvez, par exemple, inclure le comportement de plusieurs sources dans une classe. Cette fonctionnalité est importante en C#, car le langage ne prend pas en charge l'héritage multiple de classes. De plus, vous devez utiliser une interface si vous voulez simuler l'héritage pour des structs, car ils ne peuvent en fait pas hériter d'un autre struct ou d'une autre classe.

Vous définissez une interface à l'aide du mot clé interface, comme le montre l'exemple suivant.

C#

```
interface IEquatable<T>
{
    bool Equals(T obj);
}
```

Le nom de l'interface doit être un nom d'identificateur C# valide. Par convention, les noms d'interface commencent par un `I` majuscule.

Toute classe ou tout struct qui implémentent l'interface IEquatable<T> doivent contenir une définition pour une méthode Equals qui correspond à la signature spécifiée par l'interface. Ainsi, vous pouvez compter sur une classe qui implémente `IEquatable<T>` pour contenir une méthode `Equals` avec laquelle une instance de la classe peut déterminer si elle est égale à une autre instance de la même classe.

La définition de `IEquatable<T>` ne fournit pas d'implémentation pour `Equals`. Une classe ou un struct peut implémenter plusieurs interfaces, mais une classe peut uniquement hériter d'une classe unique.

Pour plus d'informations sur les classes abstraites, consultez Classes abstract et sealed et membres de classe.

Les interfaces peuvent contenir des méthodes d'instance, propriétés, événements, indexeurs ou toute combinaison de ces quatre types de membres. Les interfaces peuvent contenir des constructeurs statiques, des champs, des constantes ou des opérateurs. À compter de C# 11, les membres d'interface qui ne sont pas des champs peuvent être `static abstract`. Une interface ne peut pas contenir de champs d'instance, de constructeurs d'instance ou de finaliseurs. Les membres de l'interface sont publics par défaut et vous pouvez spécifier explicitement des modificateurs d'accessibilité, tels que `public`, `protected`, `internal`, `private`, `protected internal` ou `private protected`. Un membre `private` doit avoir une implémentation par défaut.

Pour implémenter un membre d'interface, le membre correspondant de la classe d'implémentation doit être public, non statique et porter le même nom et la même signature que le membre d'interface.

> **Notes**
>
> Lorsqu'une interface déclare des membres statiques, un type implémentant cette interface peut également déclarer des membres statiques avec la même signature. Celles-ci sont distinctes et identifiées de manière unique par le type déclarant le membre. Le membre statique déclaré dans un type ne remplace *pas* le membre statique déclaré dans l'interface.

Une classe ou un struct qui implémente une interface doit fournir une implémentation pour tous les membres déclarés sans implémentation par défaut fournie par l'interface. Toutefois, si une classe de base implémente une interface, toute classe dérivée de la classe de base hérite de cette implémentation.

L'exemple suivant illustre une implémentation de l'interface IEquatable<T>. La classe d'implémentation, `Car`, doit fournir une implémentation de la méthode Equals.

C#

```
public class Car : IEquatable<Car>
{
    public string? Make { get; set; }
    public string? Model { get; set; }
    public string? Year { get; set; }

    // Implementation of IEquatable<T> interface
    public bool Equals(Car? car)
    {
```

```
        return (this.Make, this.Model, this.Year) ==
            (car?.Make, car?.Model, car?.Year);
    }
}
```

Les propriétés et indexeurs d'une classe peuvent définir des accesseurs supplémentaires pour une propriété ou un indexeur qui est défini dans une interface. Par exemple, une interface peut déclarer une propriété qui a un accesseur get. La classe qui implémente l'interface peut déclarer la même propriété avec à la fois un accesseur `get` et un accesseur set. Toutefois, si la propriété ou l'indexeur utilisent une implémentation explicite, les accesseurs doivent correspondre. Pour plus d'informations sur l'implémentation explicite, consultez les pages Implémentation d'interface explicite et Propriétés d'interface.

Des interfaces peuvent hériter d'une ou de plusieurs interfaces. L'interface dérivée hérite des membres de ses interfaces de base. Une classe qui implémente une interface dérivée doit implémenter tous les membres de l'interface dérivée, y compris tous les membres des interfaces de base de l'interface dérivée. Cette classe peut être convertie implicitement en l'interface dérivée ou l'une de ses interfaces de base. Une classe peut inclure une interface plusieurs fois par le biais de classes de base qu'elle hérite ou d'interfaces dont d'autres interfaces héritent. Toutefois, la classe peut fournir une implémentation d'une interface une seule fois et uniquement si la classe déclare l'interface dans le cadre de la définition de la classe (`class ClassName : InterfaceName`). Si l'interface est héritée car vous avez hérité d'une classe de base qui implémente l'interface, la classe de base fournit l'implémentation des membres de l'interface. Toutefois, la classe dérivée peut réimplémenter des membres d'interface virtuels au lieu d'utiliser l'implémentation héritée. Lorsque des interfaces déclarent une implémentation par défaut d'une méthode, toute classe implémentant cette interface hérite de cette implémentation (vous devez caster l'instance de classe vers le type d'interface pour accéder à l'implémentation par défaut sur le membre Interface).

Une classe de base peut également implémenter des membres d'interface à l'aide de membres virtuels. Dans ce cas, une classe dérivée peut modifier le comportement de l'interface en substituant les membres virtuels. Pour plus d'informations sur les membres virtuels, consultez la page Polymorphisme.

Résumé des interfaces

Une interface possède les propriétés suivantes :

- Dans les versions C# antérieures à 8.0, une interface est comme une classe de base abstraite avec uniquement des membres abstraits. Une classe ou un struct qui

implémente l'interface doivent implémenter tous ses membres.

- À compter de C# 8.0, une interface peut définir des implémentations par défaut pour tout ou partie de ses membres. Une classe ou un struct qui implémente l'interface n'a pas besoin d'implémenter les membres qui ont des implémentations par défaut. Pour plus d'informations, consultez Méthodes d'interface par défaut.
- Une interface ne peut pas être instanciée directement. Ses membres sont implémentées par une classe ou un struct qui implémentent l'interface.
- Une classe ou un struct peuvent implémenter plusieurs interfaces. Une classe peut hériter d'une classe de base et également implémenter une ou plusieurs interfaces.

Classes et méthodes génériques

Article • 13/07/2023

Les génériques introduisent le concept des paramètres de type dans .NET, qui permettent de concevoir des classes et des méthodes qui diffèrent la spécification d'un ou de plusieurs types jusqu'à ce que la classe ou la méthode soit déclarée et instanciée par le code client. Par exemple, à l'aide d'un paramètre de type générique `T`, vous pouvez écrire une seule classe qui peut être utilisée par un autre code client sans impliquer le coût ou le risque des casts de runtime ou des opérations de boxing à l'exécution, comme illustré ici :

C#

```
// Declare the generic class.
public class GenericList<T>
{
    public void Add(T input) { }
}
class TestGenericList
{
    private class ExampleClass { }
    static void Main()
    {
        // Declare a list of type int.
        GenericList<int> list1 = new GenericList<int>();
        list1.Add(1);

        // Declare a list of type string.
        GenericList<string> list2 = new GenericList<string>();
        list2.Add("");

        // Declare a list of type ExampleClass.
        GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();
        list3.Add(new ExampleClass());
    }
}
```

Les méthodes et les classes génériques combinent la réutilisabilité, la cohérence des types et l'efficacité, ce que ne peuvent pas faire leurs équivalents non génériques. Les génériques sont plus fréquemment utilisés dans des collections et des méthodes qui agissent sur eux. L'espace de noms System.Collections.Generic contient plusieurs classes de collection génériques. Les collections non génériques, telles que ArrayList, ne sont pas recommandées et sont conservées à des fins de compatibilité. Pour plus d'informations, consultez Génériques en .NET.

Bien sûr, vous pouvez également créer des types et des méthodes génériques personnalisés pour fournir vos propres solutions et modèles de conception généralisés qui soient efficaces et de type sécurisé. L'exemple de code suivant montre une classe de liste liée générique simple, à des fins de démonstration. (Dans la plupart des cas, vous devez utiliser la classe List<T> fournie par .NET au lieu de créer la vôtre.) Le paramètre de type `T` est utilisé dans plusieurs emplacements où un type concret est généralement utilisé pour indiquer le type de l'élément stocké dans la liste. Il est utilisé de la façon suivante :

- Comme le type d'un paramètre de méthode dans la méthode `AddHead`.
- Comme le type de retour de la propriété `Data` de la classe imbriquée `Node`.
- Comme le type de membre privé `data` de la classe imbriquée.

`T` est disponible pour la classe imbriquée `Node`. Quand `GenericList<T>` est instancié avec un type concret, par exemple comme un `GenericList<int>`, chaque occurrence de `T` est remplacée par `int`.

C#

```
// type parameter T in angle brackets
public class GenericList<T>
{
    // The nested class is also generic on T.
    private class Node
    {
        // T used in non-generic constructor.
        public Node(T t)
        {
            next = null;
            data = t;
        }

        private Node? next;
        public Node? Next
        {
            get { return next; }
            set { next = value; }
        }

        // T as private member data type.
        private T data;

        // T as return type of property.
        public T Data
        {
            get { return data; }
            set { data = value; }
        }
    }
```

```
    private Node? head;

    // constructor
    public GenericList()
    {
        head = null;
    }

    // T as method parameter type:
    public void AddHead(T t)
    {
        Node n = new Node(t);
        n.Next = head;
        head = n;
    }

    public IEnumerator<T> GetEnumerator()
    {
        Node? current = head;

        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }
}
```

L'exemple de code suivant montre comment le code client utilise la classe générique `GenericList<T>` pour créer une liste d'entiers. En changeant l'argument de type, vous pouvez facilement modifier le code suivant pour créer des listes de chaînes ou tout autre type personnalisé :

C#

```
class TestGenericList
{
    static void Main()
    {
        // int is the type argument
        GenericList<int> list = new GenericList<int>();

        for (int x = 0; x < 10; x++)
        {
            list.AddHead(x);
        }

        foreach (int i in list)
        {
            System.Console.Write(i + " ");
        }
```

```
        System.Console.WriteLine("\nDone");
    }
}
```

> **Notes**
>
> Les exemples suivants s'appliquent non seulement aux `class` types, mais aussi `interface` aux types et `struct`

Vue d'ensemble des génériques

- Utilisez des types génériques pour optimiser la réutilisation de code, la sécurité des types et les performances.
- L'utilisation la plus courante des génériques est de créer des classes de collection.
- La bibliothèque de classes de .NET contient plusieurs nouvelles classes de collection génériques dans l'espace de noms System.Collections.Generic. Les collections génériques doivent être utilisées chaque fois que c'est possible, à la place de classes comme ArrayList dans l'espace de noms System.Collections.
- Vous pouvez créer vos propres interfaces, classes, méthodes, événements et délégués génériques.
- Les classes génériques peuvent être contraintes pour permettre l'accès à des méthodes sur des types de données particuliers.
- Des informations sur les types qui sont utilisés dans un type de données générique peuvent être obtenues à l'exécution à l'aide de la réflexion.

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#.

Voir aussi

- System.Collections.Generic
- Génériques en .NET

Types anonymes

Article • 10/05/2023

Les types anonymes permettent d'encapsuler un ensemble de propriétés en lecture seule dans un unique objet sans avoir à définir explicitement un type. Le nom du type est généré par le compilateur et n'est pas disponible au niveau du code source. Le type de chaque propriété est déduit par le compilateur.

Vous créez des types anonymes en utilisant l'opérateur new avec un initialiseur d'objet. Pour plus d'informations sur les initialiseurs d'objets, consultez Initialiseurs d'objets et de collections.

L'exemple suivant montre un type anonyme qui est initialisé avec deux propriétés nommées `Amount` et `Message`.

C#

```
var v = new { Amount = 108, Message = "Hello" };

// Rest the mouse pointer over v.Amount and v.Message in the following
// statement to verify that their inferred types are int and string.
Console.WriteLine(v.Amount + v.Message);
```

Les types anonymes sont en règle générale utilisés dans la clause select d'une expression de requête pour retourner un sous-ensemble de propriétés de chaque objet dans la séquence source. Pour plus d'informations sur les requêtes, voir LINQ en C#.

Les types anonymes contiennent une ou plusieurs propriétés en lecture seule publiques. Aucun autre type de membres de classe, tels que des méthodes ou des événements, n'est valide. L'expression qui est utilisée pour initialiser une propriété ne peut pas être `null`, une fonction anonyme ou un type pointeur.

Le scénario le plus courant consiste à initialiser un type anonyme avec les propriétés d'un autre type. Dans l'exemple suivant, une classe existe qui porte le nom `Product`. La classe `Product` comprend des propriétés `Color` et `Price`, ainsi que d'autres propriétés qui ne vous intéressent pas. La variable `products` est une collection d'objets `Product`. La déclaration de type anonyme commence par le mot clé `new`. La déclaration initialise un nouveau type qui utilise uniquement deux propriétés de `Product`. L'utilisation de types anonymes entraîne le retour d'une quantité réduite de données dans la requête.

Si vous n'indiquez pas les noms de membre dans le type anonyme, le compilateur attribue aux membres de type anonyme le même nom que celui de la propriété utilisée

pour les initialiser. Vous fournissez un nom pour une propriété qui est initialisée avec une expression, comme indiqué dans l'exemple précédent. Dans l'exemple suivant, les noms des propriétés du type anonyme sont `Color` et `Price`.

C#

```
var productQuery =
    from prod in products
    select new { prod.Color, prod.Price };

foreach (var v in productQuery)
{
    Console.WriteLine("Color={0}, Price={1}", v.Color, v.Price);
}
```

Conseil

Vous pouvez utiliser la règle de style .NET **IDE0037** pour indiquer si les noms de membres déduits ou explicites sont préférés.

Il est également possible de définir un champ par objet d'un autre type : classe, struct ou même un autre type anonyme. Pour ce faire, utilisez la variable contenant cet objet comme dans l'exemple suivant, où deux types anonymes sont créés à l'aide de types définis par l'utilisateur déjà instanciés. Dans les deux cas, le champ `product` du type anonyme `shipment` et `shipmentWithBonus` sera de type `Product` contenant les valeurs par défaut de chaque champ. Et le champ `bonus` sera de type anonyme créé par le compilateur.

C#

```
var product = new Product();
var bonus = new { note = "You won!" };
var shipment = new { address = "Nowhere St.", product };
var shipmentWithBonus = new { address = "Somewhere St.", product, bonus };
```

Quand vous utilisez un type anonyme pour initialiser une variable, vous déclarez la variable en tant que variable locale implicitement typée en utilisant var. Le nom de type ne peut pas être spécifié dans la déclaration de variable, car seul le compilateur a accès au nom sous-jacent du type anonyme. Pour plus d'informations sur `var`, consultez Variables locales implicitement typées.

Vous pouvez créer un tableau d'éléments typés anonymement en associant une variable locale implicitement typée et un tableau typé implicitement, comme indiqué dans

l'exemple suivant.

```
C#
```

```
var anonArray = new[] { new { name = "apple", diam = 4 }, new { name =
"grape", diam = 1 }};
```

Les types anonymes sont des types class qui dérivent directement de object et qui ne peuvent pas être castés en type à l'exception du type object. Le compilateur fournit un nom pour chaque type anonyme, bien que votre application ne puisse pas y accéder. Du point de vue du CLR, un type anonyme n'est pas différent des autres types de référence.

Si plusieurs initialiseurs d'objet dans un assembly spécifient une séquence de propriétés dans le même ordre et qui sont du même type et portent le même nom, le compilateur traite les objets comme des instances du même type. Elles partagent les mêmes informations de type générées par le compilateur.

Les types anonymes prennent en charge la mutation non destructrice sous la forme d'expressions. Cela vous permet de créer une instance d'un type anonyme où une ou plusieurs propriétés ont de nouvelles valeurs :

```
C#
```

```
var apple = new { Item = "apples", Price = 1.35 };
var onSale = apple with { Price = 0.79 };
Console.WriteLine(apple);
Console.WriteLine(onSale);
```

Vous ne pouvez pas déclarer un champ, une propriété, un événement ou le type de retour d'une méthode comme ayant un type anonyme. De même, vous ne pouvez pas déclarer un paramètre formel d'une méthode, d'une propriété, d'un constructeur ou d'un indexeur comme ayant un type anonyme. Pour passer un type anonyme, ou une collection qui contient des types anonymes, en tant qu'argument d'une méthode, vous pouvez déclarer le paramètre comme `object` de type. Toutefois, l'utilisation de `object` pour les types anonymes va à l'encontre de l'objectif d'un typage fort. Si vous devez stocker les résultats de requête ou les passer en dehors des limites de la méthode, utilisez un struct ordinaire ou une classe au lieu d'un type anonyme.

Dans la mesure où les méthodes Equals et GetHashCode dans les types anonymes sont définies selon les termes des méthodes `Equals` et `GetHashCode` des propriétés, deux instances du même type anonyme sont égales si toutes leurs propriétés sont égales.

Les types anonymes remplacent la méthode ToString, en concaténant le nom et la sortie `ToString` de chaque propriété entourée d'accolades.

```
var v = new { Title = "Hello", Age = 24 };

Console.WriteLine(v.ToString()); // "{ Title = Hello, Age = 24 }"
```

Vue d'ensemble des classes, des structs et des enregistrements en C#

Article • 23/02/2023

En C#, la définition d'un type (classe, struct ou enregistrement) ressemble à un blueprint qui spécifie ce que le type peut faire. Un objet est fondamentalement un bloc de mémoire qui a été alloué et configuré selon le plan. Cet article fournit une vue d'ensemble de ces blueprints et de leurs fonctionnalités. L'article suivant de cette série présente les objets.

Encapsulation

L'encapsulation est parfois considérée comme le premier pilier ou principe de la programmation orientée objet. Une classe ou un struct peut spécifier le degré d'accessibilité de chacun de ses membres au code situé en dehors de la classe ou du struct. Les méthodes et variables qui ne sont pas destinées à être utilisées depuis l'extérieur de la classe ou de l'assembly peuvent être masquées afin de limiter le risque d'erreurs ou de code malveillant exploitant une faille de sécurité. Pour plus d'informations, consultez le didacticiel Programmation orientée objet.

Membres

Les *membres* d'un type incluent toutes les méthodes, champs, constantes, propriétés et événements. En C#, il n'existe aucune variable ou méthode globale, à la différence d'autres langages. Même le point d'entrée d'un programme, la méthode `Main`, doit être déclaré dans une classe ou un struct (implicitement lorsque vous utilisez des instructions de niveau supérieur).

La liste suivante inclut tous les types de membres qui peuvent être déclarés dans une classe, un struct ou un enregistrement.

- Champs
- Constantes
- Propriétés
- Méthodes
- Constructeurs
- Événements
- Finaliseurs
- Indexeurs

- Opérateurs
- Types imbriqués

Pour plus d'informations, consultez Membres.

Accessibilité

Certaines méthodes et propriétés sont censées être appelées ou accessibles à partir du code situé en dehors d'une classe ou d'un struct, connu sous le nom de *code client*. D'autres méthodes et propriétés peuvent être uniquement utilisables dans la classe ou le struct proprement dits. Il est important de limiter l'accessibilité de votre code afin que seul le code client prévu puisse y accéder. Vous spécifiez la façon dont vos types et leurs membres sont accessibles au code client à l'aide des modificateurs d'accès suivants :

- public
- protected
- internal
- protected internal
- private
- private protected

L'accessibilité par défaut est `private`.

Héritage

Les classes (mais pas les structs) prennent en charge le concept d'héritage. Une classe qui dérive d'une autre classe, appelée *classe de base*, contient automatiquement tous les membres publics, protégés et internes de la classe de base, sauf ses constructeurs et finaliseurs.

Les classes peuvent être déclarées comme abstraites, ce qui signifie qu'une ou plusieurs de leurs méthodes n'ont aucune implémentation. Bien que les classes abstraites ne puissent pas être instanciées directement, elles peuvent servir de classes de base à d'autres classes qui fournissent l'implémentation manquante. Les classes peuvent également être déclarées comme scellées pour empêcher d'autres classes d'hériter d'elles.

Pour plus d'informations, consultez les pages Héritage et Polymorphisme.

Interfaces

Les classes, les structs et les enregistrements peuvent implémenter plusieurs interfaces. L'implémentation à partir d'une interface signifie que le type implémente toutes les méthodes définies dans l'interface. Pour plus d'informations, consultez Interfaces.

Types génériques

Les classes, structs et enregistrements peuvent être définis avec un ou plusieurs paramètres de type. Le code client fournit le type lorsqu'il crée une instance du type. Par exemple, la classe List<T> de l'espace de noms System.Collections.Generic est définie avec un seul paramètre de type. Le code client crée une instance d'une `List<string>` ou d'une `List<int>` pour spécifier le type que contiendra la liste. Pour plus d'informations, consultez Génériques.

Types statiques

Les classes (mais pas les structs ni les enregistrements) peuvent être déclarées comme `static`. Une classe statique ne peut contenir que des membres statiques et ne peut pas être instanciée avec le mot clé `new`. Une copie de la classe est chargée en mémoire au chargement du programme, et ses membres sont accessibles par le biais du nom de la classe. Les classes, les structs et les enregistrements peuvent contenir des membres statiques. Pour plus d'informations, consultez Classes static et membres de classe static.

Types imbriqués

Une classe, un struct et un enregistrement peuvent être imbriqués dans d'autres classe, struct et enregistrement. Pour plus d'informations, consultez Types imbriqués.

Types partiels

Vous pouvez définir une partie d'une classe, d'un struct ou d'une méthode dans un fichier de code et une autre partie dans un autre fichier de code. Pour plus d'informations, consultez la page Classes et méthodes partielles.

Initialiseurs d'objets

Vous pouvez instancier et initialiser des objets de classe ou de struct, ainsi que des collections d'objets, en affectant des valeurs aux propriétés. Pour plus d'informations, consultez Comment initialiser des objets à l'aide d'un initialiseur d'objet.

Types anonymes

Dans les situations où il n'est pas pratique ou nécessaire de créer une classe nommée, vous utilisez des types anonymes. Les types anonymes sont définis par leurs membres de données nommés. Pour plus d'informations, consultez Types anonymes.

Méthodes d'extension

Vous pouvez « étendre » une classe sans créer de classe dérivée en créant un type distinct. Ce type contient des méthodes qui peuvent être appelées comme si elles appartenaient au type d'origine. Pour plus d'informations, consultez Méthodes d'extension.

Variables locales implicitement typées

Dans une méthode de classe ou de struct, vous pouvez utiliser le typage implicite pour indiquer au compilateur de déterminer le type d'une variable au moment de la compilation. Pour plus d'informations, consultez var (Référence C#).

Enregistrements

C# 9 introduit le type `record`, un type référence que vous pouvez créer à la place d'une classe ou d'un struct. Les enregistrements sont des classes avec un comportement intégré pour encapsuler des données dans des types immuables. C# 10 introduit le type valeur `record struct`. Un enregistrement (`record class` ou `record struct`) fournit les fonctionnalités suivantes :

- Syntaxe concise pour la création d'un type référence avec des propriétés immuables.
- Égalité des valeurs. Deux variables d'un type d'enregistrement sont égales si elles ont le même type et si, pour chaque champ, les valeurs des deux enregistrements sont égales. Les classes utilisent l'égalité de référence : deux variables d'un type de classe sont égales si elles font référence au même objet.
- Syntaxe concise pour une mutation non destructrice. Une expression `with` vous permet de créer une nouvelle instance d'enregistrement qui est une copie d'une instance existante, mais avec des valeurs de propriété spécifiées modifiées.
- Mise en forme intégrée pour l'affichage. La méthode `ToString` imprime le nom du type d'enregistrement ainsi que les noms et les valeurs des propriétés publiques.

- Prise en charge des hiérarchies d'héritage dans les classes d'enregistrement. Les classes d'enregistrement prennent en charge l'héritage. Les structs d'enregistrement ne prennent pas en charge l'héritage.

Pour plus d'informations, consultez Enregistrements.

Spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Objets : créer des instances de types

Article • 10/05/2023

Une définition de classe ou de struct s'apparente à un plan qui spécifie ce que le type peut faire. Un objet est fondamentalement un bloc de mémoire qui a été alloué et configuré selon le plan. Un programme peut créer de nombreux objets de la même classe. Les objets sont également appelés instances. Ils peuvent être stockés dans une variable nommée, dans un tableau ou dans une collection. Le code client est le code qui utilise ces variables pour appeler les méthodes et accéder aux propriétés publiques de l'objet. Dans un langage orienté objet tel que C#, un programme classique se compose de plusieurs objets qui interagissent de façon dynamique.

> ⓘ **Notes**
>
> Les types statiques se comportent différemment de ce qui est décrit ici. Pour plus d'informations, consultez la page **Classes statiques et membres de classes statiques**.

Instances de struct et instances de classe

Étant donné que les classes sont des types référence, une variable d'un objet de classe conserve une référence à l'adresse de l'objet sur le tas managé. Si une deuxième variable du même type est assignée à la première variable, les deux variables font référence à l'objet à cette adresse. Ce point est abordé en détail plus loin dans cet article.

Les instances de classes sont créées à l'aide de l'opérateur new. Dans l'exemple suivant, `Person` est le type, et `person1` et `person2` sont des instances, ou objets, de ce type.

C#

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
    // Other properties, methods, events...
}
```

```
class Program
{
    static void Main()
    {
        Person person1 = new Person("Leopold", 6);
        Console.WriteLine("person1 Name = {0} Age = {1}", person1.Name,
person1.Age);

        // Declare new person, assign person1 to it.
        Person person2 = person1;

        // Change the name of person2, and person1 also changes.
        person2.Name = "Molly";
        person2.Age = 16;

        Console.WriteLine("person2 Name = {0} Age = {1}", person2.Name,
person2.Age);
        Console.WriteLine("person1 Name = {0} Age = {1}", person1.Name,
person1.Age);
    }
}
/*
    Output:
    person1 Name = Leopold Age = 6
    person2 Name = Molly Age = 16
    person1 Name = Molly Age = 16
*/
```

Étant donné que les structs sont des types valeur, une variable d'un objet de struct conserve une copie de l'objet entier. Les instances de structs peuvent également être créées à l'aide de l'opérateur `new`, mais cela n'est pas obligatoire, comme illustré dans l'exemple suivant :

C#

```
namespace Example;

public struct Person
{
    public string Name;
    public int Age;
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
}

public class Application
{
    static void Main()
```

```
    {
        // Create  struct instance and initialize by using "new".
        // Memory is allocated on thread stack.
        Person p1 = new Person("Alex", 9);
        Console.WriteLine("p1 Name = {0} Age = {1}", p1.Name, p1.Age);

        // Create  new struct object. Note that  struct can be initialized
        // without using "new".
        Person p2 = p1;

        // Assign values to p2 members.
        p2.Name = "Spencer";
        p2.Age = 7;
        Console.WriteLine("p2 Name = {0} Age = {1}", p2.Name, p2.Age);

        // p1 values remain unchanged because p2 is  copy.
        Console.WriteLine("p1 Name = {0} Age = {1}", p1.Name, p1.Age);
    }
}
/*
    Output:
    p1 Name = Alex Age = 9
    p2 Name = Spencer Age = 7
    p1 Name = Alex Age = 9
*/
```

La mémoire pour `p1` et `p2` est allouée sur la pile de threads. Cette mémoire est récupérée avec le type ou la méthode où elle est déclarée. C'est l'une des raisons pour lesquelles les structs sont copiés au moment de l'assignation. En revanche, la mémoire allouée pour une instance de classe est récupérée automatiquement (garbage collection) par le common language runtime quand toutes les références à l'objet sont hors de la portée. Il n'est pas possible de détruire de façon déterministe un objet de classe comme vous pouvez le faire dans C++. Pour plus d'informations sur l'opération de garbage collection dans .NET, consultez Garbage Collection.

> ⓘ **Notes**
>
> L'allocation et la libération de mémoire sur le tas managé sont des opérations très optimisées dans le common language runtime. Dans la plupart des cas, il n'y a pas de différence significative sur le plan des performances entre l'allocation d'une instance de classe sur le tas et l'allocation d'une instance de struct sur la pile.

Identité d'objet et égalité des valeurs

Quand vous effectuez une comparaison d'égalité entre deux objets, vous devez d'abord décider si vous souhaitez savoir si les deux variables représentent le même objet en

mémoire, ou si les valeurs d'un ou de plusieurs de leurs champs sont équivalentes. Si vous projetez de comparer des valeurs, vous devez établir si les objets sont des instances de types valeur (structs) ou de types référence (classes, délégués, tableaux).

- Pour déterminer si deux instances de classe référencent le même emplacement en mémoire (ce qui signifie qu'elles ont la même *identité*), utilisez la méthode statique Object.Equals. (System.Object est la classe de base implicite pour tous les types valeur et types référence, y compris les classes et structs définis par l'utilisateur.)

- Pour déterminer si les champs d'instance dans deux instances de struct ont les mêmes valeurs, utilisez la méthode ValueType.Equals. Comme tous les structs héritent implicitement de System.ValueType, vous appelez directement la méthode sur votre objet, comme indiqué dans l'exemple suivant :

  C#

  ```
  // Person is defined in the previous example.

  //public struct Person
  //{
  //    public string Name;
  //    public int Age;
  //    public Person(string name, int age)
  //    {
  //        Name = name;
  //        Age = age;
  //    }
  //}

  Person p1 = new Person("Wallace", 75);
  Person p2 = new Person("", 42);
  p2.Name = "Wallace";
  p2.Age = 75;

  if (p2.Equals(p1))
      Console.WriteLine("p2 and p1 have the same values.");

  // Output: p2 and p1 have the same values.
  ```

  L'implémentation System.ValueType de `Equals` utilise le boxing et la réflexion dans certains cas. Pour plus d'informations sur la façon de fournir un algorithme d'égalité efficace spécifique à votre type, consultez Comment définir l'égalité des valeurs pour un type. Les enregistrements sont des types de référence qui utilisent la sémantique de valeur pour l'égalité.

- Pour déterminer si les valeurs des champs dans deux instances de classe sont égales, vous pouvez utiliser la méthode Equals ou l'opérateur ==. Toutefois,

utilisez-les uniquement si la classe les a substitués ou surchargés pour fournir une définition personnalisée de ce que signifie « égalité » pour les objets de ce type. La classe peut également implémenter l'interface IEquatable<T> ou IEqualityComparer<T>. Les deux interfaces fournissent des méthodes qui peuvent être utilisées pour tester l'égalité des valeurs. Quand vous concevez vos propres classes qui substituent `Equals`, suivez les instructions indiquées dans Guide pratique pour définir l'égalité de valeurs pour un type et Object.Equals(Object).

Sections connexes

Pour plus d'informations :

- Classes
- Constructeurs
- Finaliseurs
- Événements
- object
- Héritage
- class
- Types de structure
- new, opérateur
- Système de type commun

Héritage : dériver des types pour créer un comportement plus spécialisé

Article • 07/04/2023

L'héritage, avec l'encapsulation et le polymorphisme, est l'une des trois principales caractéristiques de la programmation orientée objet. L'héritage vous permet de créer de nouvelles classes qui réutilisent, étendent et modifient le comportement défini dans les autres classes. La classe dont les membres sont hérités porte le nom de *classe de base* et la classe qui hérite de ces membres porte le nom de *classe dérivée*. Une classe dérivée ne peut avoir qu'une seule classe de base directe. Toutefois, l'héritage est transitif. Si `ClassC` est dérivé de `ClassB` et que `ClassB` est dérivé de `ClassA`, `ClassC` hérite des membres déclarés dans `ClassB` et `ClassA`.

> **Notes**
>
> Les structs ne prennent pas en charge l'héritage, mais ils peuvent implémenter les interfaces.

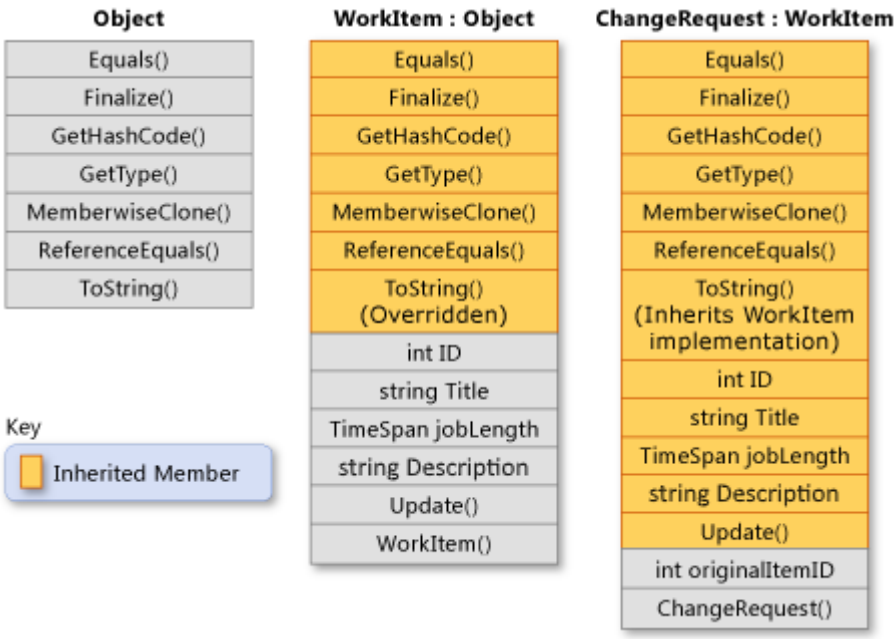
D'un point de vue conceptuel, une classe dérivée est une spécialisation de la classe de base. Par exemple, si vous avez une classe de base `Animal`, vous pouvez avoir une classe dérivée nommée `Mammal` et une autre classe dérivée nommée `Reptile`. Un `Mammal` est un `Animal`, et un `Reptile` est un `Animal`, mais chaque classe dérivée représente des spécialisations différentes de la classe de base.

Les déclarations d'interface peuvent définir une implémentation par défaut pour ses membres. Ces implémentations sont héritées par les interfaces dérivées et par les classes qui implémentent ces interfaces. Pour plus d'informations sur les méthodes d'interface par défaut, consultez l'article sur les interfaces.

Quand vous définissez une classe à dériver d'une autre classe, la classe dérivée obtient implicitement tous les membres de la classe de base, à l'exception de ses constructeurs et de ses finaliseurs. La classe dérivée réutilise le code de la classe de base sans avoir à le réimplémenter. Vous pouvez ajouter d'autres membres à la classe dérivée. La classe dérivée étend les fonctionnalités de la classe de base.

L'illustration suivante montre une classe `WorkItem` qui représente un élément de travail dans un processus métier. Comme toutes les classes, elle dérive de System.Object et hérite de toutes ses méthodes. `WorkItem` ajoute six de ses propres membres. Ces membres incluent un constructeur, parce que les constructeurs ne sont pas hérités. La

classe `ChangeRequest` hérite de `WorkItem` et représente un type particulier d'élément de travail. `ChangeRequest` ajoute deux membres supplémentaires aux membres qu'elle hérite de `WorkItem` et de Object. Elle doit ajouter son propre constructeur et ajoute également `originalItemID`. La propriété `originalItemID` permet à l'instance `ChangeRequest` d'être associée au `WorkItem` d'origine auquel s'applique la demande de modification.



L'exemple suivant montre comment les relations de classe de l'illustration précédente sont exprimées en langage C#. L'exemple montre également comment `WorkItem` substitue la méthode virtuelle Object.ToString, et comment la classe `ChangeRequest` hérite de l'implémentation `WorkItem` de la méthode. Le premier bloc définit les classes :

C#

```
// WorkItem implicitly inherits from the Object class.
public class WorkItem
{
    // Static field currentID stores the job ID of the last WorkItem that
    // has been created.
    private static int currentID;

    //Properties.
    protected int ID { get; set; }
    protected string Title { get; set; }
    protected string Description { get; set; }
    protected TimeSpan jobLength { get; set; }

    // Default constructor. If a derived class does not invoke a base-
    // class constructor explicitly, the default constructor is called
    // implicitly.
    public WorkItem()
    {
```

```
        ID = 0;
        Title = "Default title";
        Description = "Default description.";
        jobLength = new TimeSpan();
    }

    // Instance constructor that has three parameters.
    public WorkItem(string title, string desc, TimeSpan joblen)
    {
        this.ID = GetNextID();
        this.Title = title;
        this.Description = desc;
        this.jobLength = joblen;
    }

    // Static constructor to initialize the static member, currentID. This
    // constructor is called one time, automatically, before any instance
    // of WorkItem or ChangeRequest is created, or currentID is referenced.
    static WorkItem() => currentID = 0;

    // currentID is a static field. It is incremented each time a new
    // instance of WorkItem is created.
    protected int GetNextID() => ++currentID;

    // Method Update enables you to update the title and job length of an
    // existing WorkItem object.
    public void Update(string title, TimeSpan joblen)
    {
        this.Title = title;
        this.jobLength = joblen;
    }

    // Virtual method override of the ToString method that is inherited
    // from System.Object.
    public override string ToString() =>
        $"{this.ID} - {this.Title}";
}

// ChangeRequest derives from WorkItem and adds a property (originalItemID)
// and two constructors.
public class ChangeRequest : WorkItem
{
    protected int originalItemID { get; set; }

    // Constructors. Because neither constructor calls a base-class
    // constructor explicitly, the default constructor in the base class
    // is called implicitly. The base class must contain a default
    // constructor.

    // Default constructor for the derived class.
    public ChangeRequest() { }

    // Instance constructor that has four parameters.
    public ChangeRequest(string title, string desc, TimeSpan jobLen,
                         int originalID)
```

```
    {
        // The following properties and the GetNexID method are inherited
        // from WorkItem.
        this.ID = GetNextID();
        this.Title = title;
        this.Description = desc;
        this.jobLength = jobLen;

        // Property originalItemID is a member of ChangeRequest, but not
        // of WorkItem.
        this.originalItemID = originalID;
    }
}
```

Ce bloc suivant montre comment utiliser les classes de base et dérivées :

C#

```
// Create an instance of WorkItem by using the constructor in the
// base class that takes three arguments.
WorkItem item = new WorkItem("Fix Bugs",
                             "Fix all bugs in my code branch",
                             new TimeSpan(3, 4, 0, 0));

// Create an instance of ChangeRequest by using the constructor in
// the derived class that takes four arguments.
ChangeRequest change = new ChangeRequest("Change Base Class Design",
                                         "Add members to the class",
                                         new TimeSpan(4, 0, 0),
                                         1);

// Use the ToString method defined in WorkItem.
Console.WriteLine(item.ToString());

// Use the inherited Update method to change the title of the
// ChangeRequest object.
change.Update("Change the Design of the Base Class",
    new TimeSpan(4, 0, 0));

// ChangeRequest inherits WorkItem's override of ToString.
Console.WriteLine(change.ToString());
/* Output:
    1 - Fix Bugs
    2 - Change the Design of the Base Class
*/
```

Méthodes abstraites et virtuelles

Quand une classe de base déclare une méthode comme virtual (virtuelle), une classe dérivée peut remplacer (override) la méthode avec sa propre implémentation. Si une

classe de base déclare un membre comme abstract (abstraite), la méthode doit être remplacée dans toutes les classes non abstraites qui héritent directement de cette classe. Si une classe dérivée est abstraite, elle hérite des membres abstraits sans les implémenter. Les membres virtuels et abstraits sont la base du polymorphisme, qui est la deuxième caractéristique principale de la programmation orientée objet. Pour plus d'informations, consultez Polymorphisme.

Classes de base abstraites

Pour empêcher l'instanciation directe, vous pouvez déclarer une classe comme étant abstract à l'aide de l'opérateur new. Une classe abstraite peut être utilisée seulement si une nouvelle classe en est dérivée. Une classe abstraite peut contenir une ou plusieurs signatures de méthode qui sont également déclarées comme abstraites. Ces signatures spécifient les paramètres et la valeur de retour, mais n'ont aucune implémentation (corps de méthode). Une classe abstraite ne doit pas nécessairement contenir des membres abstraits. Toutefois, si une classe contient un membre abstrait, elle doit être déclarée comme abstraite. Les classes dérivées qui ne sont pas abstraites doivent fournir l'implémentation pour toutes les méthodes abstraites d'une classe de base abstraite.

Interfaces

Une *interface* est un type référence qui définit un ensemble de membres. Toutes les classes et tous les structs qui implémentent cette interface doivent implémenter cet ensemble de membres. Une interface peut définir une implémentation par défaut pour tout ou partie de ces membres. Une classe peut implémenter plusieurs interfaces, même si elle ne peut dériver que d'une seule classe de base directe.

Les interfaces sont utilisées pour définir des fonctions spécifiques pour les classes qui n'ont pas nécessairement de relation de type « est un ». Par exemple, l'interface System.IEquatable<T> peut être implémentée par toute classe ou tout struct pour déterminer si deux objets de ce type sont équivalents (quelle que soit la manière dont le type définit l'équivalence). IEquatable<T> n'implique pas le même type de relation « est un » qui existe entre une classe de base et une classe dérivée (par exemple, un `Mammal` (mammifère) est un `Animal`). Pour plus d'informations, consultez Interfaces.

Empêcher une dérivation supplémentaire

Une classe peut empêcher d'autres classes d'hériter d'elle, ou d'un de ses membres, en se déclarant elle-même ou en déclarant le membre comme sealed (scellé).

Masquage des membres de la classe de base par une classe dérivée

Une classe dérivée peut masquer des membres de la classe de base en déclarant les membres à l'aide du même nom et de la même signature. Le modificateur new peut être utilisé pour indiquer explicitement que ce membre n'est pas censé substituer le membre de base. L'utilisation de new n'est pas obligatoire, mais un avertissement du compilateur est généré si new n'est pas utilisé. Pour plus d'informations, consultez Gestion de version avec les mots clés override et new et Savoir quand utiliser les mots clés override et new.

Polymorphisme

Article • 07/04/2023

Le polymorphisme est souvent considéré comme le troisième pilier d'une programmation orientée objet, après l'encapsulation et l'héritage. Le polymorphisme est le mot grec qui signifie « plusieurs formes » et il prend deux aspects distincts :

- Au moment de l'exécution, les objets d'une classe dérivée peuvent être traités comme des objets d'une classe de base dans les paramètres de méthode et les collections ou les tableaux. Lorsque ce polymorphisme intervient, le type déclaré de l'objet n'est plus identique à son type au moment de l'exécution.
- Les classes de base peuvent définir et implémenter des *méthodes*virtuelles, et les classes dérivées peuvent les substituer, ce qui signifie qu'elles fournissent leur propre définition et implémentation. Au moment de l'exécution, quand le code client appelle la méthode, le CLR recherche le type au moment de l'exécution et appelle cette substitution de la méthode virtuelle. Dans le code source, vous pouvez appeler une méthode dans une classe de base, et provoquer l'exécution d'une version de la classe dérivée de la méthode.

Les méthodes virtuelles vous permettent d'utiliser des groupes d'objets liés de façon uniforme. Par exemple, si vous avez une application de dessin qui permet à un utilisateur de créer différents types de formes sur une surface de dessin. Vous ne savez pas au moment de la compilation les types de formes spécifiques que l'utilisateur va créer. Cependant, l'application doit conserver une trace des différents types de formes créés et les mettre à jour en réponse aux actions de la souris. Vous pouvez utiliser le polymorphisme pour résoudre ce problème en deux étapes :

1. Créer une hiérarchie de classe dans laquelle chaque classe de forme dérive d'une classe de base commune.
2. Utiliser une méthode virtuelle pour appeler la méthode appropriée dans une classe dérivée via un seul appel à la méthode de classe de base.

Tout d'abord, créez une classe de base appelée `Shape`, et des classes dérivées telles que `Rectangle`, `Circle`, et `Triangle`. Donnez à la classe `Shape` une méthode virtuelle appelée `Draw`, et substituez-la dans chaque classe dérivée pour dessiner une forme spécifique représentée par la classe. Créez un objet `List<Shape>` et ajoutez-lui un `Circle`, un `Triangle` et un `Rectangle`.

C#

```
public class Shape
{
```

```
    // A few example members
    public int X { get; private set; }
    public int Y { get; private set; }
    public int Height { get; set; }
    public int Width { get; set; }

    // Virtual method
    public virtual void Draw()
    {
        Console.WriteLine("Performing base class drawing tasks");
    }
}

public class Circle : Shape
{
    public override void Draw()
    {
        // Code to draw a circle...
        Console.WriteLine("Drawing a circle");
        base.Draw();
    }
}
public class Rectangle : Shape
{
    public override void Draw()
    {
        // Code to draw a rectangle...
        Console.WriteLine("Drawing a rectangle");
        base.Draw();
    }
}
public class Triangle : Shape
{
    public override void Draw()
    {
        // Code to draw a triangle...
        Console.WriteLine("Drawing a triangle");
        base.Draw();
    }
}
```

Pour mettre à jour la surface de dessin, utilisez une boucle foreach pour l'itération dans la liste et appelez la méthode `Draw` sur chaque objet `Shape` de la liste. Même si chaque objet de la liste a un type déclaré égal à `Shape`, il s'agit du type au moment de l'exécution (la version substituée de la méthode dans chaque classe dérivée) qui est appelé.

C#

```
// Polymorphism at work #1: a Rectangle, Triangle and Circle
// can all be used wherever a Shape is expected. No cast is
// required because an implicit conversion exists from a derived
```

```
// class to its base class.
var shapes = new List<Shape>
{
    new Rectangle(),
    new Triangle(),
    new Circle()
};

// Polymorphism at work #2: the virtual method Draw is
// invoked on each of the derived classes, not the base class.
foreach (var shape in shapes)
{
    shape.Draw();
}
/* Output:
    Drawing a rectangle
    Performing base class drawing tasks
    Drawing a triangle
    Performing base class drawing tasks
    Drawing a circle
    Performing base class drawing tasks
*/
```

En C#, chaque type est polymorphique, car tous les types, y compris les types définis par l'utilisateur, héritent de Object.

Vue d'ensemble du polymorphisme

Membres virtuels

Lorsqu'une classe dérivée hérite d'une classe de base, elle inclut tous les membres de la classe de base. Tout le comportement déclaré dans la classe de base fait partie de la classe dérivée. Ainsi, les objets de la classe dérivée sont traités comme les objets de la classe de base. Les modificateurs d'accès (`public`, `protected`, `private`, etc.) déterminent si ces membres sont accessibles à partir de l'implémentation de classe dérivée. Les méthodes virtuelles offrent au concepteur différents choix en termes de comportement de la classe dérivée :

- La classe dérivée peut remplacer les membres virtuels dans la classe de base, en définissant un nouveau comportement.
- La classe dérivée peut hériter de la méthode de classe de base la plus proche sans la remplacer, en conservant le comportement existant, mais en permettant à d'autres classes dérivées de remplacer la méthode.
- La classe dérivée peut définir une nouvelle implémentation non virtuelle de ces membres qui masque les implémentations de la classe de base.

Une classe dérivée ne peut substituer un membre de classe de base que si le membre de classe de base est déclaré comme étant virtual ou abstract. Le membre dérivé doit utiliser le mot clé override pour indiquer explicitement que la méthode est conçue pour participer à l'appel virtuel. Le code suivant est fourni à titre d'exemple :

C#

```
public class BaseClass
{
    public virtual void DoWork() { }
    public virtual int WorkProperty
    {
        get { return 0; }
    }
}
public class DerivedClass : BaseClass
{
    public override void DoWork() { }
    public override int WorkProperty
    {
        get { return 0; }
    }
}
```

Les champs ne peuvent pas être virtuels ; seuls les méthodes, propriétés, événements et indexeurs peuvent être virtuels. Quand une classe dérivée est substituée à un membre virtuel, ce membre est appelé lors de l'accès à une instance de cette classe en tant qu'instance de la classe de base. Le code suivant est fourni à titre d'exemple :

C#

```
DerivedClass B = new DerivedClass();
B.DoWork();  // Calls the new method.

BaseClass A = B;
A.DoWork();  // Also calls the new method.
```

Les méthodes et propriétés virtuelles permettent aux classes dérivées d'étendre une classe de base sans avoir besoin d'utiliser l'implémentation de classe de base d'une méthode. Pour plus d'informations, consultez Versioning avec les mots clés override et new. Une interface fournit un autre moyen pour définir une méthode ou un ensemble de méthodes dont l'implémentation est confiée aux classes dérivées.

Masquer des membres de classe de base par de nouveaux membres

Si vous souhaitez que votre classe dérivée ait un membre portant le même nom qu'un membre de classe de base, vous pouvez utiliser le nouveau mot clé pour masquer le membre de classe de base. Le mot clé `new` est placé avec le type de retour d'un membre de classe qui est remplacé. Le code suivant est fourni à titre d'exemple :

C#

```
public class BaseClass
{
    public void DoWork() { WorkField++; }
    public int WorkField;
    public int WorkProperty
    {
        get { return 0; }
    }
}

public class DerivedClass : BaseClass
{
    public new void DoWork() { WorkField++; }
    public new int WorkField;
    public new int WorkProperty
    {
        get { return 0; }
    }
}
```

Les membres de classe de base masqués sont accessibles à partir du code client en effectuant un cast de l'instance de la classe dérivée vers une instance de la classe de base. Par exemple :

C#

```
DerivedClass B = new DerivedClass();
B.DoWork();  // Calls the new method.

BaseClass A = (BaseClass)B;
A.DoWork();  // Calls the old method.
```

Empêcher les classes dérivées de remplacer les membres virtuels

Les membres virtuels restent virtuels, quel que soit le nombre de classes déclarées entre le membre virtuel et la classe qui l'a déclaré à l'origine. Si la classe `A` déclare un membre virtuel, et la classe `B` dérive de `A`, et la classe `C` dérive de `B`, la classe `C` hérite du

membre virtuel et peut le remplacer, que la classe `B` ait ou non déclaré une substitution pour ce membre. Le code suivant est fourni à titre d'exemple :

```
C#

public class A
{
    public virtual void DoWork() { }
}
public class B : A
{
    public override void DoWork() { }
}
```

Une classe dérivée peut arrêter l'héritage virtuel en déclarant une substitution comme étant sealed. L'arrêt de l'héritage nécessite l'ajout du mot clé `sealed` avant le mot clé `override` dans la déclaration de membre de classe. Le code suivant est fourni à titre d'exemple :

```
C#

public class C : B
{
    public sealed override void DoWork() { }
}
```

Dans l'exemple précédent, la méthode `DoWork` n'est plus virtuelle pour les classes dérivées de `C`. Elle est toujours virtuelle pour les instances de `C`, même si elles sont castées en type `B` ou `A`. Les méthodes sealed peuvent être remplacées par des classes dérivées en utilisant le mot clé `new`, comme le montre l'exemple suivant :

```
C#

public class D : C
{
    public new void DoWork() { }
}
```

Dans ce cas, si `DoWork` est appelé sur `D` à l'aide d'une variable de type `D`, le nouveau `DoWork` est appelé. Si une variable de type `C`, `B` ou `A` est utilisée pour accéder à une instance de `D`, un appel à `DoWork` suivra les règles de l'héritage virtuel, en routant ces appels vers l'implémentation de `DoWork` dans la classe `C`.

Accéder aux membres virtuels de la classe de base à partir de classes dérivées

Une classe dérivée qui a remplacé ou écrasé une méthode ou une propriété peut encore accéder à la méthode ou à la propriété dans la classe de base avec le mot clé `base`. Le code suivant est fourni à titre d'exemple :

C#

```
public class Base
{
    public virtual void DoWork() {/*...*/ }
}
public class Derived : Base
{
    public override void DoWork()
    {
        //Perform Derived's work here
        //...
        // Call DoWork on base class
        base.DoWork();
    }
}
```

Pour plus d'informations, consultez base.

> ⓘ **Notes**
>
> Il est recommandé que les membres virtuels utilisent `base` pour appeler l'implémentation de classe de base de ce membre dans leur propre implémentation. L'exécution du comportement de classe de base permet à la classe dérivée de se concentrer sur l'implémentation du comportement spécifique à la classe dérivée. Si l'implémentation de classe de base n'est pas appelée, la classe dérivée doit rendre son comportement compatible avec le comportement de la classe de base.

Vue d’ensemble des critères spéciaux

Article • 23/06/2023

Les critères spéciaux sont une technique dans laquelle vous testez une expression pour déterminer si elle présente certaines caractéristiques. Les critères spéciaux C# fournissent une syntaxe plus concise pour tester des expressions et prendre des mesures lorsqu’une expression correspond. L’ « expressionis » prend en charge les critères spéciaux pour tester une expression et déclarer conditionnellement une nouvelle variable au résultat de cette expression. L' « expression switch » vous permet d’effectuer des actions basées sur les premiers critères spéciaux pour une expression. Ces deux expressions prennent en charge un vocabulaire riche de *modèles*.

Cet article fournit une vue d’ensemble des scénarios dans lesquels vous pouvez utiliser les critères spéciaux. Ces techniques peuvent améliorer la lisibilité et l’exactitude de votre code. Pour une présentation complète de tous les modèles que vous pouvez appliquer, consultez l’article sur les modèles dans la référence sur le langage.

Vérifications de la valeur Null

L’un des scénarios les plus courants de critères spéciaux consiste à s’assurer que les valeurs ne sont pas `null`. Vous pouvez tester et convertir un type de valeur acceptant Null en son type sous-jacent tout en testant `null` en utlisant de l’exemple suivant :

C#

```
int? maybe = 12;

if (maybe is int number)
{
    Console.WriteLine($"The nullable int 'maybe' has the value {number}");
}
else
{
    Console.WriteLine("The nullable int 'maybe' doesn't hold a value");
}
```

Le code précédent est un *modèle de déclaration* pour tester le type de la variable et l’affecter à une nouvelle variable. Les règles du langage rendent cette technique plus sûre que bien d’autres. La variable `number` est uniquement accessible et affectée dans la partie vraie de la clause `if`. Si vous essayez d’y accéder ailleurs, soit dans la clause `else`, soit après le bloc `if`, le compilateur émet une erreur. Deuxièmement, comme vous n’utilisez pas l’opérateur `==`, ce modèle fonctionne quand un type surcharge l’opérateur

`==`. Cela en fait un moyen idéal de vérifier les valeurs de référence null, en ajoutant le modèle `not` :

C#

```
string? message = "This is not the null string";

if (message is not null)
{
    Console.WriteLine(message);
}
```

L'exemple précédent utilisait un *modèle constant* pour comparer la variable à `null`. `not` est un *modèle logique* qui correspond lorsque le modèle annulé ne correspond pas.

Tests de type

Une autre utilisation courante pour les critères spéciaux consiste à tester une variable pour voir si elle correspond à un type donné. Par exemple, le code suivant teste si une variable n'est pas null et implémente l'interface System.Collections.Generic.IList<T>. Si c'est le cas, il utilise la propriété ICollection<T>.Count de cette liste pour rechercher l'index central. Le modèle de déclaration ne correspond pas à une valeur `null`, quel que soit le type de compilation de la variable. Le code ci-dessous protège contre `null`, en plus de la protection contre un type qui n'implémente pas `IList`.

C#

```
public static T MidPoint<T>(IEnumerable<T> sequence)
{
    if (sequence is IList<T> list)
    {
        return list[list.Count / 2];
    }
    else if (sequence is null)
    {
        throw new ArgumentNullException(nameof(sequence), "Sequence can't be
null.");
    }
    else
    {
        int halfLength = sequence.Count() / 2 - 1;
        if (halfLength < 0) halfLength = 0;
        return sequence.Skip(halfLength).First();
    }
}
```

Les mêmes tests peuvent être appliqués dans une expression `switch` pour tester une variable sur plusieurs types différents. Vous pouvez utiliser ces informations pour créer de meilleurs algorithmes en fonction du type d'exécution spécifique.

Comparer des valeurs discrètes

Vous pouvez également tester une variable pour trouver une correspondance sur des valeurs spécifiques. Le code suivant montre un exemple où vous testez une valeur par rapport à toutes les valeurs possibles déclarées dans une énumération :

C#

```
public State PerformOperation(Operation command) =>
   command switch
   {
       Operation.SystemTest => RunDiagnostics(),
       Operation.Start => StartSystem(),
       Operation.Stop => StopSystem(),
       Operation.Reset => ResetToReady(),
       _ => throw new ArgumentException("Invalid enum value for command",
nameof(command)),
   };
```

L'exemple précédent illustre un dispatch de méthode basé sur la valeur d'une énumération. Le dernier cas `_` est un *modèle discard* qui correspond à toutes les valeurs. Il gère toutes les conditions d'erreur où la valeur ne correspond pas à l'une des valeurs définies `enum`. Si vous omettez ce bras de commutateur, le compilateur avertit que vous n'avez pas géré toutes les valeurs d'entrée possibles. Au moment de l'exécution, l'expression `switch` lève une exception si l'objet examiné ne correspond à aucun des bras de commutateur. Vous pouvez utiliser des constantes numériques au lieu d'un ensemble de valeurs d'énumération. Vous pouvez également utiliser cette technique similaire pour les valeurs de chaîne constante qui représentent les commandes :

C#

```
public State PerformOperation(string command) =>
   command switch
   {
       "SystemTest" => RunDiagnostics(),
       "Start" => StartSystem(),
       "Stop" => StopSystem(),
       "Reset" => ResetToReady(),
       _ => throw new ArgumentException("Invalid string value for command",
nameof(command)),
   };
```

L'exemple précédent montre le même algorithme, mais utilise des valeurs de chaîne au lieu d'une énumération. Vous pouvez utiliser ce scénario si votre application répond à des commandes de texte au lieu d'un format de données standard. À compter de C# 11, vous pouvez également utiliser `Span<char>` ou `ReadOnlySpan<char>` pour tester les valeurs de chaîne constante, comme indiqué dans l'exemple suivant :

```
C#
```

```
public State PerformOperation(ReadOnlySpan<char> command) =>
   command switch
   {
       "SystemTest" => RunDiagnostics(),
       "Start" => StartSystem(),
       "Stop" => StopSystem(),
       "Reset" => ResetToReady(),
       _ => throw new ArgumentException("Invalid string value for command",
nameof(command)),
   };
```

Dans tous ces exemples, le *modèle discard* garantit que vous gérez chaque entrée. Le compilateur vous aide en vous assurant que chaque valeur d'entrée possible est gérée.

Modèles relationnels

Vous pouvez utiliser *des modèles relationnels* pour tester la comparaison d'une valeur à des constantes. Par exemple, le code suivant retourne l'état de l'eau en fonction de la température en Fahrenheit :

```
C#
```

```
string WaterState(int tempInFahrenheit) =>
    tempInFahrenheit switch
    {
        (> 32) and (< 212) => "liquid",
        < 32 => "solid",
        > 212 => "gas",
        32 => "solid/liquid transition",
        212 => "liquid / gas transition",
    };
```

Le code précédent illustre également le `and` *modèle logique* conjonctif pour vérifier que les deux modèles relationnels correspondent. Vous pouvez également utiliser un modèle disjonctif `or` pour vérifier que l'un ou l'autre modèle correspond. Les deux modèles relationnels sont entourés de parenthèses, que vous pouvez utiliser autour de n'importe quel modèle pour plus de clarté. Les deux derniers bras de commutateur gèrent les cas

pour le point de fusion et le point d'ébullition. Sans ces deux bras, le compilateur vous avertit que votre logique ne couvre pas toutes les entrées possibles.

Le code précédent illustre également une autre fonctionnalité importante que le compilateur fournit pour les expressions de correspondance de modèle : le compilateur vous avertit si vous ne gérez pas chaque valeur d'entrée. Le compilateur émet également un avertissement si un bras de commutateur est déjà géré par un bras de commutateur précédent. Cela vous donne la liberté de refactoriser et de réorganiser les expressions de commutateur. Une autre façon d'écrire la même expression peut être :

C#

```
string WaterState2(int tempInFahrenheit) =>
    tempInFahrenheit switch
    {
        < 32 => "solid",
        32 => "solid/liquid transition",
        < 212 => "liquid",
        212 => "liquid / gas transition",
        _ => "gas",
};
```

La leçon clé de cela et de toute autre refactorisation ou réorganisation est que le compilateur vérifie que vous avez couvert toutes les entrées.

Entrées multiples

Tous les modèles que vous avez vus jusqu'à présent ont vérifié une entrée. Vous pouvez écrire des modèles qui examinent plusieurs propriétés d'un objet. Prenez l'enregistrement `Order` suivant en compte :

C#

```
public record Order(int Items, decimal Cost);
```

Le type d'enregistrement positionnel précédent déclare deux membres à des positions explicites. S'affiche d'abord `Items`, puis la commande `Cost`. Pour plus d'informations, consultez Enregistrements.

Le code suivant examine le nombre d'éléments et la valeur d'une commande pour calculer un prix réduit :

C#

```
public decimal CalculateDiscount(Order order) =>
    order switch
    {
        { Items: > 10, Cost: > 1000.00m } => 0.10m,
        { Items: > 5, Cost: > 500.00m } => 0.05m,
        { Cost: > 250.00m } => 0.02m,
        null => throw new ArgumentNullException(nameof(order), "Can't
calculate discount on null order"),
        var someObject => 0m,
    };
```

Les deux premiers bras examinent deux propriétés de `Order`. Le troisième examine uniquement le coût. La suivante vérifie par rapport à `null`, et la finale correspond à toute autre valeur. Si le type `Order` définit une méthode appropriée Deconstruct, vous pouvez omettre les noms de propriétés du modèle et utiliser la déconstruction pour examiner les propriétés :

C#

```
public decimal CalculateDiscount(Order order) =>
    order switch
    {
        ( > 10,  > 1000.00m) => 0.10m,
        ( > 5, > 50.00m) => 0.05m,
        { Cost: > 250.00m } => 0.02m,
        null => throw new ArgumentNullException(nameof(order), "Can't
calculate discount on null order"),
        var someObject => 0m,
    };
```

Le code précédent illustre le *modèle positionnel* où les propriétés sont déconstruites pour l'expression.

Modèles de liste

Vous pouvez vérifier les éléments d'une liste ou d'un tableau à l'aide d'un *modèle de liste*. Un modèle de liste permet d'appliquer un modèle à n'importe quel élément d'une séquence. En outre, vous pouvez appliquer le *modèle discard* (`_`) pour qu'il corresponde à n'importe quel élément, ou appliquer un *modèle de tranche* pour correspondre à zéro ou plusieurs éléments.

Les modèles de liste sont un outil précieux lorsque les données ne suivent pas une structure régulière. Vous pouvez utiliser les critères spéciaux pour tester la forme et les valeurs des données au lieu de les transformer en un ensemble d'objets.

Considérez l'extrait suivant d'un fichier texte contenant des transactions bancaires :

Sortie

```
04-01-2020, DEPOSIT,    Initial deposit,           2250.00
04-15-2020, DEPOSIT,    Refund,                     125.65
04-18-2020, DEPOSIT,    Paycheck,                   825.65
04-22-2020, WITHDRAWAL, Debit,          Groceries,  255.73
05-01-2020, WITHDRAWAL, #1102,          Rent, apt, 2100.00
05-02-2020, INTEREST,                                 0.65
05-07-2020, WITHDRAWAL, Debit,          Movies,      12.57
04-15-2020, FEE,                                      5.55
```

Il s'agit d'un format CSV, mais certaines lignes ont plus de colonnes que d'autres. Pire encore pour le traitement, une colonne du type `WITHDRAWAL` contient du texte généré par l'utilisateur et peut contenir une virgule dans le texte. Un *modèle de liste* qui comprend le modèle *discard*, le modèle *constant* et le modèle *var* pour capturer les données de la valeur traite les données dans ce format :

C#

```
decimal balance = 0m;
foreach (string[] transaction in ReadRecords())
{
    balance += transaction switch
    {
        [_, "DEPOSIT", _, var amount]     => decimal.Parse(amount),
        [_, "WITHDRAWAL", .., var amount] => -decimal.Parse(amount),
        [_, "INTEREST", var amount]       => decimal.Parse(amount),
        [_, "FEE", var fee]               => -decimal.Parse(fee),
        _                                 => throw new
InvalidOperationException($"Record {string.Join(", ", transaction)} is not
in the expected format!"),
    };
    Console.WriteLine($"Record: {string.Join(", ", transaction)}, New
balance: {balance:C}");
}
```

L'exemple précédent prend un tableau de chaînes, où chaque élément est un champ dans la ligne. Les clés d'expression `switch` sur le deuxième champ, qui détermine le type de transaction et le nombre de colonnes restantes. Chaque ligne garantit que les données sont au bon format. Le modèle discard (`_`) ignore le premier champ, avec la date de la transaction. Le deuxième champ correspond au type de transaction. L'élément restant correspond au champ avec le montant. La correspondance finale utilise le modèle *var* pour capturer la représentation sous forme de chaîne du montant. L'expression calcule le montant à ajouter ou à soustraire du solde.

Les modèles de liste vous permettent de faire correspondre sur la forme d'une séquence d'éléments de données. Vous utilisez les modèles *discard* et *tranche* pour correspondre à l'emplacement des éléments. Vous utilisez d'autres modèles pour faire correspondre les caractéristiques des éléments individuels.

Cet article fournit une présentation des types de code que vous pouvez écrire avec les critères spéciaux en C#. Les articles suivants présentent d'autres exemples d'utilisation de modèles dans des scénarios et le vocabulaire complet des modèles disponibles.

Voir aussi

- Utiliser la correspondance de modèle pour éviter la vérification « is » suivie d'un cast (IDE0020 et IDE0038)
- Exploration : Utiliser la correspondance de modèle pour générer votre comportement de classe pour un meilleur code
- Didacticiel : Utiliser les critères spéciaux pour générer des algorithmes pilotés par type et pilotés par les données
- Référence : Critères spéciaux

Éléments ignorés - Notions de base C#

Article • 07/04/2023

Les éléments ignorés sont des variables d'espace réservé, qui sont inutilisées de façon intentionnelle dans le code d'une application. Les éléments ignorés sont équivalents à des variables non affectées ; ils n'ont pas de valeur. Un élément ignoré communique l'intention au compilateur et à d'autres personnes qui lisent votre code : vous avez l'intention d'ignorer le résultat d'une expression. Vous pouvez ignorer le résultat d'une expression, un ou plusieurs membres d'une expression tuple, un paramètre `out` vers une méthode ou la cible d'une expression de critères spéciaux.

Les éléments ignorés permettent de clarifier l'objectif de votre code. Un élément ignoré indique que notre code n'utilise jamais la variable. Ils améliorent sa lisibilité et sa facilité de maintenance.

Vous indiquez qu'une variable est un élément ignoré en lui affectant comme nom le trait de soulignement (`_`). Par exemple, l'appel de méthode suivant retourne un tuple, où la première et la seconde valeurs sont des éléments ignorés. `area` est une variable précédemment déclarée définie sur le troisième composant retourné par `GetCityInformation`:

```
C#

(_, _, area) = city.GetCityInformation(cityName);
```

À compter de C# 9.0, vous pouvez utiliser les éléments ignorés pour spécifier les paramètres d'entrée inutilisés d'une expression lambda. Pour plus d'informations, consultez la section Paramètres d'entrée d'une expression lambda de l'article Expressions lambda.

Quand `_` constitue un discard valide, tout tentative de récupérer sa valeur ou de l'utiliser dans une opération d'affectation génère l'erreur de compilateur CS0103, « Le nom "_" n'existe pas dans le contexte actuel ». Cette erreur se produit parce qu'aucune valeur n'est affectée à `_` et qu'il n'est même pas possible de lui affecter un emplacement de stockage. S'il s'agissait d'une variable réelle, vous ne pourriez pas ignorer plus d'une valeur, comme l'a fait l'exemple précédent.

Déconstruction de tuple et d'objet

Les éléments ignorés sont utiles pour travailler avec des tuples quand le code de votre application utilise certains éléments d'un tuple, mais ignore les autres. Par exemple, la méthode `QueryCityDataForYears` suivante retourne un tuple avec le nom d'une ville, sa région, une année, la population de la ville pour cette année, une seconde année et la population de la ville pour cette seconde année. L'exemple montre la différence de population entre ces deux années. Parmi les données disponibles dans le tuple, nous ne sommes pas intéressés par la région de la ville, et nous connaissons le nom de la ville et les deux dates au moment du design. Par conséquent, nous sommes intéressés seulement par les deux valeurs de la population stockées dans le tuple et nous pouvons gérer ses valeurs restantes comme éléments ignorés.

C#

```
var (_, _, _, pop1, _, pop2) = QueryCityDataForYears("New York City", 1960,
2010);

Console.WriteLine($"Population change, 1960 to 2010: {pop2 - pop1:N0}");

static (string, double, int, int, int, int) QueryCityDataForYears(string
name, int year1, int year2)
{
    int population1 = 0, population2 = 0;
    double area = 0;

    if (name == "New York City")
    {
        area = 468.48;
        if (year1 == 1960)
        {
            population1 = 7781984;
        }
        if (year2 == 2010)
        {
            population2 = 8175133;
        }
        return (name, area, year1, population1, year2, population2);
    }

    return ("", 0, 0, 0, 0, 0);
}
// The example displays the following output:
//      Population change, 1960 to 2010: 393,149
```

Pour plus d'informations sur la déconstruction de tuples avec des éléments ignorés, consultez Déconstruction de tuples et d'autres types.

La méthode `Deconstruct` d'une classe, d'un struct ou d'une interface vous permet aussi de récupérer et de déconstruire un ensemble spécifique de données d'un objet. Vous

pouvez utiliser des éléments ignorés quand vous êtes intéressé seulement par un sous-ensemble des valeurs déconstruites. L'exemple suivant déconstruit un objet `Person` en quatre chaînes (le prénom, le nom, la ville et l'État), mais ignore le nom et l'État.

C#

```
using System;

namespace Discards
{
    public class Person
    {
        public string FirstName { get; set; }
        public string MiddleName { get; set; }
        public string LastName { get; set; }
        public string City { get; set; }
        public string State { get; set; }

        public Person(string fname, string mname, string lname,
                      string cityName, string stateName)
        {
            FirstName = fname;
            MiddleName = mname;
            LastName = lname;
            City = cityName;
            State = stateName;
        }

        // Return the first and last name.
        public void Deconstruct(out string fname, out string lname)
        {
            fname = FirstName;
            lname = LastName;
        }

        public void Deconstruct(out string fname, out string mname, out
string lname)
        {
            fname = FirstName;
            mname = MiddleName;
            lname = LastName;
        }

        public void Deconstruct(out string fname, out string lname,
                                out string city, out string state)
        {
            fname = FirstName;
            lname = LastName;
            city = City;
            state = State;
        }
    }
    class Example
```

```
    {
        public static void Main()
        {
            var p = new Person("John", "Quincy", "Adams", "Boston", "MA");

            // Deconstruct the person object.
            var (fName, _, city, _) = p;
            Console.WriteLine($"Hello {fName} of {city}!");
            // The example displays the following output:
            //      Hello John of Boston!
        }
    }
}
```

Pour plus d'informations sur la déconstruction de types définis par l'utilisateur avec des éléments ignorés, consultez Déconstruction de tuples et d'autres types.

Utilisation des critères spéciaux avec `switch`

Le *modèle d'élément ignoré* peut être utilisé dans des critères spéciaux avec l'expression de commutateur. Chaque expression, y compris `null`, correspond toujours au modèle d'élément ignoré.

L'exemple suivant définit une méthode `ProvidesFormatInfo` qui utilise une expression `switch` pour déterminer si un objet fournit une implémentation de IFormatProvider et teste si l'objet est `null`. Il utilise également le modèle d'élément ignoré pour gérer les objets non null de n'importe quel autre type.

C#

```
object?[] objects = { CultureInfo.CurrentCulture,
                      CultureInfo.CurrentCulture.DateTimeFormat,
                      CultureInfo.CurrentCulture.NumberFormat,
                      new ArgumentException(), null };
foreach (var obj in objects)
    ProvidesFormatInfo(obj);

static void ProvidesFormatInfo(object? obj) =>
    Console.WriteLine(obj switch
    {
        IFormatProvider fmt => $"{fmt.GetType()} object",
        null => "A null object reference: Its use could result in a
NullReferenceException",
        _ => "Some object type without format information"
    });
// The example displays the following output:
//    System.Globalization.CultureInfo object
//    System.Globalization.DateTimeFormatInfo object
//    System.Globalization.NumberFormatInfo object
```

```
//    Some object type without format information
//    A null object reference: Its use could result in a
NullReferenceException
```

Appels à des méthodes avec des paramètres `out`

Quand vous appelez la méthode `Deconstruct` pour déconstruire un type défini par l'utilisateur (une instance d'une classe, d'une structure ou d'une interface), vous pouvez ignorer les valeurs d'arguments `out` individuels. Vous pouvez cependant aussi ignorer la valeur d'arguments `out` lors de l'appel de n'importe quelle méthode avec un paramètre `out`.

L'exemple suivant appelle la méthode DateTime.TryParse (String, out DateTime) pour déterminer si la représentation sous forme de chaîne d'une date est valide dans la culture actuelle. Comme l'exemple concerne ici uniquement la validation de la chaîne de date et pas son analyse pour extraire la date, l'argument `out` de la méthode est un élément ignoré.

C#

```
string[] dateStrings = {"05/01/2018 14:57:32.8", "2018-05-01 14:57:32.8",
                        "2018-05-01T14:57:32.8375298-04:00", "5/01/2018",
                        "5/01/2018 14:57:32.80 -07:00",
                        "1 May 2018 2:57:32.8 PM", "16-05-2018 1:00:32 PM",
                        "Fri, 15 May 2018 20:10:57 GMT" };
foreach (string dateString in dateStrings)
{
    if (DateTime.TryParse(dateString, out _))
        Console.WriteLine($"'{dateString}': valid");
    else
        Console.WriteLine($"'{dateString}': invalid");
}
// The example displays output like the following:
//       '05/01/2018 14:57:32.8': valid
//       '2018-05-01 14:57:32.8': valid
//       '2018-05-01T14:57:32.8375298-04:00': valid
//       '5/01/2018': valid
//       '5/01/2018 14:57:32.80 -07:00': valid
//       '1 May 2018 2:57:32.8 PM': valid
//       '16-05-2018 1:00:32 PM': invalid
//       'Fri, 15 May 2018 20:10:57 GMT': invalid
```

Élément ignoré autonome

Vous pouvez utiliser un élément ignoré autonome pour indiquer une variable que vous choisissez d'ignorer. L'une des utilisations courantes consiste à utiliser une affectation pour s'assurer qu'un argument n'est pas null. Le code suivant utilise un élément ignoré pour forcer une affectation. Le côté droit de l'affectation utilise l'opérateur de fusion null pour lever un System.ArgumentNullException lorsque l'argument est `null`. Le code n'a pas besoin du résultat de l'affectation, il est donc ignoré. L'expression force une vérification null. L'élément ignoré clarifie votre intention : le résultat de l'affectation n'est ni nécessaire ni utilisé.

C#

```
public static void Method(string arg)
{
    _ = arg ?? throw new ArgumentNullException(paramName: nameof(arg),
message: "arg can't be null");

    // Do work with arg.
}
```

L'exemple suivant utilise un élément ignoré autonome pour ignorer l'objet Task retourné par une opération asynchrone. Attribuer la tâche a pour effet de supprimer l'exception que l'opération lève au moment où elle est sur le point de se terminer. Votre intention est claire : vous souhaitez ignorer le `Task` et ignorer toutes les erreurs générées à partir de cette opération asynchrone.

C#

```
private static async Task ExecuteAsyncMethods()
{
    Console.WriteLine("About to launch a task...");
    _ = Task.Run(() =>
    {
        var iterations = 0;
        for (int ctr = 0; ctr < int.MaxValue; ctr++)
            iterations++;
        Console.WriteLine("Completed looping operation...");
        throw new InvalidOperationException();
    });
    await Task.Delay(5000);
    Console.WriteLine("Exiting after 5 second delay");
}
// The example displays output like the following:
//       About to launch a task...
//       Completed looping operation...
//       Exiting after 5 second delay
```

Sans affecter la tâche à un élément ignoré, le code suivant génère un avertissement du compilateur :

```C#
private static async Task ExecuteAsyncMethods()
{
    Console.WriteLine("About to launch a task...");
    // CS4014: Because this call is not awaited, execution of the current
method continues before the call is completed.
    // Consider applying the 'await' operator to the result of the call.
    Task.Run(() =>
    {
        var iterations = 0;
        for (int ctr = 0; ctr < int.MaxValue; ctr++)
            iterations++;
        Console.WriteLine("Completed looping operation...");
        throw new InvalidOperationException();
    });
    await Task.Delay(5000);
    Console.WriteLine("Exiting after 5 second delay");
```

> ⓘ **Notes**
>
> Si vous exécutez l'un des deux échantillons précédents à l'aide d'un débogueur, le débogueur arrête le programme lorsque l'exception est levée. Sans débogueur joint, l'exception est ignorée en mode silencieux dans les deux cas.

`_` est aussi un identificateur valide. Quand il est utilisé en dehors d'un contexte pris en charge, `_` est traité non pas comme élément ignoré, mais comme variable valide. Si un identificateur nommé `_` est déjà dans l'étendue, l'utilisation de `_` comme élément ignoré autonome peut provoquer :

- Une modification accidentelle de la valeur de la variable `_` dans l'étendue en lui affectant la valeur de l'élément ignoré prévu. Par exemple :

  ```C#
  private static void ShowValue(int _)
  {
     byte[] arr = { 0, 0, 1, 2 };
     _ = BitConverter.ToInt32(arr, 0);
     Console.WriteLine(_);
  }
   // The example displays the following output:
   //       33619968
  ```

- Une erreur de compilateur pour violation de sécurité du type. Par exemple :

```
C#
```

```
private static bool RoundTrips(int _)
{
   string value = _.ToString();
   int newValue = 0;
   _ = Int32.TryParse(value, out newValue);
   return _ == newValue;
}
// The example displays the following compiler error:
//      error CS0029: Cannot implicitly convert type 'bool' to 'int'
```

- Erreur du compilateur CS0136 : « Impossible de déclarer une variable locale ou un paramètre nommé '_' dans cette portée, car ce nom est utilisé dans une portée locale englobante pour définir une variable locale ou un paramètre. » Par exemple :

```
C#
```

```
 public void DoSomething(int _)
{
 var _ = GetValue(); // Error: cannot declare local _ when one is
already in scope
}
// The example displays the following compiler error:
// error CS0136:
//       A local or parameter named '_' cannot be declared in this
scope
//       because that name is used in an enclosing local scope
//       to define a local or parameter
```

Voir aussi

- Supprimer la valeur d'expression inutile (règle de style IDE0058)
- Supprimer l'affectation de valeur inutile (règle de style IDE0059)
- Supprimer le paramètre inutilisé (règle de style IDE0060)
- Déconstruction de tuples et d'autres types
- is, opérateur
- switch, expression

Déconstruction de tuples et d'autres types

Article • 07/04/2023

Un tuple offre un moyen léger de récupérer plusieurs valeurs à partir d'un appel de méthode. Cependant, une fois que vous récupérez le tuple, vous devez gérer ses éléments individuels. Travailler sur une base élément par élément est assez fastidieux, comme le montre l'exemple suivant. La méthode `QueryCityData` retourne un tuple de trois éléments, et chacun de ses éléments est affecté à une variable dans une opération distincte.

C#

```
public class Example
{
    public static void Main()
    {
        var result = QueryCityData("New York City");

        var city = result.Item1;
        var pop = result.Item2;
        var size = result.Item3;

         // Do something with the data.
    }

    private static (string, int, double) QueryCityData(string name)
    {
        if (name == "New York City")
            return (name, 8175133, 468.48);

        return ("", 0, 0);
    }
}
```

La récupération de plusieurs valeurs de champs et de propriétés d'un objet peut également être assez fastidieuse : vous devez affecter une valeur de champ ou de propriété à une variable membre par membre.

Vous pouvez récupérer plusieurs éléments d'un tuple ou récupérer plusieurs valeurs de champ et de propriété, ainsi que des valeurs calculées, à partir d'un objet en une seule opération de *déconstruction*. Pour déconstruire un tuple, vous affectez ses éléments à des variables individuelles. Quand vous déconstruisez un objet, vous affectez des valeurs sélectionnées à des variables individuelles.

Tuples

Des fonctionnalités C# intégrées prennent en charge la déconstruction des tuples, ce qui vous permet de décomposer tous les éléments d'un tuple en une seule opération. La syntaxe générale de déconstruction d'un tuple est similaire à la syntaxe qui permet d'en définir un : vous placez les variables auxquelles chaque élément doit être affecté entre des parenthèses, dans la partie gauche d'une instruction d'affectation. Par exemple, l'instruction suivante affecte les éléments d'un tuple de quatre éléments à quatre variables distinctes :

```
C#

var (name, address, city, zip) = contact.GetAddressInfo();
```

Il existe trois façons de déconstruire un tuple :

- Vous pouvez déclarer explicitement le type de chaque champ entre des parenthèses. L'exemple suivant utilise cette approche pour déconstruire un tuple de trois éléments retourné par la méthode `QueryCityData`.

  ```
  C#

  public static void Main()
  {
      (string city, int population, double area) = QueryCityData("New
  York City");

      // Do something with the data.
  }
  ```

- Vous pouvez utiliser le mot clé `var` pour que C# infère le type de chaque variable. Vous placez le mot clé `var` en dehors des parenthèses. L'exemple suivant utilise l'inférence de type lors de la déconstruction du tuple de trois éléments retourné par la méthode `QueryCityData`.

  ```
  C#

  public static void Main()
  {
      var (city, population, area) = QueryCityData("New York City");

      // Do something with the data.
  }
  ```

Vous pouvez aussi utiliser le mot clé `var` individuellement avec tout ou partie des déclarations de variables à l'intérieur des parenthèses.

```
C#

public static void Main()
{
    (string city, var population, var area) = QueryCityData("New York
City");

    // Do something with the data.
}
```

Ceci est assez fastidieux et n'est pas recommandé.

- Enfin, vous pouvez déconstruire le tuple dans des variables qui ont déjà été déclarées.

```
C#

public static void Main()
{
    string city = "Raleigh";
    int population = 458880;
    double area = 144.8;

    (city, population, area) = QueryCityData("New York City");

    // Do something with the data.
}
```

- À partir de C# 10, vous pouvez combiner la déclaration et l'affectation de variables dans une déconstruction.

```
C#

public static void Main()
{
    string city = "Raleigh";
    int population = 458880;

    (city, population, double area) = QueryCityData("New York City");

    // Do something with the data.
}
```

Vous ne pouvez pas spécifier de type spécifique en dehors des parenthèses, même si tous les champs du tuple ont le même type. Cela génère l'erreur du compilateur CS8136,

« La déconstruction de 'var (...)' form interdit un type spécifique pour 'var'.».

Vous devez également affecter chaque élément du tuple à une variable. Si vous omettez des éléments, le compilateur génère l'erreur CS8132, « Impossible de déconstruire un tuple de « x » éléments en « y » variables ».

Éléments tuples avec des discards

Souvent, lors de la déconstruction d'un tuple, vous êtes intéressé seulement par les valeurs de certains éléments. Vous pouvez tirer parti de la prise en charge de C# des *discards*, qui sont des variables en écriture seule dont vous avez choisi d'ignorer les valeurs. Un discard est choisi par un caractère de soulignement (« _ ») dans une affectation. Vous pouvez ignorer autant de valeurs que vous le souhaitez ; pour représenter toutes les valeurs, utilisez l'élément ignoré unique, `_`.

L'exemple suivant illustre l'utilisation de tuples avec des éléments ignorés. La méthode `QueryCityDataForYears` retourne un tuple de six éléments avec le nom d'une ville, sa région, une année, la population de la ville pour cette année, une seconde année et la population de la ville pour cette seconde année. L'exemple montre la différence de population entre ces deux années. Parmi les données disponibles dans le tuple, nous ne sommes pas intéressés par la région de la ville, et nous connaissons le nom de la ville et les deux dates au moment du design. Par conséquent, nous sommes intéressés seulement par les deux valeurs de la population stockées dans le tuple et nous pouvons gérer ses valeurs restantes comme éléments ignorés.

C#

```
using System;

public class ExampleDiscard
{
    public static void Main()
    {
        var (_, _, _, pop1, _, pop2) = QueryCityDataForYears("New York
City", 1960, 2010);

        Console.WriteLine($"Population change, 1960 to 2010: {pop2 -
pop1:N0}");
    }

    private static (string, double, int, int, int, int)
QueryCityDataForYears(string name, int year1, int year2)
    {
        int population1 = 0, population2 = 0;
        double area = 0;
```

```
        if (name == "New York City")
        {
            area = 468.48;
            if (year1 == 1960)
            {
                population1 = 7781984;
            }
            if (year2 == 2010)
            {
                population2 = 8175133;
            }
            return (name, area, year1, population1, year2, population2);
        }

        return ("", 0, 0, 0, 0, 0);
    }
}
// The example displays the following output:
//      Population change, 1960 to 2010: 393,149
```

Types définis par l'utilisateur

C# n'offre pas de prise en charge intégrée pour la déconstruction de types non tuples autres que les types record et DictionaryEntry. Cependant, en tant que créateur d'une classe, d'un struct ou d'une interface, vous pouvez permettre la déconstruction du type en implémentant une ou plusieurs méthodes `Deconstruct`. La méthode retourne void, et chaque valeur à déconstruire est indiquée par un paramètre out dans la signature de la méthode. Par exemple, la méthode `Deconstruct` suivante d'une classe `Person` retourne le prénom, le deuxième prénom et le nom :

C#

```
public void Deconstruct(out string fname, out string mname, out string
lname)
```

Vous pouvez alors déconstruire une instance de la `Person`classe nommée `p` avec un code comme celui-ci :

C#

```
var (fName, mName, lName) = p;
```

L'exemple suivant surcharge la méthode `Deconstruct` de façon retourner différentes combinaisons des propriétés d'un objet `Person`. Les différentes surcharges retournent :

- Un prénom et un nom.
- Prénom, deuxième prénom et nom de famille.
- Un prénom, un nom, un nom de ville et un nom d'état.

C#

```
using System;

public class Person
{
    public string FirstName { get; set; }
    public string MiddleName { get; set; }
    public string LastName { get; set; }
    public string City { get; set; }
    public string State { get; set; }

    public Person(string fname, string mname, string lname,
                  string cityName, string stateName)
    {
        FirstName = fname;
        MiddleName = mname;
        LastName = lname;
        City = cityName;
        State = stateName;
    }

    // Return the first and last name.
    public void Deconstruct(out string fname, out string lname)
    {
        fname = FirstName;
        lname = LastName;
    }

    public void Deconstruct(out string fname, out string mname, out string
lname)
    {
        fname = FirstName;
        mname = MiddleName;
        lname = LastName;
    }

    public void Deconstruct(out string fname, out string lname,
                            out string city, out string state)
    {
        fname = FirstName;
        lname = LastName;
        city = City;
        state = State;
    }
}

public class ExampleClassDeconstruction
{
```

```
    public static void Main()
    {
        var p = new Person("John", "Quincy", "Adams", "Boston", "MA");

        // Deconstruct the person object.
        var (fName, lName, city, state) = p;
        Console.WriteLine($"Hello {fName} {lName} of {city}, {state}!");
    }
}
// The example displays the following output:
//    Hello John Adams of Boston, MA!
```

Plusieurs méthodes `Deconstruct` ayant le même nombre de paramètres sont ambiguës. Vous devez veiller à définir des méthodes `Deconstruct` avec différents nombres de paramètres, ou « arity ». Les méthodes `Deconstruct` avec le même nombre de paramètres ne peuvent pas être distinguées lors de la résolution de surcharge.

Type défini par l'utilisateur avec des discards

Tout comme vous le faites avec des tuples, vous pouvez utiliser des éléments ignorés pour ignorer des éléments sélectionnés retournés par une méthode `Deconstruct`. Chaque élément ignoré est défini par une variable nommée « _ », et une même opération de déconstruction peut inclure plusieurs éléments ignorés.

L'exemple suivant déconstruit un objet `Person` en quatre chaînes (le prénom et le nom, la ville et l'état), mais ignore le nom et l'état.

C#

```
// Deconstruct the person object.
var (fName, _, city, _) = p;
Console.WriteLine($"Hello {fName} of {city}!");
// The example displays the following output:
//      Hello John of Boston!
```

Méthodes d'extension pour les types définis par l'utilisateur

Si vous n'avez pas créé une classe, un struct ou une interface, vous pouvez néanmoins déconstruire des objets de ce type en implémentant une ou plusieurs `Deconstruct`méthodes d'extension pour retourner les valeurs qui vous intéressent.

L'exemple suivant définit deux méthodes d'extension `Deconstruct` pour la classe System.Reflection.PropertyInfo. La première retourne un ensemble de valeurs qui indique les caractéristiques de la propriété, notamment son type, si elle est statique ou d'instance, si elle est en lecture seule et si elle est indexée. La seconde indique l'accessibilité de la propriété. Comme l'accessibilité des accesseurs get et set peut varier, des valeurs booléennes indiquent si la propriété a des accesseurs get et set distincts et, le cas échéant, s'ils ont la même accessibilité. S'il n'existe qu'un seul accesseur, ou si les deux accesseurs get et set ont la même accessibilité, la variable `access` indique l'accessibilité de la propriété comme un tout. Sinon, l'accessibilité des accesseurs get et set est indiquée par les variables `getAccess` et `setAccess`.

C#

```
using System;
using System.Collections.Generic;
using System.Reflection;

public static class ReflectionExtensions
{
    public static void Deconstruct(this PropertyInfo p, out bool isStatic,
                                   out bool isReadOnly, out bool isIndexed,
                                   out Type propertyType)
    {
        var getter = p.GetMethod;

        // Is the property read-only?
        isReadOnly = ! p.CanWrite;

        // Is the property instance or static?
        isStatic = getter.IsStatic;

        // Is the property indexed?
        isIndexed = p.GetIndexParameters().Length > 0;

        // Get the property type.
        propertyType = p.PropertyType;
    }

    public static void Deconstruct(this PropertyInfo p, out bool
hasGetAndSet,
                                   out bool sameAccess, out string access,
                                   out string getAccess, out string
setAccess)
    {
        hasGetAndSet = sameAccess = false;
        string getAccessTemp = null;
        string setAccessTemp = null;

        MethodInfo getter = null;
        if (p.CanRead)
            getter = p.GetMethod;
```

```
        MethodInfo setter = null;
        if (p.CanWrite)
            setter = p.SetMethod;

        if (setter != null && getter != null)
            hasGetAndSet = true;

        if (getter != null)
        {
            if (getter.IsPublic)
                getAccessTemp = "public";
            else if (getter.IsPrivate)
                getAccessTemp = "private";
            else if (getter.IsAssembly)
                getAccessTemp = "internal";
            else if (getter.IsFamily)
                getAccessTemp = "protected";
            else if (getter.IsFamilyOrAssembly)
                getAccessTemp = "protected internal";
        }

        if (setter != null)
        {
            if (setter.IsPublic)
                setAccessTemp = "public";
            else if (setter.IsPrivate)
                setAccessTemp = "private";
            else if (setter.IsAssembly)
                setAccessTemp = "internal";
            else if (setter.IsFamily)
                setAccessTemp = "protected";
            else if (setter.IsFamilyOrAssembly)
                setAccessTemp = "protected internal";
        }

        // Are the accessibility of the getter and setter the same?
        if (setAccessTemp == getAccessTemp)
        {
            sameAccess = true;
            access = getAccessTemp;
            getAccess = setAccess = String.Empty;
        }
        else
        {
            access = null;
            getAccess = getAccessTemp;
            setAccess = setAccessTemp;
        }
    }
}

public class ExampleExtension
{
    public static void Main()
```

```
    {
        Type dateType = typeof(DateTime);
        PropertyInfo prop = dateType.GetProperty("Now");
        var (isStatic, isRO, isIndexed, propType) = prop;
        Console.WriteLine($"\nThe {dateType.FullName}.{prop.Name}
property:");
        Console.WriteLine($"   PropertyType: {propType.Name}");
        Console.WriteLine($"   Static:       {isStatic}");
        Console.WriteLine($"   Read-only:    {isRO}");
        Console.WriteLine($"   Indexed:      {isIndexed}");

        Type listType = typeof(List<>);
        prop = listType.GetProperty("Item",
                                    BindingFlags.Public |
BindingFlags.NonPublic | BindingFlags.Instance | BindingFlags.Static);
        var (hasGetAndSet, sameAccess, accessibility, getAccessibility,
setAccessibility) = prop;
        Console.Write($"\nAccessibility of the {listType.FullName}.
{prop.Name} property: ");

        if (!hasGetAndSet | sameAccess)
        {
            Console.WriteLine(accessibility);
        }
        else
        {
            Console.WriteLine($"\n   The get accessor: {getAccessibility}");
            Console.WriteLine($"   The set accessor: {setAccessibility}");
        }
    }
}
// The example displays the following output:
//       The System.DateTime.Now property:
//          PropertyType: DateTime
//          Static:       True
//          Read-only:    True
//          Indexed:      False
//
//       Accessibility of the System.Collections.Generic.List`1.Item
property: public
```

Méthode d’extension pour les types de système

Certains types de système fournissent la méthode `Deconstruct` par commodité. Par exemple, le type System.Collections.Generic.KeyValuePair<TKey,TValue> fournit cette fonctionnalité. Lorsque vous effectuez une itération surSystem.Collections.Generic.Dictionary<TKey,TValue>, chaque élément est un `KeyValuePair<TKey, TValue>` et peut être déconstruit. Prenons l’exemple suivant :

```
C#
```

```
Dictionary<string, int> snapshotCommitMap =
new(StringComparer.OrdinalIgnoreCase)
{
    ["https://github.com/dotnet/docs"] = 16_465,
    ["https://github.com/dotnet/runtime"] = 114_223,
    ["https://github.com/dotnet/installer"] = 22_436,
    ["https://github.com/dotnet/roslyn"] = 79_484,
    ["https://github.com/dotnet/aspnetcore"] = 48_386
};

foreach (var (repo, commitCount) in snapshotCommitMap)
{
    Console.WriteLine(
        $"The {repo} repository had {commitCount:N0} commits as of November
10th, 2021.");
}
```

Vous pouvez ajouter une méthode `Deconstruct` aux types de système qui n'en ont pas. Considérez la méthode d'extension suivante :

C#

```
public static class NullableExtensions
{
    public static void Deconstruct<T>(
        this T? nullable,
        out bool hasValue,
        out T value) where T : struct
    {
        hasValue = nullable.HasValue;
        value = nullable.GetValueOrDefault();
    }
}
```

Cette méthode d'extension permet à tous les types Nullable<T> d'être déconstruits dans un tuple `(bool hasValue, T value)`. L'exemple suivant montre le code qui utilise cette méthode d'extension :

C#

```
DateTime? questionableDateTime = default;
var (hasValue, value) = questionableDateTime;
Console.WriteLine(
    $"{{ HasValue = {hasValue}, Value = {value} }}");

questionableDateTime = DateTime.Now;
(hasValue, value) = questionableDateTime;
Console.WriteLine(
    $"{{ HasValue = {hasValue}, Value = {value} }}");
```

```
// Example outputs:
// { HasValue = False, Value = 1/1/0001 12:00:00 AM }
// { HasValue = True, Value = 11/10/2021 6:11:45 PM }
```

Types `record`

Lorsque vous déclarez un type d'enregistrement à l'aide de deux paramètres positionnels ou plus, le compilateur crée une méthode `Deconstruct` avec un paramètre `out` pour chaque paramètre positionnel dans la déclaration `record`. Pour plus d'informations, consultez Syntaxe positionnelle pour la définition de propriété et Comportement de déconstruction dans les enregistrements dérivés.

Voir aussi

- Déclaration de variable de déconstruction (règle de style IDE0042)
- Éléments ignorés
- Types de tuples

Exceptions et gestion des exceptions

Article • 23/02/2023

Les fonctionnalités de gestion des exceptions du langage C# vous aident à gérer les situations inattendues ou exceptionnelles qui se produisent lorsqu'un programme est en cours d'exécution. La gestion des exceptions utilise les mots clés `try`, `catch` et `finally` pour tenter des actions susceptibles de ne pas réussir, pour gérer les défaillances lorsque vous pensez que c'est justifié et pour nettoyer ensuite les ressources. Les exceptions peuvent être générées par le common language runtime (CLR), par .NET ou des bibliothèques tierces, ou par le code de l'application. Les exceptions sont créées avec le mot clé `throw`.

Dans de nombreux cas, une exception peut être levée non pas par une méthode appelée directement par votre code, mais par une autre méthode plus loin dans la pile des appels. Quand une exception est levée, le CLR déroule la pile à la recherche d'une méthode avec un bloc `catch` pour le type d'exception spécifique et exécute le premier bloc `catch` de ce type qu'il trouve. S'il ne trouve pas de bloc `catch` dans la pile des appels, il termine le processus et affiche un message à l'utilisateur.

Dans cet exemple, une méthode teste la division par zéro et intercepte l'erreur. Sans la gestion des exceptions, ce programme se terminerait avec une erreur **DivideByZeroException non gérée**.

C#

```
public class ExceptionTest
{
    static double SafeDivision(double x, double y)
    {
        if (y == 0)
            throw new DivideByZeroException();
        return x / y;
    }

    public static void Main()
    {
        // Input for test purposes. Change the values to see
        // exception handling behavior.
        double a = 98, b = 0;
        double result;

        try
        {
            result = SafeDivision(a, b);
            Console.WriteLine("{0} divided by {1} = {2}", a, b, result);
        }
```

```
        catch (DivideByZeroException)
        {
            Console.WriteLine("Attempted divide by zero.");
        }
    }
}
```

Vue d'ensemble des exceptions

Les exceptions ont les propriétés suivantes :

- Les exceptions sont des types qui dérivent tous en définitive de `System.Exception`.
- Utilisez un bloc `try` autour des instructions qui peuvent lever des exceptions.
- Dès qu'une exception se produit dans le bloc `try`, le flux de contrôle passe immédiatement au premier gestionnaire d'exceptions associé présent dans la pile des appels. En C#, le mot clé `catch` est utilisé pour définir un gestionnaire d'exceptions.
- Si aucun gestionnaire d'exceptions n'est présent pour une exception donnée, le programme s'arrête avec un message d'erreur.
- N'interceptez pas une exception à moins que vous puissiez la gérer tout en laissant l'application dans un état connu. Si vous interceptez `System.Exception`, levez-la de nouveau avec le mot clé `throw` à la fin du bloc `catch`.
- Si un bloc `catch` définit une variable d'exception, vous pouvez l'utiliser pour obtenir plus d'informations sur le type d'exception qui s'est produit.
- Les exceptions peuvent être générées explicitement par un programme avec le mot clé `throw`.
- Les objets Exception contiennent des informations détaillées sur l'erreur, telles que l'état de la pile des appels et une description du texte de l'erreur.
- Le code qui se trouve dans un bloc `finally` est exécuté même si une exception est levée. Utilisez un bloc `finally` pour libérer des ressources, par exemple pour fermer tous les flux ou fichiers qui ont été ouverts dans le bloc `try`.
- Les exceptions gérées dans .NET sont implémentées au-dessus du mécanisme de gestion structurée des exceptions de Win32. Pour plus d'informations, consultez les pages Gestion structurée des exceptions (C/C++) et Cours intensif sur la gestion structurée des exceptions de Win32 .

Spécification du langage C#

Pour plus d'informations, consultez Exceptions dans la Spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- SystemException
- Mots clés C#
- throw
- try-catch
- try-finally
- try-catch-finally
- Exceptions

Utiliser des exceptions

Article • 10/05/2023

En C#, les erreurs qui se produisent dans le programme au moment de l'exécution sont propagées à travers le programme à l'aide du mécanisme des exceptions. Les exceptions sont levées par le code qui rencontre une erreur et interceptées par le code qui peut corriger l'erreur. Les exceptions peuvent être levées par le runtime .NET ou par du code dans un programme. Une fois qu'une exception est levée, elle se propage jusqu'en haut de la pile des appels, jusqu'à ce qu'une instruction `catch` soit trouvée pour l'exception. Les exceptions non interceptées sont gérées par un gestionnaire d'exceptions génériques fourni par le système qui affiche une boîte de dialogue.

Les exceptions sont représentées par des classes dérivées de Exception. Cette classe identifie le type d'exception et contient des propriétés comportant des détails sur l'exception. Lever une exception implique de créer une instance d'une classe dérivée d'une exception, de configurer éventuellement les propriétés de l'exception, puis de lever l'objet à l'aide du mot clé `throw`. Par exemple :

C#

```
class CustomException : Exception
{
    public CustomException(string message)
    {
    }
}
private static void TestThrow()
{
    throw new CustomException("Custom exception in TestThrow()");
}
```

Après la levée d'une exception, le runtime vérifie l'instruction actuelle pour voir si elle se trouve dans un bloc `try`. Si tel est le cas, tous les blocs `catch` associés au bloc `try` font l'objet d'un contrôle permettant de déterminer s'ils peuvent intercepter l'exception. Les blocs `Catch` spécifient généralement des types d'exception ; si le type du bloc `catch` est identique à celui de l'exception ou d'une classe de base de l'exception, le bloc `catch` peut gérer la méthode. Par exemple :

C#

```
try
{
    TestThrow();
}
```

```
catch (CustomException ex)
{
    System.Console.WriteLine(ex.ToString());
}
```

Si l'instruction qui lève une exception ne se trouve pas à l'intérieur d'un bloc `try` ou si le bloc `try` qui la contient n'a pas de bloc `catch` correspondant, le runtime vérifie que la méthode d'appel dispose d'une instruction `try` et de blocs `catch`. Le runtime continue jusqu'à la pile appelante, à la recherche d'un bloc `catch` compatible. Une fois le bloc `catch` trouvé et exécuté, le contrôle est passé à l'instruction suivante après le bloc `catch`.

Une instruction `try` peut contenir plusieurs blocs `catch`. La première instruction `catch` qui peut gérer l'exception est exécutée ; toutes les instructions `catch` suivantes, même compatibles, sont ignorées. Ordonnez les blocs catch du plus spécifique (ou plus dérivé) au moins spécifique. Par exemple :

C#

```
using System;
using System.IO;

namespace Exceptions
{
    public class CatchOrder
    {
        public static void Main()
        {
            try
            {
                using (var sw = new StreamWriter("./test.txt"))
                {
                    sw.WriteLine("Hello");
                }
            }
            // Put the more specific exceptions first.
            catch (DirectoryNotFoundException ex)
            {
                Console.WriteLine(ex);
            }
            catch (FileNotFoundException ex)
            {
                Console.WriteLine(ex);
            }
            // Put the least specific exception last.
            catch (IOException ex)
            {
                Console.WriteLine(ex);
            }
```

```
            Console.WriteLine("Done");
        }
    }
}
```

Avant que le bloc `catch` ne soit exécuté, le runtime vérifie la présence de blocs `finally`. Les blocs `Finally` permettent au programmeur de nettoyer tout état ambigu qui pourrait subsister après l'abandon d'un bloc `try`, ou de libérer les ressources externes (telles que les handles graphiques, les connexions de bases de données ou les flux de fichiers) sans attendre que le garbage collector du runtime ne finalise les objets. Par exemple :

C#

```
static void TestFinally()
{
    FileStream? file = null;
    //Change the path to something that works on your machine.
    FileInfo fileInfo = new System.IO.FileInfo("./file.txt");

    try
    {
        file = fileInfo.OpenWrite();
        file.WriteByte(0xF);
    }
    finally
    {
        // Closing the file allows you to reopen it immediately - otherwise
IOException is thrown.
        file?.Close();
    }

    try
    {
        file = fileInfo.OpenWrite();
        Console.WriteLine("OpenWrite() succeeded");
    }
    catch (IOException)
    {
        Console.WriteLine("OpenWrite() failed");
    }
}
```

Si `WriteByte()` a levé une exception, le code du deuxième bloc `try` qui essaie de rouvrir le fichier échoue si `file.Close()` n'est pas appelé, et le fichier reste verrouillé. Étant donné que les blocs `finally` sont exécutés même si une exception est levée, le bloc `finally` de l'exemple précédent autorise la fermeture correcte du fichier et permet d'éviter une erreur.

Si aucun bloc `catch` compatible n'est trouvé sur la pile des appels après la levée d'une exception, l'une des trois situations suivantes se produit :

- Si l'exception se trouve dans un finaliseur, le finaliseur est abandonné et le finaliseur de base, le cas échéant, est appelé.
- Si la pile des appels contient un constructeur statique ou un initialiseur de champ statique, une TypeInitializationException est levée, avec l'exception d'origine assignée à la propriété InnerException de la nouvelle exception.
- Si le début du thread est atteint, le thread est terminé.

Gestion des exceptions (Guide de programmation C#)

Article • 21/03/2023

Un bloc try est utilisé par les programmeurs C# pour partitionner du code susceptible d'être affecté par une exception. Des blocs catch associés sont utilisés pour gérer les exceptions générées. Un bloc finally contient du code qui s'exécute dans tous les cas, qu'une exception soit levée ou non dans le bloc `try` (il peut s'agir par exemple de la libération des ressources allouées dans le bloc `try`). Un bloc `try` doit être associé à un ou plusieurs blocs `catch`, à un bloc `finally`, ou aux deux.

Les exemples suivants montrent une instruction `try-catch`, une instruction `try-finally` et une instruction `try-catch-finally`.

C#

```
try
{
    // Code to try goes here.
}
catch (SomeSpecificException ex)
{
    // Code to handle the exception goes here.
    // Only catch exceptions that you know how to handle.
    // Never catch base class System.Exception without
    // rethrowing it at the end of the catch block.
}
```

C#

```
try
{
    // Code to try goes here.
}
finally
{
    // Code to execute after the try block goes here.
}
```

C#

```
try
{
    // Code to try goes here.
}
```

```
catch (SomeSpecificException ex)
{
    // Code to handle the exception goes here.
}
finally
{
    // Code to execute after the try (and possibly catch) blocks
    // goes here.
}
```

Un bloc `try` sans bloc `catch` ou `finally` provoque une erreur du compilateur.

Blocs catch

Un bloc `catch` peut spécifier le type d'exception à intercepter. La spécification de type est appelée *filtre d'exception*. Le type d'exception doit être dérivé de Exception. En général, ne spécifiez pas Exception comme filtre d'exception sauf si vous savez comment gérer toutes les exceptions susceptibles d'être levées dans le bloc `try` ou si vous avez inclus une instruction throw à la fin de votre bloc `catch`.

Vous pouvez chaîner plusieurs blocs `catch` avec des classes d'exception différents. Les blocs `catch` sont évalués de haut en bas dans votre code, mais un seul bloc `catch` est exécuté pour chaque exception levée. Le premier bloc `catch` qui spécifie le type exact ou une classe de base de l'exception levée est exécuté. Si aucun bloc `catch` ne spécifie une classe d'exception correspondant, un bloc `catch` qui n'a pas de type est sélectionné, s'il y en a un dans l'instruction. Lors du positionnement des blocs `catch`, il est important de placer en premier les classes d'exception les plus spécifiques (c'est-à-dire les plus dérivés).

Interceptez les exceptions quand les conditions suivantes sont remplies :

- Vous savez pourquoi l'exception a été levée et vous pouvez implémenter une récupération spécifique, par exemple inviter l'utilisateur à entrer un nouveau nom de fichier quand vous interceptez un objet FileNotFoundException.
- Vous pouvez créer et lever une nouvelle exception plus spécifique.

  C#

  ```
  int GetInt(int[] array, int index)
  {
      try
      {
          return array[index];
      }
      catch (IndexOutOfRangeException e)
  ```

```
        {
            throw new ArgumentOutOfRangeException(
                "Parameter index is out of range.", e);
        }
    }
```

- Vous voulez traiter partiellement une exception avant de la transmettre en vue d'un traitement supplémentaire. Dans l'exemple suivant, un bloc `catch` est utilisé pour ajouter une entrée à un journal d'erreurs avant de relever l'exception.

```
C#

try
{
    // Try to access a resource.
}
catch (UnauthorizedAccessException e)
{
    // Call a custom error logging procedure.
    LogError(e);
    // Re-throw the error.
    throw;
}
```

Vous pouvez également indiquer des *filtres* d'exception pour ajouter une expression booléenne à une clause catch. Les filtres d'exception indiquent qu'une clause catch spécifique ne correspond que lorsque cette condition est vraie. Dans l'exemple suivant, les deux clauses catch utilisent la même classe d'exception, mais une condition supplémentaire est vérifiée pour créer un message d'erreur différent :

```
C#

int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e) when (index < 0)
    {
        throw new ArgumentOutOfRangeException(
            "Parameter index cannot be negative.", e);
    }
    catch (IndexOutOfRangeException e)
    {
        throw new ArgumentOutOfRangeException(
            "Parameter index cannot be greater than the array size.", e);
    }
}
```

Un filtre d'exception qui retourne toujours `false` peut être utilisé pour examiner toutes les exceptions sans les traiter. Une pratique courante consiste à enregistrer les exceptions :

C#

```
public static void Main()
{
    try
    {
        string? s = null;
        Console.WriteLine(s.Length);
    }
    catch (Exception e) when (LogException(e))
    {
    }
    Console.WriteLine("Exception must have been handled");
}

private static bool LogException(Exception e)
{
    Console.WriteLine($"\tIn the log routine. Caught {e.GetType()}");
    Console.WriteLine($"\tMessage: {e.Message}");
    return false;
}
```

La méthode `LogException` retourne toujours `false`, aucune clause `catch` utilisant ce filtre d'exception ne correspond. La clause catch peut être générale et utiliser System.Exception. Les clauses ultérieures peuvent traiter des classes d'exception plus spécifiques.

Blocs Finally

Un bloc `finally` vous permet de nettoyer les actions qui sont exécutées dans un bloc `try`. S'il est présent, le bloc `finally` s'exécute en dernier, après le bloc `try` et tout bloc `catch` mis en correspondance. Un bloc `finally` s'exécute toujours, qu'une exception soit levée ou non ou même si aucun bloc `catch` correspondant au type d'exception n'est trouvé.

Le bloc `finally` peut être utilisé pour libérer des ressources telles que des flux de fichiers, des connexions de base de données et des handles graphiques, sans attendre que le récupérateur de mémoire dans le runtime finalise les objets.

Dans l'exemple suivant, le bloc `finally` est utilisé pour fermer un fichier ouvert dans le bloc `try`. Notez que l'état du handle de fichier est vérifié avant la fermeture du fichier. Si

le bloc `try` ne peut pas ouvrir le fichier, le handle de fichier a encore la valeur `null` et le bloc `finally` n'essaie pas de le fermer. Au lieu de cela, si le fichier est ouvert avec succès dans le bloc `try`, le bloc `finally` ferme le fichier ouvert.

C#

```
FileStream? file = null;
FileInfo fileinfo = new System.IO.FileInfo("./file.txt");
try
{
    file = fileinfo.OpenWrite();
    file.WriteByte(0xF);
}
finally
{
    // Check for null because OpenWrite might have failed.
    file?.Close();
}
```

Spécification du langage C#

Pour plus d'informations, consultez Exceptions et Instruction try dans la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- try-catch
- try-finally
- try-catch-finally
- using, instruction

Création et levée d'exceptions

Article • 07/04/2023

Les exceptions sont utilisées pour indiquer qu'une erreur s'est produite pendant l'exécution du programme. Les objets d'exception qui décrivent une erreur sont créés, puis *levés* avec le mot clé throw. Le runtime recherche ensuite le gestionnaire d'exceptions le plus compatible.

Les programmeurs doivent lever des exceptions quand une ou plusieurs des conditions suivantes sont vérifiées :

- La méthode ne peut pas remplir sa fonction définie. Par exemple, si un paramètre d'une méthode a une valeur non valide :

  C#

  ```
  static void CopyObject(SampleClass original)
  {
      _ = original ?? throw new ArgumentException("Parameter cannot be
  null", nameof(original));
  }
  ```

- Un appel inapproprié à un objet est effectué en fonction de l'état de l'objet. Une tentative d'écriture dans un fichier en lecture seule en est un exemple. Dans les cas où l'état d'un objet n'autorise pas une opération, levez une instance d'InvalidOperationException ou un objet basé sur une dérivation de cette classe. Voici un exemple d'une méthode qui lève un objet InvalidOperationException :

  C#

  ```
  public class ProgramLog
  {
      FileStream logFile = null!;
      public void OpenLog(FileInfo fileName, FileMode mode) { }

      public void WriteLog()
      {
          if (!logFile.CanWrite)
          {
              throw new InvalidOperationException("Logfile cannot be
  read-only");
          }
          // Else write data to the log and return.
      }
  }
  ```

- Quand un argument d'une méthode provoque une exception. Dans ce cas, l'exception d'origine doit être interceptée et une instance d'ArgumentException doit être créée. L'exception d'origine doit être passée au constructeur d'ArgumentException comme paramètre InnerException :

  C#

  ```
  static int GetValueFromArray(int[] array, int index)
  {
      try
      {
          return array[index];
      }
      catch (IndexOutOfRangeException e)
      {
          throw new ArgumentOutOfRangeException(
              "Parameter index is out of range.", e);
      }
  }
  ```

> ⓘ Notes
>
> L'exemple ci-dessus est utilisé à des fins d'illustration. La validation d'index via des exceptions est dans la plupart des cas une mauvaise pratique. Les exceptions doivent être réservées pour se prémunir contre les conditions exceptionnelles du programme, et non pour la vérification des arguments comme ci-dessus.

Les exceptions contiennent une propriété nommée StackTrace. Cette chaîne contient le nom des méthodes sur la pile des appels actuelle, avec le nom de fichier et le numéro de ligne où l'exception a été levée pour chaque méthode. Un objet StackTrace est créé automatiquement par le CLR (Common Language Runtime) à partir du point de l'instruction `throw`, de sorte que les exceptions doivent être levées à partir du point où la trace de la pile doit commencer.

Toutes les exceptions contiennent une propriété nommée Message. Cette chaîne doit être définie pour expliquer la raison de l'exception. Les informations sensibles du point de vue de la sécurité ne doivent pas être placées dans le texte du message. Outre Message, ArgumentException contient une propriété nommée ParamName dont la valeur doit être le nom de l'argument qui a provoqué la levée de l'exception. Dans le cas d'une méthode setter de propriété, ParamName doit être défini en tant que `value`.

Les méthodes public et protected doivent lever des exceptions chaque fois qu'elles ne parviennent pas à remplir leurs fonctions habituelles. La classe d'exception levée est la plus spécifique disponible qui répond aux conditions d'erreur. Ces exceptions doivent

être documentées dans le cadre de la fonctionnalité de la classe. De plus, les classes dérivées ou les mises à jour de la classe d'origine doivent conserver le même comportement afin d'assurer la compatibilité descendante.

Pratiques à éviter lors de la levée d'exceptions

La liste suivante identifie les pratiques à éviter lors de la levée d'exceptions :

- N'utilisez pas d'exceptions pour changer le flux d'un programme dans le cadre d'une exécution ordinaire, mais utilisez-les plutôt pour signaler et gérer les conditions d'erreur.
- Les exceptions ne doivent pas être retournées comme valeur de retour ou paramètre au lieu d'être levées.
- Ne levez pas intentionnellement System.Exception, System.SystemException, System.NullReferenceException ni System.IndexOutOfRangeException à partir de votre propre code source.
- Ne créez pas d'exceptions qui peuvent être levées en mode débogage, mais pas en mode mise en production. Pour identifier des erreurs d'exécution pendant la phase de développement, utilisez plutôt Debug Assert.

Définition de classes d'exceptions

Les programmes peuvent lever une classe d'exceptions prédéfinie dans l'espace de noms System (sauf dans les endroits préalablement signalés) ou créer leurs propres classes d'exceptions en les dérivant d'Exception. Les classes dérivées doivent définir au moins trois constructeurs : un constructeur sans paramètre, un qui définit la propriété du message et un qui définit à la fois la propriété Message et la propriété InnerException. Par exemple :

C#

```
[Serializable]
public class InvalidDepartmentException : Exception
{
    public InvalidDepartmentException() : base() { }
    public InvalidDepartmentException(string message) : base(message) { }
    public InvalidDepartmentException(string message, Exception inner) :
base(message, inner) { }
}
```

Ajoutez de nouvelles propriétés à la classe d'exception lorsque les données qu'elles fournissent sont utiles pour résoudre l'exception. Si de nouvelles propriétés sont

ajoutées à la classe d'exceptions dérivée, la méthode `ToString()` doit être substituée pour retourner les informations ajoutées.

Spécification du langage C#

Pour plus d'informations, consultez Exceptions et Instruction throw dans la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Hiérarchie des exceptions

Exceptions générées par le compilateur

Article • 03/06/2023

Certaines exceptions sont levées automatiquement par le runtime .NET lors de l'échec des opérations de base. Ces exceptions et leurs conditions d'erreur sont répertoriées dans le tableau suivant.

Exception	Description
ArithmeticException	Classe de base pour les exceptions qui se produisent pendant des opérations arithmétiques, telles que DivideByZeroException et OverflowException.
ArrayTypeMismatchException	Levée quand un tableau ne peut pas stocker un élément donné, car le type réel de l'élément est incompatible avec le type réel du tableau.
DivideByZeroException	Levée lors d'une tentative de division d'une valeur intégrale par zéro.
IndexOutOfRangeException	Levée lors d'une tentative d'indexation d'un tableau à l'aide d'un index qui est inférieur à zéro ou en dehors des limites du tableau.
InvalidCastException	Levée quand une conversion explicite d'un type de base en interface ou en un type dérivé échoue au temps de l'exécution.
NullReferenceException	Levée quand vous essayez de référencer un objet dont la valeur est nul.
OutOfMemoryException	Levée quand une tentative d'allocation de mémoire à l'aide de l'opérateur new échoue. Cette exception indique que la mémoire disponible pour le Commun Language Runtime est épuisée.
OverflowException	Levée quand une opération dans un contexte `checked` engendre un dépassement.
StackOverflowException	Levée quand la pile d'exécution est épuisée par un trop grand nombre d'appels de méthode en attente ; cela indique généralement une récurrence très profonde ou infinie.
TypeInitializationException	Levée quand un constructeur statique lève une exception et qu'il n'existe aucune clause `catch` pour l'intercepter.

Voir aussi

- Instructions de gestion des exceptions

Règles et conventions de nommage des identificateurs C#

Article • 17/08/2023

Un **identificateur** est le nom que vous assignez à un type (classe, interface, struct, délégué ou enum), un membre, une variable ou un espace de noms.

Règles de nommage

Les identificateurs valides doivent suivre les règles suivantes :

- Les identificateurs doivent commencer par une lettre ou un trait de soulignement (`_`).
- Les identificateurs peuvent contenir des lettres Unicode, des nombres décimaux, des caractères de connexion Unicode, des caractères de combinaison Unicode ou des caractères de mise en forme Unicode. Pour plus d'informations sur les catégories Unicode, consultez la Base de données des catégories Unicode.

Vous pouvez déclarer des identificateurs qui correspondent à des mots clés C# en utilisant le préfixe `@` sur l'identificateur. `@` ne fait pas partie du nom de l'identificateur. Par exemple, `@if` déclare un identificateur nommé `if`. Ces identificateurs textuels sont destinés principalement à assurer l'interopérabilité avec les identificateurs déclarés dans d'autres langages.

Pour une définition complète des identificateurs valides, consultez la rubrique Identificateurs dans la spécification du langage C#.

Conventions d'attribution d'un nom

En plus des règles, il existe de nombreuses conventions de nommage d'identificateur utilisées dans les API .NET. Par convention, les programmes C# utilisent `PascalCase` pour les noms de types, les espaces de noms et tous les membres publics. En outre, l'équipe `dotnet/docs` utilise les conventions suivantes, adoptées à partir du style de codage de l'équipe .NET Runtime :

- Les noms d'interfaces commencent par un `I` majuscule.
- Les types d'attribut finissent par le mot `Attribute`.
- Les types enum utilisent un nom singulier pour les non-indicateurs et un nom pluriel pour les indicateurs.

- Les identificateurs ne doivent pas contenir deux traits de soulignement consécutifs (`_`). Ces noms sont réservés aux identificateurs générés par le compilateur.
- Utilisez des noms utiles et descriptifs pour les variables, les méthodes et les classes.
- Évitez d'utiliser des noms à lettres uniques, sauf pour les compteurs à boucles simples. Consultez les exceptions pour obtenir des exemples de syntaxe indiqués dans la section suivante.
- Préférez la clarté à la concision.
- Utilisez la casse Pascal pour les noms de classes et de méthode.
- Utilisez la casse camel pour les arguments de méthode, les variables locales et les champs privés.
- Utilisez la casse Pascal pour les noms de constantes, à la fois pour les champs et les constantes locales.
- Les champs d'instances privées commencent par un trait de soulignement (`_`).
- Les champs statiques commencent par `s_`. Notez qu'il ne s'agit pas du comportement par défaut de Visual Studio, ni d'une partie des instructions de conception d'infrastructure, mais qu'il est configurable dans editorconfig.
- Évitez d'utiliser des abréviations ou des acronymes dans des noms, à l'exception d'abréviations bien connues et acceptées.
- Utilisez des espaces de noms explicites et descriptifs qui respectent la notation de nom de domaine inverse.
- Choisissez des noms d'assembly qui représentent l'objectif principal de l'assembly.

Les exemples qui décrivent la syntaxe des constructions C# utilisent souvent des noms à lettres uniques qui correspondent à la convention utilisée dans la spécification du langage C# :

- Utilisez `S` pour les structs et `C` pour les classes.
- Utilisez `M` pour les méthodes.
- Utilisez `v` pour les variables et `p` pour les paramètres.
- Utilisez `r` pour les paramètres `ref`.

Les noms à lettres uniques précédents sont autorisés uniquement dans la section de référence de langage.

Dans les exemples suivants, l'aide relative aux éléments marqués `public` s'applique également lors de l'utilisation d'éléments `protected` et `protected internal`, qui sont tous destinés à être visibles par des appelants externes.

Casse Pascal

Utilisez la casse Pascal (« PascalCasing ») lorsque vous nommez un type `class`, `Interface`, `struct` ou `delegate`.

C#

```
public class DataService
{
}
```

C#

```
public record PhysicalAddress(
    string Street,
    string City,
    string StateOrProvince,
    string ZipCode);
```

C#

```
public struct ValueCoordinate
{
}
```

C#

```
public delegate void DelegateType(string message);
```

Lorsque vous nommez un `interface`, utilisez la casse Pascal en plus d'ajouter le préfixe `I` au nom. Ce préfixe indique clairement aux consommateurs qu'il s'agit d'une `interface`.

C#

```
public interface IWorkerQueue
{
}
```

Lorsque vous nommez des membres `public` de types, tels que des champs, des propriétés et des événements, utilisez la casse Pascal. Utilisez également la casse Pascal pour toute méthode et fonction locale.

C#

```
public class ExampleEvents
{
    // A public field, these should be used sparingly
    public bool IsValid;

    // An init-only property
    public IWorkerQueue WorkerQueue { get; init; }

    // An event
    public event Action EventProcessing;

    // Method
    public void StartEventProcessing()
    {
        // Local function
        static int CountQueueItems() => WorkerQueue.Count;
        // ...
    }
}
```

Lors de l'écriture d'enregistrements positionnels, utilisez la casse Pascal pour les paramètres, car ils sont les propriétés publiques de l'enregistrement.

C#

```
public record PhysicalAddress(
    string Street,
    string City,
    string StateOrProvince,
    string ZipCode);
```

Pour plus d'informations sur les enregistrements positionnels, consultez Syntaxe positionnelle pour la définition de propriété.

Casse mixte

Utilisez la casse camel (« camelCasing ») lors du nommage des champs `private` ou `internal` et ajoutez-leur un préfixe avec `_`. Utilisez la casse camel lors du nommage de variables locales, y compris les instances d'un type délégué.

C#

```
public class DataService
{
    private IWorkerQueue _workerQueue;
}
```

> **Conseil**
>
> Lors de la modification de code C# qui suit ces conventions d'affectation de noms dans un IDE qui prend en charge la saisie semi-automatique des instructions, la saisie de `_` affiche tous les membres délimités à l'objet.

Lorsque vous utilisez des champs `static` qui sont `private` ou `internal`, utilisez le préfixe `s_` et pour le thread static, utilisez `t_`.

```
C#

public class DataService
{
    private static IWorkerQueue s_workerQueue;

    [ThreadStatic]
    private static TimeSpan t_timeSpan;
}
```

Lors de l'écriture de paramètres de méthode, utilisez la casse mixte.

```
C#

public T SomeMethod<T>(int someNumber, bool isValid)
{
}
```

Pour plus d'informations sur les conventions d'affectation de noms C#, consultez C# Coding Style.

Recommandations concernant le nom des paramètres de type

Les instructions suivantes s'appliquent aux paramètres de type sur les paramètres de type générique. Il s'agit des espaces réservés pour les arguments dans un type générique ou dans une méthode générique. Vous pouvez en savoir plus sur les paramètres de type générique dans le guide de programmation C#.

- **Nommez** les paramètres de type générique avec des noms descriptifs, sauf si un nom composé d'une seule lettre est suffisamment évocateur et qu'un nom descriptif n'apporte rien de plus.

  ```
  ./snippets/coding-conventions
  ```

```
public interface ISessionChannel<TSession> { /*...*/ }
public delegate TOutput Converter<TInput, TOutput>(TInput from);
public class List<T> { /*...*/ }
```

- **Envisagez** d'utiliser `T` comme nom de paramètre de type pour les types ayant un paramètre de type avec une seule lettre.

./snippets/coding-conventions

```
public int IComparer<T>() { return 0; }
public delegate bool Predicate<T>(T item);
public struct Nullable<T> where T : struct { /*...*/ }
```

- **Préfixez** les noms des paramètres de type descriptifs avec « T ».

./snippets/coding-conventions

```
public interface ISessionChannel<TSession>
{
    TSession Session { get; }
}
```

- **Pensez** à indiquer les contraintes placées sur un paramètre de type dans le nom de paramètre. Par exemple, un paramètre limité à `ISession` peut être appelé `TSession`.

La règle d'analyse du code CA1715 vérifie que les paramètres de type sont correctement nommés.

Conventions supplémentaires d'affectation de noms

- Dans les exemples qui ne comprennent pas de directives, utilisez les qualifications d'espace de noms. Si vous savez qu'un espace de noms est importé par défaut dans un projet, vous n'êtes pas obligé de qualifier entièrement les noms à partir de cet espace de noms. Les noms qualifiés peuvent être interrompus après un point (.) s'ils sont trop longs pour tenir sur une seule ligne, comme indiqué dans l'exemple suivant.

C#

```
var currentPerformanceCounterCategory = new System.Diagnostics.
    PerformanceCounterCategory();
```

- Vous n'avez pas à modifier les noms des objets qui ont été créés à l'aide des outils du concepteur Visual Studio pour les rendre conformes aux autres directives.

Conventions de code C# courantes

Article • 17/08/2023

Une norme de code est essentielle pour maintenir la lisibilité et la cohérence du code ainsi que la collaboration au sein d'une équipe de développement. Suivre les pratiques du secteur et les recommandations établies permet de s'assurer que le code est plus facile à comprendre, à gérer et à étendre. La plupart des projets appliquent un style cohérent à l'aide de conventions de code. Les projets dotnet/docs et dotnet/samples ne font pas exception. Dans cette série d'articles, vous allez découvrir nos conventions de codage et les outils que nous utilisons pour les appliquer. Vous pouvez utiliser nos conventions en l'état ou les modifier en fonction des besoins de votre équipe.

Nous avons choisi nos conventions en fonction des objectifs suivants :

1. *Correction* : nos exemples sont copiés et collés dans vos applications. Il s'agit de quelque chose d'attendu, et nous devons donc garantir la résilience et la correction du code, même après plusieurs modifications.
2. *Enseignement* : l'objectif de nos exemples est d'enseigner tout ce qui touche au codage .NET et C#. Pour cette raison, nous n'avons pas de restrictions concernant une quelconque fonctionnalité de langage ou API. Au lieu de cela, ces exemples montrent quand une fonctionnalité est un bon choix.
3. *Cohérence* : les lecteurs attendent une expérience cohérente avec notre contenu. Tous les exemples doivent être conformes au même style.
4. *Adoption* : nous mettons à jour de manière agressive nos exemples pour utiliser de nouvelles fonctionnalités de langage. Cette pratique permet de sensibiliser aux nouvelles fonctionnalités et les rend plus familières à tous les développeurs C#.

> **ⓘ Important**
>
> Ces recommandations sont utilisées par Microsoft pour développer les exemples et la documentation. Elles ont été adoptées à partir des recommandations de **.NET Runtime, Style de codage C#** et du **compilateur C# (roslyn)**. Nous avons choisi ces recommandations, car elles ont été testées au cours de plusieurs années de développement open source. Elles ont aidé les membres de la communauté à participer aux projets runtime et de compilateur. Elles servent d'exemples de conventions C# courantes, et non de liste faisant autorité (pour cela, voir **Règles de conception de .NET Framework**).
>
> Les objectifs d'*enseignement* et d'*adoption* sont les raisons pour lesquelles la convention de codage de la documentation diffère des conventions du runtime et du compilateur. Le runtime et le compilateur ont des métriques de performances

> strictes pour les chemins chauds. De nombreuses autres applications n'en ont pas. Notre objectif d'*enseignement* nous oblige à n'interdire aucune construction. Au lieu de cela, les exemples indiquent quand les constructions doivent être utilisées. Nous mettons à jour les exemples de manière plus agressive que la plupart des applications de production. Notre objectif d'*adoption* nous oblige à afficher le code que vous devez écrire aujourd'hui, même lorsque le code écrit l'année dernière n'a pas besoin de modifications.

Cet article explique nos recommandations. Celles-ci ont évolué au fil du temps et vous trouverez des exemples qui ne les suivent pas. Nous accueillons favorablement les demandes de tirage qui mettent ces exemples en conformité ou les signalements de problème qui attirent notre attention sur les exemples que nous devrions mettre à jour. Nos recommandations sont en open source et nous accueillons favorablement les demandes de tirage et signalements de problème. Si toutefois votre soumission viendrait à modifier ces recommandations, signalez d'abord un problème pour discussion. Nous vous invitons à utiliser nos recommandations ou à les adapter à vos besoins.

Outils et analyseurs

Les outils peuvent aider votre équipe à appliquer vos normes. Vous pouvez activer n'importe lequel des outils d'analyse du code pour appliquer les règles que vous préférez. Vous pouvez également créer un editorconfig pour que Visual Studio applique automatiquement vos recommandations relatives au style. Vous pouvez commencer par utiliser le référentiel dotnet/docs pour utiliser notre style comme point de départ.

Ces outils facilitent l'adoption de vos recommandations par votre équipe. Visual Studio applique les règles de tous les fichiers `.editorconfig` dans l'étendue pour mettre en forme votre code. Vous pouvez utiliser plusieurs ensembles de règles pour appliquer des normes à l'échelle de l'entreprise, des normes d'équipe et même des normes de projet granulaires.

Tous les outils d'analyse du code configurés génèrent des avertissements et des diagnostics en cas de violation de leurs règles. Vous configurez les règles que vous souhaitez appliquer à votre projet. Ensuite, chaque build CI avertit les développeurs lorsqu'ils enfreignent l'une des règles.

Directives du langage

Les sections suivantes décrivent les pratiques que l'équipe responsable de la documentation .NET suit pour préparer les exemples de code et autres. En général, suivez ces pratiques :

- Utilisez des fonctionnalités de langage et des versions C# modernes dans la mesure du possible.
- Évitez les constructions de langage obsolètes.
- Interceptez uniquement les exceptions qui peuvent être gérées correctement ; évitez d'intercepter les exceptions génériques.
- Utilisez des types d'exceptions spécifiques pour fournir des messages d'erreur significatifs.
- Utilisez des requêtes et des méthodes LINQ pour la manipulation de collection afin d'améliorer la lisibilité du code.
- Utilisez la programmation asynchrone avec async et await pour les opérations liées aux E/S.
- Faites attention aux interblocages et utilisez Task.ConfigureAwait le cas échéant.
- Utilisez les mots clés de langage pour les types de données au lieu des types de runtime. Par exemple, utilisez `string` plutôt que System.String ou `int` plutôt que System.Int32.
- Utilisez `int` plutôt que les types non signés. L'utilisation de `int` est commune en C# et il est plus facile d'interagir avec d'autres bibliothèques lorsque vous utilisez `int`. Les exceptions concernent la documentation spécifique aux types de données non signés.
- Utilisez `var` uniquement lorsqu'un lecteur peut déduire le type de l'expression. Les lecteurs affichent nos exemples sur la plateforme de documentation. Ils n'ont pas de messages lors du pointage ou d'info-bulles qui affichent le type de variables.
- Écrivez du code avec clarté et simplicité.
- Évitez les logiques de code trop complexes.

Des instructions plus spécifiques sont fournies ci-dessous.

Données de chaîne

- Utilisez une interpolation de chaîne pour concaténer les chaînes courtes, comme illustré dans le code suivant.

```
C#
```

```
string displayName = $"{nameList[n].LastName},
{nameList[n].FirstName}";
```

- Pour ajouter des chaînes dans des boucles, en particulier lorsque vous travaillez avec une quantité de texte importante, utilisez un objet System.Text.StringBuilder.

```
C#

var phrase =
"lalalalalalalalalalalalalalalalalalalalalalalalalalalalalala";
var manyPhrases = new StringBuilder();
for (var i = 0; i < 10000; i++)
{
    manyPhrases.Append(phrase);
}
//Console.WriteLine("tra" + manyPhrases);
```

Tableaux

- Utilisez la syntaxe concise lorsque vous initialisez des tableaux sur la ligne de déclaration. Dans l'exemple suivant, vous ne pouvez pas utiliser `var` à la place de `string[]`.

```
C#

string[] vowels1 = { "a", "e", "i", "o", "u" };
```

- Si vous utilisez l'instanciation explicite, vous pouvez utiliser `var`.

```
C#

var vowels2 = new string[] { "a", "e", "i", "o", "u" };
```

Délégués

- Utilisez Func<> et Action<> au lieu de définir des types délégués. Dans une classe, définissez la méthode déléguée.

```
C#

Action<string> actionExample1 = x => Console.WriteLine($"x is: {x}");

Action<string, string> actionExample2 = (x, y) =>
    Console.WriteLine($"x is: {x}, y is {y}");

Func<string, int> funcExample1 = x => Convert.ToInt32(x);
```

```
Func<int, int, int> funcExample2 = (x, y) => x + y;
```

- Appelez la méthode à l'aide de la signature définie par le délégué `Func<>` ou `Action<>`.

C#

```
actionExample1("string for x");

actionExample2("string for x", "string for y");

Console.WriteLine($"The value is {funcExample1("1")}");

Console.WriteLine($"The sum is {funcExample2(1, 2)}");
```

- Pour créer des instances d'un type délégué, utilisez la syntaxe concise. Dans une classe, définissez le type délégué et une méthode qui a une signature correspondante.

  C#

  ```
  public delegate void Del(string message);

  public static void DelMethod(string str)
  {
      Console.WriteLine("DelMethod argument: {0}", str);
  }
  ```

- Créez une instance du type délégué et appelez-la. La déclaration suivante montre la syntaxe condensée.

  C#

  ```
  Del exampleDel2 = DelMethod;
  exampleDel2("Hey");
  ```

- La déclaration suivante utilise la syntaxe complète.

  C#

  ```
  Del exampleDel1 = new Del(DelMethod);
  exampleDel1("Hey");
  ```

Instructions `try-catch` et `using` dans la gestion des exceptions

- Utilisez une instruction try-catch pour la plus grande part de la gestion des exceptions.

  C#

  ```
  static double ComputeDistance(double x1, double y1, double x2, double
  y2)
  {
      try
      {
          return Math.Sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 -
  y2));
      }
      catch (System.ArithmeticException ex)
      {
          Console.WriteLine($"Arithmetic overflow or underflow: {ex}");
          throw;
      }
  }
  ```

- Simplifiez votre code à l'aide de l'instruction using C#. Si vous avez une instruction try-finally dans laquelle le seul code du bloc `finally` est un appel à la méthode Dispose, utilisez à la place une instruction `using`.

  Dans l'exemple suivant, l'instruction `try-finally` appelle uniquement `Dispose` dans le bloc `finally`.

  C#

  ```
  Font bodyStyle = new Font("Arial", 10.0f);
  try
  {
      byte charset = bodyStyle.GdiCharSet;
  }
  finally
  {
      if (bodyStyle != null)
      {
          ((IDisposable)bodyStyle).Dispose();
      }
  }
  ```

  Vous pouvez faire la même chose avec une instruction `using`.

  C#

```
using (Font arial = new Font("Arial", 10.0f))
{
    byte charset2 = arial.GdiCharSet;
}
```

Utilisez la nouvelle usingsyntaxe qui ne nécessite pas d'accolades :

C#

```
using Font normalStyle = new Font("Arial", 10.0f);
byte charset3 = normalStyle.GdiCharSet;
```

Opérateurs `&&` et `||`

- Utilisez && plutôt que & et || plutôt que | lorsque vous effectuez des comparaisons, comme illustré dans l'exemple suivant.

C#

```
Console.Write("Enter a dividend: ");
int dividend = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter a divisor: ");
int divisor = Convert.ToInt32(Console.ReadLine());

if ((divisor != 0) && (dividend / divisor > 0))
{
    Console.WriteLine("Quotient: {0}", dividend / divisor);
}
else
{
    Console.WriteLine("Attempted division by 0 ends up here.");
}
```

Si le diviseur est 0, la deuxième clause de l'instruction `if` provoque une erreur d'exécution. Mais l'opérateur && court-circuite lorsque la première expression est fausse. Autrement dit, elle n'évalue pas la deuxième expression. L'opérateur & évalue les deux, ce qui entraîne une erreur d'exécution quand `divisor` est 0.

L'opérateur `new`

- Utilisez l'une des formes concises d'instanciation d'objet, comme indiqué dans les déclarations suivantes. Le deuxième exemple montre la syntaxe disponible à partir de C# 9.

```
C#

var firstExample = new ExampleClass();
```

```
C#

ExampleClass instance2 = new();
```

Les déclarations précédentes sont équivalentes à la déclaration suivante.

```
C#

ExampleClass secondExample = new ExampleClass();
```

- Utilisez des initialiseurs d'objets pour simplifier la création d'objets, comme illustré dans l'exemple suivant.

```
C#

var thirdExample = new ExampleClass { Name = "Desktop", ID = 37414,
    Location = "Redmond", Age = 2.3 };
```

L'exemple suivant définit les mêmes propriétés que l'exemple précédent, mais il n'utilise pas d'initialiseurs.

```
C#

var fourthExample = new ExampleClass();
fourthExample.Name = "Desktop";
fourthExample.ID = 37414;
fourthExample.Location = "Redmond";
fourthExample.Age = 2.3;
```

Gestion des événements

- Utilisez une expression lambda pour définir un gestionnaire d'événements que vous n'aurez pas besoin de supprimer ultérieurement :

```
C#

public Form2()
{
    this.Click += (s, e) =>
        {
```

```
            MessageBox.Show(
                ((MouseEventArgs)e).Location.ToString());
        };
}
```

L'expression lambda raccourcit la définition traditionnelle suivante.

C#

```
public Form1()
{
    this.Click += new EventHandler(Form1_Click);
}

void Form1_Click(object? sender, EventArgs e)
{
    MessageBox.Show(((MouseEventArgs)e).Location.ToString());
}
```

Membres static

Appelez les membres static en utilisant le nom de la classe : *Nom_classe.Membre_statique*. Cette pratique rend le code plus lisible en clarifiant l'accès aux membres static. Ne qualifiez pas un membre statique défini dans une classe de base avec le nom d'une classe dérivée. Lorsque ce code est compilé, la lisibilité du code est trompeuse et le code peut s'interrompre à l'avenir si vous ajoutez à la classe dérivée un membre static de même nom.

Requêtes LINQ

- Utilisez des noms explicites pour les variables de requête. L'exemple suivant utilise `seattleCustomers` pour les clients qui se trouvent à Seattle.

  C#

  ```
  var seattleCustomers = from customer in customers
                         where customer.City == "Seattle"
                         select customer.Name;
  ```

- Utilisez des alias pour vous assurer que les noms de propriétés des types anonymes sont correctement écrits en majuscules à l'aide de la casse Pascal.

  C#

```
var localDistributors =
    from customer in customers
    join distributor in distributors on customer.City equals
distributor.City
    select new { Customer = customer, Distributor = distributor };
```

- Renommez les propriétés lorsque les noms de propriétés dans le résultat sont ambigus. Par exemple, si votre requête retourne un nom de client et un ID de distributeur, au lieu de les laisser sous la forme `Name` et `ID` dans le résultat, renommez-les pour montrer clairement que `Name` est le nom d'un client et `ID` l'ID d'un distributeur.

```
C#

var localDistributors2 =
    from customer in customers
    join distributor in distributors on customer.City equals
distributor.City
    select new { CustomerName = customer.Name, DistributorID =
distributor.ID };
```

- Utilisez le typage implicite dans la déclaration des variables de requête et des variables de portée. Cette aide sur le typage implicite dans les requêtes LINQ remplace les règles générales pour les variables locales implicitement typées. Les requêtes LINQ utilisent souvent des projections qui créent des types anonymes. D'autres expressions de requête créent des résultats avec des types génériques imbriqués. Les variables implicitement typées sont souvent plus lisibles.

```
C#

var seattleCustomers = from customer in customers
                       where customer.City == "Seattle"
                       select customer.Name;
```

- Alignez les clauses de requête sous la clause from, comme illustré dans les exemples précédents.

- Utilisez les clauses where avant les autres clauses de requête pour garantir que les clauses de requête ultérieures opèrent sur l'ensemble de données réduit et filtré.

```
C#

var seattleCustomers2 = from customer in customers
                        where customer.City == "Seattle"
```

```
                          orderby customer.Name
                          select customer;
```

- Utilisez plusieurs clauses `from` au lieu d'une clause join pour accéder aux collections internes. Par exemple, une collection d'objets `Student` peut contenir chacune une collection de scores de test. Lorsque la requête suivante est exécutée, elle retourne chaque score supérieur à 90, avec le nom de l'étudiant correspondant.

```
C#

var scoreQuery = from student in students
                 from score in student.Scores!
                 where score > 90
                 select new { Last = student.LastName, score };
```

Variables locales implicitement typées

- Utilisez le typage implicite pour les variables locales quand le type de la variable est évident à droite de l'assignation.

```
C#

var message = "This is clearly a string.";
var currentTemperature = 27;
```

- N'utilisez pas var quand le type n'est pas apparent à droite de l'assignation. Ne supposez pas que le type est clair à partir du nom de méthode. Un type de variable est considéré comme clair s'il s'agit d'un opérateur `new`, d'un cast explicite ou d'une assignation à une valeur littérale.

```
C#

int numberOfIterations = Convert.ToInt32(Console.ReadLine());
int currentMaximum = ExampleClass.ResultSoFar();
```

- N'utilisez pas le nom de la variable pour spécifier son type. Il peut ne pas être correct. Utilisez plutôt le type pour spécifier son type et utilisez le nom de la variable pour indiquer ses informations sémantiques. L'exemple suivant doit utiliser `string` pour le type et quelque chose comme `iterations` pour indiquer la signification des informations lues à partir de la console.

```
C#
```

```
var inputInt = Console.ReadLine();
Console.WriteLine(inputInt);
```

- Évitez d'utiliser `var` à la place de dynamic. Utilisez `dynamic` quand vous souhaitez une inférence de type d'exécution. Pour plus d'informations, consultez Utilisation du type dynamic (Guide de programmation C#).

- Utilisez le typage implicite pour la variable de boucle dans les boucles for.

  L'exemple suivant utilise un typage implicite dans une instruction `for`.

  C#

  ```
  var phrase =
  "lalalalalalalalalalalalalalalalalalalalalalalalalalalalalala";
  var manyPhrases = new StringBuilder();
  for (var i = 0; i < 10000; i++)
  {
      manyPhrases.Append(phrase);
  }
  //Console.WriteLine("tra" + manyPhrases);
  ```

- N'utilisez pas le typage implicite pour déterminer le type de la variable de boucle dans les boucles foreach. Dans la plupart des cas, le type d'éléments de la collection n'est pas immédiatement évident. Le nom de la collection ne doit pas être uniquement utilisé pour déduire le type de ses éléments.

  L'exemple suivant utilise un typage explicite dans une instruction `foreach`.

  C#

  ```
  foreach (char ch in laugh)
  {
      if (ch == 'h')
          Console.Write("H");
      else
          Console.Write(ch);
  }
  Console.WriteLine();
  ```

- Utilisez un type implicite pour les séquences de résultats dans les requêtes LINQ. La section sur LINQ explique que de nombreuses requêtes LINQ entraînent des types anonymes où des types implicites doivent être utilisés. D'autres requêtes entraînent des types génériques imbriqués où `var` est plus lisible.

> ⓘ **Notes**
>
> Veillez à ne pas modifier accidentellement un type d'élément de la collection itérable. Par exemple, il est facile de passer de **System.Linq.IQueryable** à **System.Collections.IEnumerable** dans une instruction `foreach`, ce qui modifie l'exécution d'une requête.

Certains de nos exemples expliquent le *type naturel* d'une expression. Ces exemples doivent utiliser `var` afin que le compilateur sélectionne le type naturel. Même si ces exemples sont moins évidents, l'utilisation de `var` est nécessaire pour chacun. Le texte doit expliquer le comportement.

Placer les directives using en dehors de la déclaration d'espace de noms

Lorsqu'une directive `using` se trouve en dehors d'une déclaration d'espace de noms, cet espace de noms importé représente son nom complet. Le nom qualifié complet est plus clair. Lorsque la directive `using` se trouve à l'intérieur de l'espace de noms, elle peut être relative à cet espace de noms ou à son nom complet.

C#

```
using Azure;

namespace CoolStuff.AwesomeFeature
{
    public class Awesome
    {
        public void Stuff()
        {
            WaitUntil wait = WaitUntil.Completed;
            // ...
        }
    }
}
```

En supposant qu'il existe une référence (directe ou indirecte) à la classe WaitUntil.

Nous allons maintenant légèrement modifier le code :

C#

```
namespace CoolStuff.AwesomeFeature
{
```

```
    using Azure;

    public class Awesome
    {
        public void Stuff()
        {
            WaitUntil wait = WaitUntil.Completed;
            // ...
        }
    }
}
```

Il compile aujourd'hui. Et demain. Mais la semaine suivante, le code précédent (non modifié) échoue en générant deux erreurs :

Console

```
- error CS0246: The type or namespace name 'WaitUntil' could not be found
(are you missing a using directive or an assembly reference?)
- error CS0103: The name 'WaitUntil' does not exist in the current context
```

L'une des dépendances a introduit cette classe dans un espace de noms, puis se termine par `.Azure` :

C#

```
namespace CoolStuff.Azure
{
    public class SecretsManagement
    {
        public string FetchFromKeyVault(string vaultId, string secretId) {
return null; }
    }
}
```

Une directive `using` placée à l'intérieur d'un espace de noms respecte le contexte et complique la résolution de noms. Dans cet exemple, c'est le premier espace de nom qu'elle trouve.

- `CoolStuff.AwesomeFeature.Azure`
- `CoolStuff.Azure`
- `Azure`

Ajout d'un nouvel espace de noms qui correspond à `CoolStuff.Azure` ou à `CoolStuff.AwesomeFeature.Azure` avant l'espace de noms global `Azure`. Vous pouvez le

résoudre en ajoutant le modificateur `global::` à la déclaration `using`. Mais il est plus facile de placer des déclarations `using` en dehors de l'espace de noms.

C#

```
namespace CoolStuff.AwesomeFeature
{
    using global::Azure;

    public class Awesome
    {
        public void Stuff()
        {
            WaitUntil wait = WaitUntil.Completed;
            // ...
        }
    }
}
```

Recommandations sur le style

En général, utilisez le format suivant pour les exemples de code :

- Utilisez quatre espaces pour la mise en retrait. N'utilisez pas d'onglets.
- Alignez le code de manière cohérente pour améliorer la lisibilité.
- Limitez les lignes à 65 caractères pour améliorer la lisibilité du code sur les documents, en particulier sur les écrans d'appareil mobile.
- Décomposez les instructions longues en plusieurs lignes pour améliorer la clarté.
- Utilisez le style « Allman » pour les accolades : les accolades ouvrantes et fermantes ouvrent et ferment leur propre nouvelle ligne. Les accolades s'alignent sur le niveau de retrait actuel.
- Les sauts de ligne doivent être insérés avant les opérateurs binaires, si nécessaire.

Style de commentaire

- Utilisez des commentaires à ligne unique (`//`) pour de brèves explications.
- Évitez les commentaires multilignes (`/* */`) pour les explications plus longues. Les commentaires ne sont pas localisés. Au lieu de cela, les explications plus longues sont données dans l'article complément.
- Placez le commentaire sur une ligne séparée, pas à la fin d'une ligne de code.
- Commencez le commentaire par une lettre majuscule.

- Terminez le texte de commentaire par un point.
- Insérez un espace entre le délimiteur de commentaire (//) et le texte du commentaire, comme illustré dans l'exemple suivant.

```
C#

// The following declaration creates a query. It does not run
// the query.
```

Conventions de disposition

Une bonne disposition utilise la mise en forme pour souligner la structure de votre code et en faciliter la lecture. Les exemples Microsoft et autres se conforment aux conventions suivantes :

- Utilisez les paramètres par défaut de l'éditeur de code (retrait intelligent, retrait de quatre caractères, tabulations enregistrées en tant qu'espaces). Pour plus d'informations, consultez Options, Éditeur de texte, C#, Mise en forme.
- Écrivez une seule instruction par ligne.
- Écrivez une seule déclaration par ligne.
- Si les lignes de continuation ne sont pas mises en retrait automatiquement, indentez-les à l'aide d'un taquet de tabulation (quatre espaces).
- Ajoutez au moins une ligne blanche entre les définitions des méthodes et les définitions des propriétés.
- Utilisez des parenthèses pour rendre apparentes les clauses d'une expression, comme illustré dans le code suivant.

```
C#

if ((startX > endX) && (startX > previousX))
{
    // Take appropriate action.
}
```

Il y a exception lorsque l'exemple explique la priorité de l'opérateur ou de l'expression.

Sécurité

Suivez les instructions indiquées dans Instructions de codage sécurisé.

Comment afficher les arguments de ligne de commande

Article • 08/06/2023

Les arguments fournis à un exécutable sur la ligne de commande sont accessibles dans instructions de niveau supérieur ou via un paramètre facultatif pour `Main`. Les arguments sont fournis sous la forme d'un tableau de chaînes. Chaque élément du tableau contient un argument. Les espaces blancs entre arguments sont supprimés. Par exemple, considérez ces appels de ligne de commande d'un fichier exécutable fictif :

Entrée sur la ligne de commande	Tableau de chaînes passé à Main
executable.exe a b c	"a" "b" "c"
executable.exe un deux	"un" "deux"
executable.exe "un deux" trois	"un deux" "trois"

> ⓘ Notes
>
> Quand vous exécutez une application dans Visual Studio, vous pouvez spécifier des arguments de ligne de commande dans la **page Déboguer du Concepteur de projets**.

Exemple

Cet exemple affiche les arguments de ligne de commande passés à une application de ligne de commande. La sortie présentée concerne la première entrée du tableau ci-dessus.

C#

```
// The Length property provides the number of array elements.
Console.WriteLine($"parameter count = {args.Length}");
```

```
for (int i = 0; i < args.Length; i++)
{
    Console.WriteLine($"Arg[{i}] = [{args[i]}]");
}

/* Output (assumes 3 cmd line args):
    parameter count = 3
    Arg[0] = [a]
    Arg[1] = [b]
    Arg[2] = [c]
*/
```

Voir aussi

- Vue d’ensemble de System.CommandLine
- Tutoriel : Prise en main de System.CommandLine

Explorez la programmation orientée objet avec des classes et des objets

Article • 01/06/2023

Dans ce didacticiel, vous allez créer une application console et voir les fonctionnalités orientées objet de base qui font partie du langage C#.

Prérequis

- Nous vous recommandons Visual Studio pour Windows ou Mac. Vous pouvez télécharger une version gratuite à partir de la page téléchargements de Visual Studio. Visual Studio inclut le Kit de développement logiciel (SDK) .NET.
- Vous pouvez également utiliser l'éditeur Visual Studio Code. Vous devez installer le dernier SDK .NET séparément.
- Si vous préférez un autre éditeur, vous devez installer le dernier kit SDK .NET.

Créer votre application

Dans une fenêtre de terminal, créez un répertoire nommé *classes*. Vous y créerez votre application. Sélectionnez ce répertoire et tapez `dotnet new console` dans la fenêtre de console. Cette commande crée votre application. Ouvrez *Program.cs*. Il doit se présenter comme suit :

C#

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

Dans ce tutoriel, vous allez créer des types qui représentent un compte bancaire. Les développeurs définissent généralement chaque classe dans un fichier texte différent. Cela simplifie la gestion au fur et à mesure qu'un programme augmente en taille. Créez un fichier nommé *BankAccount.cs* dans le répertoire *Classes*.

Ce fichier contiendra la définition d'un ***compte bancaire***. La programmation orientée objet organise le code en créant des types sous la forme de ***classes***. Ces classes contiennent le code qui représente une entité spécifique. La classe `BankAccount` représente un compte bancaire. Le code implémente des opérations spécifiques à travers des méthodes et des propriétés. Dans ce tutoriel, le compte bancaire prend en charge le comportement suivant :

1. Il contient un numéro à 10 chiffres qui identifie le compte bancaire de manière unique.
2. Il contient une chaîne qui stocke le nom du ou des détenteurs.
3. Le solde peut être récupéré.
4. Il accepte les dépôts.
5. Il accepte les retraits.
6. Le solde initial doit être positif.
7. Les retraits ne peuvent pas générer un solde négatif.

Définir le type de compte bancaire

Vous pouvez commencer par créer les éléments de base d'une classe définissant ce comportement. Créez un fichier à l'aide de la commande **File:New**. Nommez-le *BankAccount.cs*. Ajoutez le code suivant dans le fichier *BankAccount.cs* :

C#

```
namespace Classes;

public class BankAccount
{
    public string Number { get; }
    public string Owner { get; set; }
    public decimal Balance { get; }

    public void MakeDeposit(decimal amount, DateTime date, string note)
    {
    }

    public void MakeWithdrawal(decimal amount, DateTime date, string note)
    {
    }
}
```

Avant de poursuivre, examinons ce que vous venez de créer. La déclaration `namespace` permet d'organiser logiquement votre code. Ce tutoriel étant relativement petit, vous allez placer tout le code dans un même espace de noms.

`public class BankAccount` définit la classe, ou le type, que vous créez. Tout ce qui est situé entre `{` et `}` après la déclaration de classe définit l'état et le comportement de la classe. La classe `BankAccount` a cinq ***membres***. Les trois premiers sont des ***propriétés***. Les propriétés sont des éléments de données qui peuvent avoir un code qui applique la validation ou d'autres règles. Les deux derniers sont des ***méthodes***. Les méthodes sont des blocs de code qui effectuent une fonction unique. La lecture des noms de chacun

des membres doit fournir suffisamment d'informations pour vous permettre (ou à tout autre développeur) de comprendre ce que fait la classe.

Ouvrir un nouveau compte

La première fonctionnalité à implémenter est l'ouverture d'un compte bancaire. Quand un client ouvre un compte, il doit fournir un solde initial, ainsi que des informations sur le ou les détenteurs du compte.

La création d'un objet de type `BankAccount` implique la définition d'un ***constructeur*** qui affecte ces valeurs. Un ***constructeur*** est un membre qui porte le même nom que la classe. Il est utilisé pour initialiser des objets de ce type de classe. Ajoutez le constructeur suivant au type `BankAccount`. Placez le code suivant au-dessus de la déclaration de `MakeDeposit` :

```
C#

public BankAccount(string name, decimal initialBalance)
{
    this.Owner = name;
    this.Balance = initialBalance;
}
```

Le code précédent identifie les propriétés de l'objet en cours de construction en incluant le qualificateur `this`. Ce qualificateur est généralement facultatif et omis. Vous auriez également pu écrire :

```
C#

public BankAccount(string name, decimal initialBalance)
{
    Owner = name;
    Balance = initialBalance;
}
```

Le qualificateur `this` est requis uniquement quand une variable ou un paramètre local porte le même nom que ce champ ou cette propriété. Le qualificateur `this` est omis dans le reste de cet article, sauf s'il est nécessaire.

Les constructeurs sont appelés quand vous créez un objet à l'aide de new. Remplacez la ligne `Console.WriteLine("Hello World!");` dans *Program.cs* par le code suivant (remplacez `<name>` par votre nom) :

```
```

C#

```
using Classes;

var account = new BankAccount("<name>", 1000);
Console.WriteLine($"Account {account.Number} was created for {account.Owner}
with {account.Balance} initial balance.");
```

Exécutons ce que vous avez créé à ce stade. Si vous utilisez Visual Studio, sélectionnez **Démarrer sans débogage** dans le menu **Déboguer**. Si vous utilisez une ligne de commande, tapez `dotnet run` dans le répertoire où vous avez créé votre projet.

Avez-vous remarqué que le numéro de compte est vide ? L'heure est venue de traiter ce point. Le numéro de compte doit être assigné une fois l'objet construit. Mais ce ne devrait pas être à l'appelant de le créer. Le code de la classe `BankAccount` devrait savoir comment assigner de nouveaux numéros de compte. Un moyen simple consiste à commencer avec un nombre à 10 chiffres. Incrémentez-le chaque fois qu'un compte est créé. Enfin, stockez le numéro de compte actuel quand un objet est construit.

Ajoutez une déclaration de membre à la classe `BankAccount`. Placez la ligne de code suivante après l'accolade ouvrante `{` au début de la classe `BankAccount` :

C#

```
private static int accountNumberSeed = 1234567890;
```

`accountNumberSeed` est un membre de données. Celui-ci est `private`, ce qui signifie qu'il est uniquement accessible par code dans la classe `BankAccount`. C'est un moyen de séparer les responsabilités publiques (comme le fait de disposer d'un numéro de compte) de l'implémentation privée (c'est-à-dire la façon dont les numéros de compte sont générés). Il est également `static`, ce qui signifie qu'il est partagé par tous les objets `BankAccount`. La valeur d'une variable non statique est unique pour chaque instance de l'objet `BankAccount`. Ajoutez les deux lignes suivantes au constructeur pour affecter le numéro de compte. Placez-les après la ligne qui stipule `this.Balance = initialBalance` :

C#

```
this.Number = accountNumberSeed.ToString();
accountNumberSeed++;
```

Tapez `dotnet run` pour afficher les résultats.

Créer des dépôts et des retraits

La classe de votre compte bancaire doit accepter les dépôts et les retraits pour fonctionner correctement. Nous allons implémenter des dépôts et des retraits en créant un journal de chaque transaction du compte. Le suivi de chaque transaction présente plusieurs avantages par rapport à la simple mise à jour du solde à chaque transaction. L'historique peut être utilisé pour auditer toutes les transactions et gérer les soldes au quotidien. Le calcul du solde à partir de l'historique de toutes les transactions, lorsque nécessaire, garantit que toute erreur corrigée dans une transaction individuelle sera reflétée correctement dans le solde lors du calcul suivant.

Commençons par créer un type pour représenter une transaction. La transaction est un type simple qui n'a aucune responsabilité. Il a besoin de quelques propriétés. Créez un fichier nommé *Transaction.cs*. Ajoutez-lui le code suivant :

C#

```
namespace Classes;

public class Transaction
{
    public decimal Amount { get; }
    public DateTime Date { get; }
    public string Notes { get; }

    public Transaction(decimal amount, DateTime date, string note)
    {
        Amount = amount;
        Date = date;
        Notes = note;
    }
}
```

Nous allons maintenant ajouter une List<T> d'objets `Transaction` à la classe `BankAccount`. Ajoutez la déclaration suivante après le constructeur dans votre fichier *BankAccount.cs* :

C#

```
private List<Transaction> allTransactions = new List<Transaction>();
```

À présent, nous allons calculer correctement `Balance`. Le solde actuel peut être obtenu en additionnant les valeurs de toutes les transactions. Dans l'état actuel du code, vous ne pouvez obtenir que le solde initial du compte. Vous devrez donc mettre à jour la

propriété `Balance`. Remplacez la ligne `public decimal Balance { get; }` dans *BankAccount.cs* par le code suivant :

C#

```
public decimal Balance
{
    get
    {
        decimal balance = 0;
        foreach (var item in allTransactions)
        {
            balance += item.Amount;
        }

        return balance;
    }
}
```

Cet exemple montre un aspect important des ***propriétés***. Vous calculez à présent le solde quand un autre programmeur en demande la valeur. Votre calcul énumère toutes les transactions et retourne la somme en tant que solde actuel.

Implémentez ensuite les méthodes `MakeDeposit` et `MakeWithdrawal`. Ces méthodes appliquent les deux règles finales : le solde initial doit être positif et un retrait ne doit pas créer un solde négatif.

Ces règles introduisent le concept d'***exceptions***. La manière standard d'indiquer qu'une méthode ne peut pas effectuer correctement son travail consiste à lever une exception. Le type d'exception et le message associé décrivent l'erreur. Ici, la méthode `MakeDeposit` lève une exception si le montant du dépôt n'est pas supérieur à 0. La méthode `MakeWithdrawal` lève une exception si le montant du retrait n'est pas supérieur à 0 ou si l'application du retrait entraîne un solde négatif. Ajoutez le code suivant après la déclaration de la liste `allTransactions` :

C#

```
public void MakeDeposit(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of
deposit must be positive");
    }
    var deposit = new Transaction(amount, date, note);
    allTransactions.Add(deposit);
}
```

```
public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of
withdrawal must be positive");
    }
    if (Balance - amount < 0)
    {
        throw new InvalidOperationException("Not sufficient funds for this
withdrawal");
    }
    var withdrawal = new Transaction(-amount, date, note);
    allTransactions.Add(withdrawal);
}
```

L'instruction throw**lève** une exception. L'exécution du bloc actuel se termine et le contrôle est transféré au premier bloc correspondant `catch` trouvé dans la pile des appels. Vous ajouterez un bloc `catch` pour tester ce code un peu plus tard.

Le constructeur devrait obtenir une modification lui permettant d'ajouter une transaction initiale au lieu de mettre directement le solde à jour. Étant donné que vous avez déjà écrit la méthode `MakeDeposit`, appelez-la à partir de votre constructeur. Le constructeur terminé doit être semblable à ce qui suit :

C#

```
public BankAccount(string name, decimal initialBalance)
{
    Number = accountNumberSeed.ToString();
    accountNumberSeed++;

    Owner = name;
    MakeDeposit(initialBalance, DateTime.Now, "Initial balance");
}
```

DateTime.Now est une propriété qui retourne la date et l'heure actuelles. Testez ce code en ajoutant quelques dépôts et retraits dans votre méthode `Main`, à la suite du code qui crée un nouveau `BankAccount` :

C#

```
account.MakeWithdrawal(500, DateTime.Now, "Rent payment");
Console.WriteLine(account.Balance);
account.MakeDeposit(100, DateTime.Now, "Friend paid me back");
Console.WriteLine(account.Balance);
```

Ensuite, vérifiez que vous interceptez des conditions d'erreur en essayant de créer un compte avec un solde négatif. Ajoutez le code suivant après le code précédent que vous venez d'ajouter :

C#

```
// Test that the initial balances must be positive.
BankAccount invalidAccount;
try
{
    invalidAccount = new BankAccount("invalid", -55);
}
catch (ArgumentOutOfRangeException e)
{
    Console.WriteLine("Exception caught creating account with negative
balance");
    Console.WriteLine(e.ToString());
    return;
}
```

Vous utilisez l'instruction try-catch pour marquer un bloc de code pouvant lever des exceptions et intercepter les erreurs que vous attendez. Vous pouvez utiliser la même technique pour tester le code qui lève une exception pour un solde négatif. Ajoutez le code suivant avant la déclaration de `invalidAccount` dans votre méthode `Main` :

C#

```
// Test for a negative balance.
try
{
    account.MakeWithdrawal(750, DateTime.Now, "Attempt to overdraw");
}
catch (InvalidOperationException e)
{
    Console.WriteLine("Exception caught trying to overdraw");
    Console.WriteLine(e.ToString());
}
```

Enregistrez le fichier et tapez `dotnet run` pour effectuer un essai.

Test : consigner toutes les transactions

Pour terminer ce tutoriel, vous pouvez écrire la méthode `GetAccountHistory` qui crée une `string` pour l'historique des transactions. Ajoutez cette méthode au type `BankAccount` :

C#

```
public string GetAccountHistory()
{
    var report = new System.Text.StringBuilder();

    decimal balance = 0;
    report.AppendLine("Date\t\tAmount\tBalance\tNote");
    foreach (var item in allTransactions)
    {
        balance += item.Amount;
        report.AppendLine($"
{item.Date.ToShortDateString()}\t{item.Amount}\t{balance}\t{item.Notes}");
    }

    return report.ToString();
}
```

L'historique utilise la classe StringBuilder pour mettre en forme une chaîne contenant une ligne pour chaque transaction. Vous avez vu le code de mise en forme de chaîne précédemment dans ces tutoriels. Vous pouvez observer le nouveau caractère `\t`. Celui-ci insère une tabulation pour mettre en forme la sortie.

Ajoutez cette ligne pour effectuer un essai dans *Program.cs* :

C#

```
Console.WriteLine(account.GetAccountHistory());
```

Exécutez votre programme pour voir les résultats.

Étapes suivantes

Si vous êtes bloqué, vous pouvez afficher la source de ce didacticiel dans notre dépôt GitHub.

Vous pouvez continuer avec le didacticiel sur la programmation orientée objet.

Pour en savoir plus sur ces concepts, consultez les articles :

- Instructions de sélection
- Instructions d'itération

Programmation orientée objet (C#)

Article • 08/06/2023

C# est un langage de programmation orienté objet. Les quatre principes de base de la programmation orientée objet sont les suivants :

- *Abstraction* Modélisation des attributs et interactions pertinents des entités en tant que classes, afin de définir une représentation abstraite d'un système.
- *Encapsulation* Masquage de l'état interne et des fonctionnalités d'un objet et autorisation d'accès uniquement par le biais d'un ensemble public de fonctions.
- *Héritage* Possibilité de créer des abstractions basées sur des abstractions existantes.
- *Polymorphisme* Possibilité d'implémenter des propriétés ou des méthodes héritées de différentes manières parmi plusieurs abstractions.

Dans le didacticiel précédent, Introduction aux classes, vous avez vu à la fois *l'abstraction* et *l'encapsulation*. La classe `BankAccount` a fourni une abstraction pour le concept d'un compte bancaire. Vous pouvez modifier son implémentation sans affecter le code qui a utilisé la classe `BankAccount`. Les classes `BankAccount` et `Transaction` fournissent l'encapsulation des composants nécessaires pour décrire ces concepts dans le code.

Dans ce didacticiel, vous allez étendre cette application pour utiliser *l'héritage* et le *polymorphisme*, afin d'ajouter de nouvelles fonctionnalités. Vous allez également ajouter des fonctionnalités à la classe `BankAccount`, en tirant parti des techniques *d'abstraction* et *d'encapsulation* que vous avez apprises dans le didacticiel précédent.

Créer différents types de comptes

Après avoir généré ce programme, vous recevez des demandes d'ajout de fonctionnalités. Il fonctionne très bien dans la situation où il n'y a qu'un seul type de compte bancaire. Au fil du temps, les besoins changent et des types de comptes associés sont demandés :

- Un compte générant des intérêts qui s'accumulent à la fin de chaque mois.
- Une marge de crédit qui peut avoir un solde négatif, mais quand il y a un solde, il y a des frais d'intérêt chaque mois.
- Un compte de carte cadeau prépayée qui commence par un seul versement et qui peut uniquement être remboursé. Il peut être rempli une fois au début de chaque mois.

Tous ces différents comptes sont similaires à la classe `BankAccount` définie dans le didacticiel précédent. Vous pouvez copier ce code, renommer les classes et apporter des modifications. Cette technique fonctionnerait à court terme, mais engendrerait davantage de travail au fil du temps. Toutes les modifications seraient copiées dans toutes les classes affectées.

Au lieu de cela, vous pouvez créer de nouveaux types de comptes bancaires qui héritent des méthodes et des données de la classe `BankAccount` créée dans le didacticiel précédent. Ces nouvelles classes peuvent élargir la classe `BankAccount` avec le comportement spécifique nécessaire pour chaque type :

C#

```
public class InterestEarningAccount : BankAccount
{
}

public class LineOfCreditAccount : BankAccount
{
}

public class GiftCardAccount : BankAccount
{
}
```

Chacune de ces classes *hérite* du comportement partagé de sa *classe de base* partagée, la classe `BankAccount`. Écrivez les implémentations pour les fonctionnalités nouvelles et différentes dans chacune des *classes dérivées*. Ces classes dérivées possèdent déjà tout le comportement défini dans la classe `BankAccount`.

Il est recommandé de créer chaque nouvelle classe dans un fichier source différent. Dans Visual Studio, vous pouvez cliquer avec le bouton droit sur le projet, puis sélectionner *Ajouter une classe* pour ajouter une nouvelle classe dans un nouveau fichier. Dans Visual Studio Code, sélectionnez *Fichier*, puis *Nouveau* pour créer un nouveau fichier source. Dans l'un des deux outils, nommez le fichier pour qu'il corresponde à la classe : *InterestEarningAccount.cs*, *LineOfCreditAccount.cs* et *GiftCardAccount.cs*.

Lorsque vous créez les classes comme indiqué dans l'exemple précédent, vous constatez qu'aucune de vos classes dérivées ne se compile. Un constructeur est responsable de l'initialisation d'un objet. Un constructeur de classe dérivée doit initialiser la classe dérivée et fournir des instructions sur la manière d'initialiser l'objet de classe de base inclus dans la classe dérivée. L'initialisation appropriée se produit normalement sans code supplémentaire. La classe `BankAccount` déclare un constructeur public avec la signature suivante :

C#

```
public BankAccount(string name, decimal initialBalance)
```

Le compilateur ne génère pas de constructeur par défaut lorsque vous définissez vous-même un constructeur. Cela signifie que chaque classe dérivée doit appeler explicitement ce constructeur. Vous déclarez un constructeur qui peut passer des arguments au constructeur de classe de base. Le code suivant montre le constructeur pour la `InterestEarningAccount` :

C#

```
public InterestEarningAccount(string name, decimal initialBalance) :
base(name, initialBalance)
{
}
```

Les paramètres de ce nouveau constructeur correspondent au type de paramètre et aux noms du constructeur de classe de base. Vous utilisez la syntaxe `: base()` pour indiquer un appel à un constructeur de classe de base. Certaines classes définissent plusieurs constructeurs, et cette syntaxe vous permet de choisir le constructeur de classe de base que vous appelez. Une fois que vous avez mis à jour les constructeurs, vous pouvez développer le code pour chacune des classes dérivées. Les exigences pour les nouvelles classes peuvent être indiquées comme suit :

- Un compte porteur d'intérêt :
  - Obtiendra un crédit de 2 % du solde de fin de mois.
- Une marge de crédit :
  - Peut avoir un solde négatif, mais ne pas être supérieur en valeur absolue à la limite de crédit.
  - Entraîne des frais d'intérêt chaque mois lorsque le solde de fin de mois n'est pas de 0.
  - Entraîne des frais sur chaque retrait qui dépasse la limite de crédit.
- Un compte de carte cadeau :
  - Peut être rempli avec un montant spécifié une fois par mois, le dernier jour du mois.

Vous pouvez voir que ces trois types de compte ont une action qui a lieu à la fin de chaque mois. Toutefois, chaque type de compte effectue des tâches différentes. Vous utilisez le *polymorphisme* pour implémenter ce code. Créez une méthode unique `virtual` dans la classe `BankAccount` :

```
```

C#

```
public virtual void PerformMonthEndTransactions() { }
```

Le code précédent montre comment vous utilisez le mot clé `virtual` pour déclarer une méthode dans la classe de base pour laquelle une classe dérivée peut fournir une implémentation différente. Une méthode `virtual` est une méthode dans laquelle n'importe quelle classe dérivée peut choisir d'implémenter de nouveau. Les classes dérivées utilisent le mot clé `override` pour définir la nouvelle implémentation. En règle générale, on parle de « substitution de l'implémentation de la classe de base ». Le mot clé `virtual` spécifie que les classes dérivées peuvent remplacer le comportement. Vous pouvez également déclarer des méthodes `abstract` où les classes dérivées doivent remplacer le comportement. La classe de base ne fournit pas d'implémentation pour une méthode `abstract`. Ensuite, vous devez définir l'implémentation de deux des nouvelles classes que vous avez créées. Commencez par la `InterestEarningAccount` :

C#

```
public override void PerformMonthEndTransactions()
{
    if (Balance > 500m)
    {
        decimal interest = Balance * 0.05m;
        MakeDeposit(interest, DateTime.Now, "apply monthly interest");
    }
}
```

Ajoutez le code suivant à `LineOfCreditAccount`. Le code annule le solde pour calculer une charge d'intérêt positive qui est retirée du compte :

C#

```
public override void PerformMonthEndTransactions()
{
    if (Balance < 0)
    {
        // Negate the balance to get a positive interest charge:
        decimal interest = -Balance * 0.07m;
        MakeWithdrawal(interest, DateTime.Now, "Charge monthly interest");
    }
}
```

La classe `GiftCardAccount` a besoin de deux modifications pour implémenter sa fonctionnalité de fin de mois. Tout d'abord, modifiez le constructeur pour inclure un montant facultatif à ajouter chaque mois :

C#

```
private readonly decimal _monthlyDeposit = 0m;

public GiftCardAccount(string name, decimal initialBalance, decimal
monthlyDeposit = 0) : base(name, initialBalance)
    => _monthlyDeposit = monthlyDeposit;
```

Le constructeur fournit une valeur par défaut pour la valeur `monthlyDeposit`, afin que les appelants puissent omettre un `0` sans aucun versement mensuel. Ensuite, remplacez la méthode `PerformMonthEndTransactions` pour ajouter le versement mensuel, s'il a été défini sur une valeur différente de zéro dans le constructeur :

C#

```
public override void PerformMonthEndTransactions()
{
    if (_monthlyDeposit != 0)
    {
        MakeDeposit(_monthlyDeposit, DateTime.Now, "Add monthly deposit");
    }
}
```

Le remplacement applique l'ensemble des versements mensuels dans le constructeur. Ajoutez le code suivant à la méthode `Main` pour tester ces modifications pour `GiftCardAccount` et `InterestEarningAccount` :

C#

```
var giftCard = new GiftCardAccount("gift card", 100, 50);
giftCard.MakeWithdrawal(20, DateTime.Now, "get expensive coffee");
giftCard.MakeWithdrawal(50, DateTime.Now, "buy groceries");
giftCard.PerformMonthEndTransactions();
// can make additional deposits:
giftCard.MakeDeposit(27.50m, DateTime.Now, "add some additional spending
money");
Console.WriteLine(giftCard.GetAccountHistory());

var savings = new InterestEarningAccount("savings account", 10000);
savings.MakeDeposit(750, DateTime.Now, "save some money");
savings.MakeDeposit(1250, DateTime.Now, "Add more savings");
savings.MakeWithdrawal(250, DateTime.Now, "Needed to pay monthly bills");
savings.PerformMonthEndTransactions();
Console.WriteLine(savings.GetAccountHistory());
```

Vérifier les résultats. À présent, ajoutez un ensemble similaire de code de test pour `LineOfCreditAccount` :

C#

```
var lineOfCredit = new LineOfCreditAccount("line of credit", 0);
// How much is too much to borrow?
lineOfCredit.MakeWithdrawal(1000m, DateTime.Now, "Take out monthly
advance");
lineOfCredit.MakeDeposit(50m, DateTime.Now, "Pay back small amount");
lineOfCredit.MakeWithdrawal(5000m, DateTime.Now, "Emergency funds for
repairs");
lineOfCredit.MakeDeposit(150m, DateTime.Now, "Partial restoration on
repairs");
lineOfCredit.PerformMonthEndTransactions();
Console.WriteLine(lineOfCredit.GetAccountHistory());
```

Lorsque vous ajoutez le code précédent et que vous exécutez le programme, vous voyez une erreur similaire à celle-ci :

Console

```
Unhandled exception. System.ArgumentOutOfRangeException: Amount of deposit
must be positive (Parameter 'amount')
   at OOProgramming.BankAccount.MakeDeposit(Decimal amount, DateTime date,
String note) in BankAccount.cs:line 42
   at OOProgramming.BankAccount..ctor(String name, Decimal initialBalance)
in BankAccount.cs:line 31
   at OOProgramming.LineOfCreditAccount..ctor(String name, Decimal
initialBalance) in LineOfCreditAccount.cs:line 9
   at OOProgramming.Program.Main(String[] args) in Program.cs:line 29
```

ⓘ Notes

La sortie réelle inclut le chemin d'accès complet vers le dossier avec le projet. Les noms de dossiers ont été omis par souci de brièveté. En outre, selon le format de votre code, les numéros de ligne peuvent être légèrement différents.

Ce code échoue, car le `BankAccount` suppose que le solde initial doit être supérieur à 0. Une autre hypothèse inscrite dans la classe `BankAccount` est que le solde ne peut pas passer au négatif. Au contraire, tout retrait qui met à découvert le compte est rejeté. Ces deux hypothèses doivent changer. Le compte de marge de crédit commence à 0 et a généralement un solde négatif. En outre, si un client emprunte trop d'argent, il encourt des frais. La transaction est acceptée, mais elle coûte plus cher. La première règle peut être implémentée en ajoutant un argument facultatif au constructeur `BankAccount` qui spécifie le solde minimal. Par défaut, il s'agit de `0`. La deuxième règle nécessite un mécanisme qui permet aux classes dérivées de modifier l'algorithme par défaut. Dans un sens, la classe de base « demande » au type dérivé ce qui doit se produire en cas de

découvert. Le comportement par défaut consiste à rejeter la transaction en levant une exception.

Commençons par ajouter un deuxième constructeur qui inclut un paramètre `minimumBalance` facultatif. Ce nouveau constructeur effectue toutes les actions effectuées par le constructeur existant. En outre, il définit la propriété d'équilibre minimal. Vous pouvez copier le corps du constructeur existant, mais cela signifie que deux emplacements doivent être modifiés à l'avenir. Au lieu de cela, vous pouvez utiliser le *chaînage du constructeur* pour qu'un constructeur en appelle un autre. Le code suivant montre les deux constructeurs et le nouveau champ supplémentaire :

C#

```
private readonly decimal _minimumBalance;

public BankAccount(string name, decimal initialBalance) : this(name,
initialBalance, 0) { }

public BankAccount(string name, decimal initialBalance, decimal
minimumBalance)
{
    Number = s_accountNumberSeed.ToString();
    s_accountNumberSeed++;

    Owner = name;
    _minimumBalance = minimumBalance;
    if (initialBalance > 0)
        MakeDeposit(initialBalance, DateTime.Now, "Initial balance");
}
```

Le code précédent montre deux nouvelles techniques. Tout d'abord, le champ `minimumBalance` est marqué comme `readonly`. Cela signifie que la valeur ne peut pas être modifiée une fois l'objet construit. Une fois qu'un `BankAccount` est créé, le `minimumBalance` ne peut pas changer. Deuxièmement, le constructeur qui prend deux paramètres utilise `: this(name, initialBalance, 0) { }` comme implémentation. L'expression `: this()` appelle l'autre constructeur, celui avec trois paramètres. Cette technique vous permet d'avoir une implémentation unique pour initialiser un objet, même si le code client peut en choisir un parmi de nombreux constructeurs.

Cette implémentation appelle `MakeDeposit` uniquement si le solde initial est supérieur à `0`. Cela maintient la règle selon laquelle les versements doivent être positifs, tout en permettant au compte de crédit d'ouvrir avec un solde `0`.

Maintenant que la classe `BankAccount` a un champ en lecture seule pour le solde minimal, la dernière modification consiste à remplacer le code en dur `0` par

`minimumBalance` dans la méthode `MakeWithdrawal` :

```
C#

if (Balance - amount < minimumBalance)
```

Après avoir étendu la classe `BankAccount`, vous pouvez modifier le constructeur `LineOfCreditAccount` pour appeler le nouveau constructeur de base, comme indiqué dans le code suivant :

```
C#

public LineOfCreditAccount(string name, decimal initialBalance, decimal
creditLimit) : base(name, initialBalance, -creditLimit)
{
}
```

Notez que le constructeur `LineOfCreditAccount` modifie le signe du paramètre `creditLimit`, afin qu'il corresponde à la signification du paramètre `minimumBalance`.

Différentes règles de découvert

La dernière fonctionnalité à ajouter permet à `LineOfCreditAccount` de facturer des frais en cas de dépassement de la limite de crédit, plutôt que de refuser la transaction.

Une technique consiste à définir une fonction virtuelle dans laquelle vous implémentez le comportement requis. La classe `BankAccount` refactorise la méthode `MakeWithdrawal` en deux méthodes. La nouvelle méthode effectue l'action spécifiée lorsque le retrait amène le solde sous le minimum. La méthode existante `MakeWithdrawal` a le code suivant :

```
C#

public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of
withdrawal must be positive");
    }
    if (Balance - amount < _minimumBalance)
    {
        throw new InvalidOperationException("Not sufficient funds for this
withdrawal");
    }
```

```
    var withdrawal = new Transaction(-amount, date, note);
    allTransactions.Add(withdrawal);
}
```

Remplacez-le par le code suivant :

C#

```
public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of
withdrawal must be positive");
    }
    Transaction? overdraftTransaction = CheckWithdrawalLimit(Balance -
amount < _minimumBalance);
    Transaction? withdrawal = new(-amount, date, note);
    _allTransactions.Add(withdrawal);
    if (overdraftTransaction != null)
        _allTransactions.Add(overdraftTransaction);
}

protected virtual Transaction? CheckWithdrawalLimit(bool isOverdrawn)
{
    if (isOverdrawn)
    {
        throw new InvalidOperationException("Not sufficient funds for this
withdrawal");
    }
    else
    {
        return default;
    }
}
```

La méthode ajoutée est `protected`, ce qui signifie qu'elle peut être appelée uniquement à partir de classes dérivées. Cette déclaration empêche d'autres clients d'appeler la méthode. Elle est également `virtual`, afin que les classes dérivées puissent modifier le comportement. Le type de retour est `Transaction?`. L'annotation `?` indique que la méthode peut retourner `null`. Ajoutez l'implémentation suivante dans le `LineOfCreditAccount` pour facturer des frais lorsque la limite de retrait est dépassée :

C#

```
protected override Transaction? CheckWithdrawalLimit(bool isOverdrawn) =>
    isOverdrawn
    ? new Transaction(-20, DateTime.Now, "Apply overdraft fee")
    : default;
```

Le remplacement retourne une transaction de frais lorsque le compte est à découvert. Si le retrait ne dépasse pas la limite, la méthode retourne une transaction `null`. Ceci indique qu'il n'y a pas de frais. Testez ces modifications en ajoutant le code suivant à votre méthode `Main` dans la classe `Program` :

C#

```
var lineOfCredit = new LineOfCreditAccount("line of credit", 0, 2000);
// How much is too much to borrow?
lineOfCredit.MakeWithdrawal(1000m, DateTime.Now, "Take out monthly
advance");
lineOfCredit.MakeDeposit(50m, DateTime.Now, "Pay back small amount");
lineOfCredit.MakeWithdrawal(5000m, DateTime.Now, "Emergency funds for
repairs");
lineOfCredit.MakeDeposit(150m, DateTime.Now, "Partial restoration on
repairs");
lineOfCredit.PerformMonthEndTransactions();
Console.WriteLine(lineOfCredit.GetAccountHistory());
```

Exécutez le programme et vérifiez les résultats.

Résumé

Si vous êtes bloqué, vous pouvez afficher la source de ce didacticiel dans notre référentiel GitHub.

Ce didacticiel a présenté un grand nombre des techniques utilisées dans la programmation orientée objet :

- Vous avez utilisé *l'abstraction* lorsque vous avez défini des classes pour chacun des différents types de compte. Ces classes ont décrit le comportement de ce type de compte.
- Vous avez utilisé *l'encapsulation* lorsque vous avez conservé de nombreux détails `private` dans chaque classe.
- Vous avez utilisé *l'héritage* lorsque vous avez tiré parti de l'implémentation déjà créée dans la classe `BankAccount` pour enregistrer le code.
- Vous avez utilisé *le polymorphisme* lorsque vous avez créé `virtual` des méthodes que les classes dérivées peuvent remplacer pour créer un comportement spécifique pour ce type de compte.

Héritage dans C# et .NET

Article • 10/05/2023

Ce didacticiel vous présente l'héritage dans C#. L'héritage est une fonctionnalité des langages de programmation orientés objet qui vous permet de définir une classe de base qui fournit des fonctionnalités spécifiques (données et comportement) et de définir des classes dérivées qui héritent ou substituent cette fonctionnalité.

Prérequis

- Nous vous recommandons Visual Studio pour Windows ou Mac. Vous pouvez télécharger une version gratuite à partir de la page téléchargements de Visual Studio. Visual Studio inclut le Kit de développement logiciel (SDK) .NET.
- Vous pouvez également utiliser l'éditeur Visual Studio Code. Vous devez installer le dernier SDK .NET séparément.
- Si vous préférez un autre éditeur, vous devez installer le dernier SDK .NET.

Exécution des exemples

Pour créer et exécuter les exemples de ce didacticiel, vous utilisez l'utilitaire dotnet en ligne de commande. Pour chaque exemple, procédez comme suit :

1. Créez un répertoire pour stocker l'exemple.

2. Entrez la commande dotnet new console dans l'invite de commandes pour créer un projet .NET Core.

3. Copiez et collez le code de l'exemple dans votre éditeur de code.

4. Entrez la commande dotnet restore à partir de la ligne de commande pour charger ou restaurer les dépendances du projet.

   Vous n'avez pas besoin d'exécuter dotnet restore, car il est exécuté implicitement par toutes les commandes qui nécessitent une restauration pour se produire, comme `dotnet new`, `dotnet build`, `dotnet run`, `dotnet test`, `dotnet publish` et `dotnet pack`. Pour désactiver la restauration implicite, utilisez l'option `--no-restore` .

   La commande `dotnet restore` est toujours utile dans certains scénarios où la restauration explicite est logique, comme les builds d'intégration continue dans

Azure DevOps Services ou dans les systèmes de génération qui doivent contrôler explicitement le moment où la restauration se produit.

Pour plus d'informations sur la gestion des flux NuGet, consultez la documentation dotnet restore.

5. Entrez la commande dotnet run pour compiler et exécuter l'exemple.

Présentation : Qu'est-ce que l'héritage ?

L'héritage est un des attributs fondamentaux de la programmation orientée objet. Il vous permet de définir une classe enfant qui réutilise (hérite), étend ou modifie le comportement d'une classe parente. La classe dont les membres sont hérités s'appelle la *classe de base*. La classe qui hérite des membres de la classe de base est appelée la *classe dérivée*.

C# et .NET prennent uniquement en charge *l'héritage simple*. C'est-à-dire qu'une classe ne peut hériter que d'une seule classe. Toutefois, l'héritage est transitif, ce qui permet de définir une hiérarchie d'héritage pour un ensemble de types. En d'autres termes, le type `D` peut hériter du type `C`, qui hérite du type `B`, qui hérite du type de classe de base `A`. Étant donné que l'héritage est transitif, les membres de type `A` sont disponibles pour le type `D`.

Tous les membres d'une classe de base ne sont pas hérités par les classes dérivées. Les membres suivants ne sont pas hérités :

- Les constructeurs statiques, qui initialisent les données statiques d'une classe.
- Les Constructeurs d'instance, que vous appelez pour créer une nouvelle instance de la classe. Chaque classe doit définir ses propres constructeurs.
- Les finaliseurs, qui sont appelés par le récupérateur de mémoire du runtime pour détruire les instances d'une classe.

Bien que tous les autres membres de classe de base sont hérités par les classes dérivées, leur visibilité dépend de leur accessibilité. L'accessibilité d'un membre affecte sa visibilité pour les classes dérivées de la manière suivante :

- Les membres Privés sont visibles uniquement dans les classes dérivées qui sont imbriquées dans leur classe de base. Sinon, ils ne sont pas visibles dans les classes dérivées. Dans l'exemple suivant, `A.B` est une classe imbriquée qui dérive de `A`, et `C` dérive de `A`. Le champ `A._value` privé est visible dans A.B. Toutefois, si vous supprimez les commentaires de la méthode `C.GetValue` et essayez de compiler

l'exemple, il génère l'erreur de compilateur CS0122 : « 'A.value est inaccessible en raison de son niveau de protection ».

C#

```
public class A
{
    private int _value = 10;

    public class B : A
    {
        public int GetValue()
        {
            return _value;
        }
    }
}

public class C : A
{
    //    public int GetValue()
    //    {
    //        return _value;
    //    }
}

public class AccessExample
{
    public static void Main(string[] args)
    {
        var b = new A.B();
        Console.WriteLine(b.GetValue());
    }
}
// The example displays the following output:
//       10
```

- Les membres protégés sont visibles uniquement dans les classes dérivées.
- Les membres internes sont visibles uniquement dans les classes dérivées qui sont trouvent dans le même assembly que la classe de base. Ils ne sont pas visibles dans les classes dérivées situées dans un autre assembly à partir de la classe de base.
- Les membres publics sont visibles dans les classes dérivées et font partie de l'interface publique de la classe dérivée. Les membres publics hérités peuvent être appelés comme s'ils étaient définis dans la classe dérivée. Dans l'exemple suivant, la classe `A` définit une méthode nommée `Method1`, et la classe `B` hérite de la classe

`A`. L'exemple appelle ensuite `Method1` comme s'il s'agissait d'une méthode d'instance sur `B`.

```
C#

public class A
{
    public void Method1()
    {
        // Method implementation.
    }
}

public class B : A
{ }

public class Example
{
    public static void Main()
    {
        B b = new ();
        b.Method1();
    }
}
```

Les classes dérivées peuvent également *substituer* les membres hérités en fournissant une implémentation alternative. Pour être en mesure de substituer un membre, le membre de la classe de base doit être marqué avec le mot-clé virtual. Par défaut, les membres de classe de base ne sont pas marqués comme `virtual` et ne peut pas être substitués. Une tentative de substituer un membre non virtuel, comme dans l'exemple suivant, génère l'erreur de compilateur CS0506 : « <member> : impossible de substituer le membre hérité <member>, car il n'est pas marqué comme virtual, abstract ou override. »

```
C#

public class A
{
    public void Method1()
    {
        // Do something.
    }
}

public class B : A
{
    public override void Method1() // Generates CS0506.
    {
        // Do something else.
```

```
    }
}
```

Dans certains cas, une classe dérivée *doit* remplacer l'implémentation de la classe de base. Les membres de classe de base marqués avec le mot-clé abstract requièrent que les classes dérivées les remplacent. Compiler l'exemple suivant génère l'erreur de compilateur CS0534, « <class> does not implement inherited abstract member <member> », car la classe `B` ne fournit aucune implémentation de `A.Method1`.

C#

```
public abstract class A
{
    public abstract void Method1();
}

public class B : A // Generates CS0534.
{
    public void Method3()
    {
        // Do something.
    }
}
```

L'héritage s'applique uniquement aux classes et interfaces. Les autres catégories de type (structures, délégués et énumérations) ne permettent pas l'héritage. En raison de ces règles, la tentative de compilation du code comme dans l'exemple suivant génère l'erreur de compilateur CS0527 : « Le type « ValueType » dans la liste d'interface n'est pas une interface ». Le message d'erreur indique que, bien que vous puissiez définir les interfaces qu'un struct implémente, l'héritage n'est pas pris en charge.

C#

```
public struct ValueStructure : ValueType // Generates CS0527.
{
}
```

Héritage implicite

Outre les types qui peuvent hériter via l'héritage simple, tous les types dans le système de types de .NET héritent implicitement de Object ou d'un type dérivé. Les fonctionnalités communes de Object sont disponibles pour n'importe quel type.

Pour comprendre ce que l'héritage implicite signifie, nous allons définir une nouvelle classe, `SimpleClass`, qui est simplement une définition de classe vide :

C#

```
public class SimpleClass
{ }
```

Vous pouvez ensuite utiliser la réflexion (qui permet d'inspecter les métadonnées d'un type pour obtenir des informations sur ce type) pour obtenir la liste des membres qui appartiennent au type `SimpleClass`. Même si vous n'avez pas défini de membres dans votre classe `SimpleClass`, la sortie de l'exemple indique qu'il a en fait neuf membres. Un de ces membres est un constructeur sans paramètre (ou par défaut) qui est fourni automatiquement pour le type `SimpleClass` par le compilateur C#. Les huit restants sont membres de Object, le type à partir duquel toutes les classes et interfaces du système de type .NET héritent implicitement.

C#

```
using System.Reflection;

public class SimpleClassExample
{
    public static void Main()
    {
        Type t = typeof(SimpleClass);
        BindingFlags flags = BindingFlags.Instance | BindingFlags.Static |
BindingFlags.Public |
                             BindingFlags.NonPublic |
BindingFlags.FlattenHierarchy;
        MemberInfo[] members = t.GetMembers(flags);
        Console.WriteLine($"Type {t.Name} has {members.Length} members: ");
        foreach (MemberInfo member in members)
        {
            string access = "";
            string stat = "";
            var method = member as MethodBase;
            if (method != null)
            {
                if (method.IsPublic)
                    access = " Public";
                else if (method.IsPrivate)
                    access = " Private";
                else if (method.IsFamily)
                    access = " Protected";
                else if (method.IsAssembly)
                    access = " Internal";
                else if (method.IsFamilyOrAssembly)
                    access = " Protected Internal ";
```

```
                if (method.IsStatic)
                    stat = " Static";
            }
            string output = $"{member.Name} ({member.MemberType}): {access}
{stat}, Declared by {member.DeclaringType}";
            Console.WriteLine(output);
        }
    }
}
// The example displays the following output:
//  Type SimpleClass has 9 members:
//  ToString (Method):  Public, Declared by System.Object
//  Equals (Method):  Public, Declared by System.Object
//  Equals (Method):  Public Static, Declared by System.Object
//  ReferenceEquals (Method):  Public Static, Declared by System.Object
//  GetHashCode (Method):  Public, Declared by System.Object
//  GetType (Method):  Public, Declared by System.Object
//  Finalize (Method):  Internal, Declared by System.Object
//  MemberwiseClone (Method):  Internal, Declared by System.Object
//  .ctor (Constructor):  Public, Declared by SimpleClass
```

L'héritage implicite à partir de la classe Object rend ces méthodes disponibles pour la classe `SimpleClass` :

- La méthode public `ToString`, qui convertit un objet `SimpleClass` en sa représentation de chaîne, retourne le nom de type complet. Dans ce cas, la méthode `ToString` retourne la chaîne « SimpleClass ».

- Voici trois méthodes de test d'égalité de deux objets : la méthode public instance `Equals(Object)`, la méthode public static `Equals(Object, Object)` et la méthode public static `ReferenceEquals(Object, Object)`. Par défaut, ces méthodes testent l'égalité des références. Autrement dit, pour être égales, deux variables d'objet doivent faire référence au même objet.

- La méthode public `GetHashCode`, qui calcule une valeur qui permet à une instance du type d'être utilisée dans des collections hachées.

- La méthode `GetType` publique qui retourne un objet Type qui représente le type `SimpleClass`.

- La méthode protected Finalize, qui est conçue pour libérer les ressources non gérées avant que la mémoire d'un objet soit récupérée par le récupérateur de mémoire.

- La méthode protected MemberwiseClone, qui crée un clone partiel de l'objet actuel.

En raison de l'héritage implicite, vous pouvez appeler n'importe quel membre hérité d'un objet `SimpleClass` exactement comme s'il était en fait un membre défini dans la classe `SimpleClass`. Par exemple, l'exemple suivant appelle la méthode `SimpleClass.ToString`, dont `SimpleClass` hérite de Object.

C#

```
public class EmptyClass
{ }

public class ClassNameExample
{
    public static void Main()
    {
        EmptyClass sc = new();
        Console.WriteLine(sc.ToString());
    }
}
// The example displays the following output:
//        EmptyClass
```

Le tableau suivant répertorie les catégories de types que vous pouvez créer en C# et les types à partir desquels ils héritent implicitement. Chaque type de base apporte un autre ensemble de membres disponibles via l'héritage aux types dérivés implicitement.

Catégorie de type	Hérite implicitement de
class	Object
struct	ValueType, Object
enum	Enum, ValueType, Object
délégué	MulticastDelegate, Delegate, Object

L'héritage et une relation « est un »

En règle générale, l'héritage est utilisé pour exprimer une relation « est un » entre une classe de base et une ou plusieurs classes dérivées, où les classes dérivées sont des versions spécialisées de la classe de base ; la classe dérivée est un type de la classe de base. Par exemple, la classe `Publication` représente une publication de tout type et les classes `Book` et `Magazine` représentent les classes des types spécifiques de publications.

ⓘ Notes

Les classes et structures peuvent implémenter une ou plusieurs interfaces. Bien que l'implémentation d'interface est souvent présentée comme une solution de contournement pour l'héritage unique ou comme une façon d'utiliser l'héritage avec les structures, elle est conçue pour exprimer une autre relation (« peut faire ») entre une interface et son type d'implémentation que l'héritage. Une interface définit un sous-ensemble de fonctionnalités (comme la capacité à tester l'égalité, comparer ou trier des objets, ou pour prendre en charge la mise en forme et l'analyse dépendant de la culture) que l'interface met à disposition pour ses types d'implémentation.

Notez que « est un » exprime également la relation entre un type et une instanciation spécifique de ce type. Dans l'exemple suivant, `Automobile` est une classe qui possède trois propriétés en lecture seule uniques : `Make`, le fabricant de l'automobile ; `Model`, le type de voiture et `Year`, son année de fabrication. Votre classe `Automobile` comporte également un constructeur dont les arguments sont assignés aux valeurs de propriété, et elle remplace la méthode Object.ToString pour générer une chaîne qui identifie de façon unique l'instance `Automobile` plutôt que la classe `Automobile`.

C#

```
public class Automobile
{
    public Automobile(string make, string model, int year)
    {
        if (make == null)
            throw new ArgumentNullException(nameof(make), "The make cannot
be null.");
        else if (string.IsNullOrWhiteSpace(make))
            throw new ArgumentException("make cannot be an empty string or
have space characters only.");
        Make = make;

        if (model == null)
            throw new ArgumentNullException(nameof(model), "The model cannot
be null.");
        else if (string.IsNullOrWhiteSpace(model))
            throw new ArgumentException("model cannot be an empty string or
have space characters only.");
        Model = model;

        if (year < 1857 || year > DateTime.Now.Year + 2)
            throw new ArgumentException("The year is out of range.");
        Year = year;
    }

    public string Make { get; }

    public string Model { get; }
```

```
    public int Year { get; }

    public override string ToString() => $"{Year} {Make} {Model}";
}
```

Dans ce cas, vous ne devriez pas vous reposer sur l'héritage pour représenter les modèles et constructeurs spécifiques. Par exemple, il est inutile de définir un type `Packard` pour représenter les véhicules automobiles fabriqués par la société Packard Motor Car. Au lieu de cela, vous pouvez les représenter en créant un objet `Automobile` avec les valeurs appropriées passées à son constructeur de classe, comme dans l'exemple suivant.

C#

```
using System;

public class Example
{
    public static void Main()
    {
        var packard = new Automobile("Packard", "Custom Eight", 1948);
        Console.WriteLine(packard);
    }
}
// The example displays the following output:
//       1948 Packard Custom Eight
```

Une relation « est un » basée sur l'héritage est préférablement appliquée à une classe de base et aux classes dérivées qui ajoutent des membres supplémentaires à la classe de base ou qui nécessitent des fonctionnalités supplémentaires non présentes dans la classe de base.

Conception de la classe de base et des classes dérivées

Examinons le processus de conception d'une classe de base et de ses classes dérivées. Dans cette section, vous allez définir une classe de base, `Publication`, qui représente une publication de tout type, comme un livre, un magazine, un journal, une revue, un article, etc. Vous allez également définir une classe `Book` dérivée de `Publication`. Vous pourriez facilement étendre l'exemple pour définir d'autres classes dérivées, comme `Magazine`, `Journal`, `Newspaper` et `Article`.

Classe Publication de base

Lors de la conception de votre classe `Publication`, vous devez prendre plusieurs décisions de conception :

- Les membres à inclure dans votre classe de base `Publication` et si les membres `Publication` fournissent des implémentations de méthode, ou si `Publication` est une classe de base abstraite qui sert de modèle pour ses classes dérivées.

  Dans ce cas, la classe `Publication` fournit des implémentations de méthode. La section Conception de classes de base abstraites et leurs classes dérivées contient un exemple qui utilise une classe de base abstraite pour définir les méthodes que les classes dérivées doivent substituer. Les classes dérivées sont libres de fournir une implémentation qui convient pour le type dérivé.

  La possibilité de réutiliser le code (autrement dit, plusieurs classes dérivées partagent la déclaration et l'implémentation de méthodes de classe de base et n'ont pas besoin de les substituer) constitue un avantage des classes de base non abstraites. Par conséquent, vous devez ajouter des membres à `Publication` si leur code est susceptible d'être partagé par certains ou la majorité des types `Publication` spécialisés. Si vous ne fournissez pas efficacement les implémentations de classe de base, vous devrez fournir des implémentations de membres en grande partie identiques dans les classes dérivées au lieu d'une implémentation unique dans la classe de base. La nécessité de maintenir le code dupliqué à plusieurs emplacements est une source potentielle de bogues.

  Pour optimiser la réutilisation du code et créer une hiérarchie d'héritage logique et intuitive, vous voulez vous assurer d'ajouter à la classe `Publication` uniquement les données et fonctionnalités communes à toutes les publications ou la plupart d'entre elles. Les classes dérivées implémentent ensuite les membres qui sont uniques pour les types particuliers de publications qu'ils représentent.

- La mesure dans laquelle étendre votre hiérarchie de classes. Souhaitez-vous développer une hiérarchie de trois classes ou plus, plutôt que simplement une classe de base et une ou plusieurs classes dérivées ? Par exemple, `Publication` peut être une classe de base de `Periodical`, qui est elle-même une classe de base de `Magazine`, `Journal` et `Newspaper`.

  Dans votre exemple, vous allez utiliser la hiérarchie simple d'une classe `Publication` et d'une classe dérivée unique, `Book`. Vous pourriez facilement étendre l'exemple pour créer un certain nombre d'autres classes qui dérivent de `Publication`, comme `Magazine` et `Article`.

- S'il est judicieux d'instancier la classe de base. Si ce n'est pas le cas, vous devez appliquer le mot-clé abstract à la classe. Dans le cas contraire, votre classe `Publication` peut être instanciée en appelant son constructeur de classe. Si une tentative est effectuée pour instancier une classe marquée avec le mot clé `abstract` par un appel direct à son constructeur de classe, le compilateur C# génère l'erreur CS0144 « Impossible de créer une instance de l'interface ou de la classe abstraite ». Si une tentative d'instanciation de la classe à l'aide de la réflexion est effectuée, la méthode de réflexion lève une exception MemberAccessException.

  Par défaut, une classe de base peut être instanciée en appelant son constructeur de classe. Vous n'avez pas à définir explicitement un constructeur de classe. S'il n'y en a aucun présent dans le code source de la classe de base, le compilateur C# fournit automatiquement un constructeur par défaut (sans paramètre).

  Dans votre exemple, vous marquez la classe `Publication` comme abstract afin qu'elle ne puisse pas être instanciée. Une classe `abstract` sans méthode `abstract` indique que cette classe représente un concept abstrait qui est partagé entre plusieurs classes concrètes (comme `Book`, `Journal`).

- Si les classes dérivées doivent hériter de l'implémentation de classe de base de membres particuliers, si elles ont l'option de substituer l'implémentation de la classe de base ou si elles doivent fournir une implémentation. Vous utilisez le mot clé abstract pour forcer les classes dérivées à fournir une implémentation. Vous devez utiliser le mot clé virtual pour permettre aux classes dérivées de substituer une méthode de classe de base. Par défaut, les méthodes définies dans la classe de base ne sont *pas* substituables.

  La classe `Publication` n'a aucune méthode `abstract`, mais la classe elle-même est `abstract`.

- Si une classe dérivée représente la classe finale dans la hiérarchie d'héritage et ne peut pas elle-même être utilisée comme classe de base pour les classes dérivées supplémentaires. Par défaut, toute classe peut servir de classe de base. Vous pouvez appliquer le mot clé sealed pour indiquer qu'une classe ne peut pas servir de classe de base pour des classes supplémentaires. La tentative de dériver à partir d'une classe sealed a généré l'erreur de compilateur CS0509, « impossible de dériver à partir du type sealed <typeName> ».

  Dans votre exemple, vous allez marquer votre classe dérivée en tant que `sealed`.

L'exemple suivant montre le code source pour la classe `Publication` ainsi qu'une énumération `PublicationType` retournée par la propriété `Publication.PublicationType`.

Outre les membres qu'elle hérite de Object, la classe `Publication` définit les membres uniques et substitutions de membres suivants :

C#

```
public enum PublicationType { Misc, Book, Magazine, Article };

public abstract class Publication
{
    private bool _published = false;
    private DateTime _datePublished;
    private int _totalPages;

    public Publication(string title, string publisher, PublicationType type)
    {
        if (string.IsNullOrWhiteSpace(publisher))
            throw new ArgumentException("The publisher is required.");
        Publisher = publisher;

        if (string.IsNullOrWhiteSpace(title))
            throw new ArgumentException("The title is required.");
        Title = title;

        Type = type;
    }

    public string Publisher { get; }

    public string Title { get; }

    public PublicationType Type { get; }

    public string? CopyrightName { get; private set; }

    public int CopyrightDate { get; private set; }

    public int Pages
    {
        get { return _totalPages; }
        set
        {
            if (value <= 0)
                throw new ArgumentOutOfRangeException(nameof(value), "The
number of pages cannot be zero or negative.");
            _totalPages = value;
        }
    }

    public string GetPublicationDate()
    {
        if (!_published)
            return "NYP";
        else
```

```
            return _datePublished.ToString("d");
    }

    public void Publish(DateTime datePublished)
    {
        _published = true;
        _datePublished = datePublished;
    }

    public void Copyright(string copyrightName, int copyrightDate)
    {
        if (string.IsNullOrWhiteSpace(copyrightName))
            throw new ArgumentException("The name of the copyright holder is
required.");
        CopyrightName = copyrightName;

        int currentYear = DateTime.Now.Year;
        if (copyrightDate < currentYear - 10 || copyrightDate > currentYear
+ 2)
            throw new ArgumentOutOfRangeException($"The copyright year must
be between {currentYear - 10} and {currentYear + 1}");
        CopyrightDate = copyrightDate;
    }

    public override string ToString() => Title;
}
```

- Un constructeur

  Étant donné que la classe `Publication` est `abstract`, elle ne peut pas être instanciée directement à partir du code comme dans l'exemple suivant :

  C#

  ```
  var publication = new Publication("Tiddlywinks for Experts", "Fun and
  Games",
                                    PublicationType.Book);
  ```

  Toutefois, son constructeur d'instance peut être appelé directement à partir des constructeurs de classes dérivées, comme le code source pour la classe `Book` le montre.

- Deux propriétés liées à la publication

  `Title` est une propriété String en lecture seule dont la valeur est fournie en appelant le constructeur `Publication`.

  `Pages` est une propriété Int32 en lecture seule qui indique le nombre total de pages de la publication. La valeur est stockée dans un champ privé nommé

`totalPages`. Elle doit être positive, sans quoi une exception ArgumentOutOfRangeException est levée.

- Membres liés à l'éditeur

  Deux propriétés en lecture seule, `Publisher` et `Type`. Les valeurs sont à l'origine fournies par l'appel au constructeur de la classe `Publication`.

- Membres liés à la publication

  Deux méthodes, `Publish` et `GetPublicationDate` définissent et retournent la date de publication. La méthode `Publish` définit l'indicateur privé `published` sur `true` lorsqu'elle est appelée et affecte la date passée comme argument au champ privé `datePublished`. La méthode `GetPublicationDate` retourne la chaîne « NYP » si l'indicateur `published` est `false`, et la valeur du champ `datePublished` si l'indicateur est `true`.

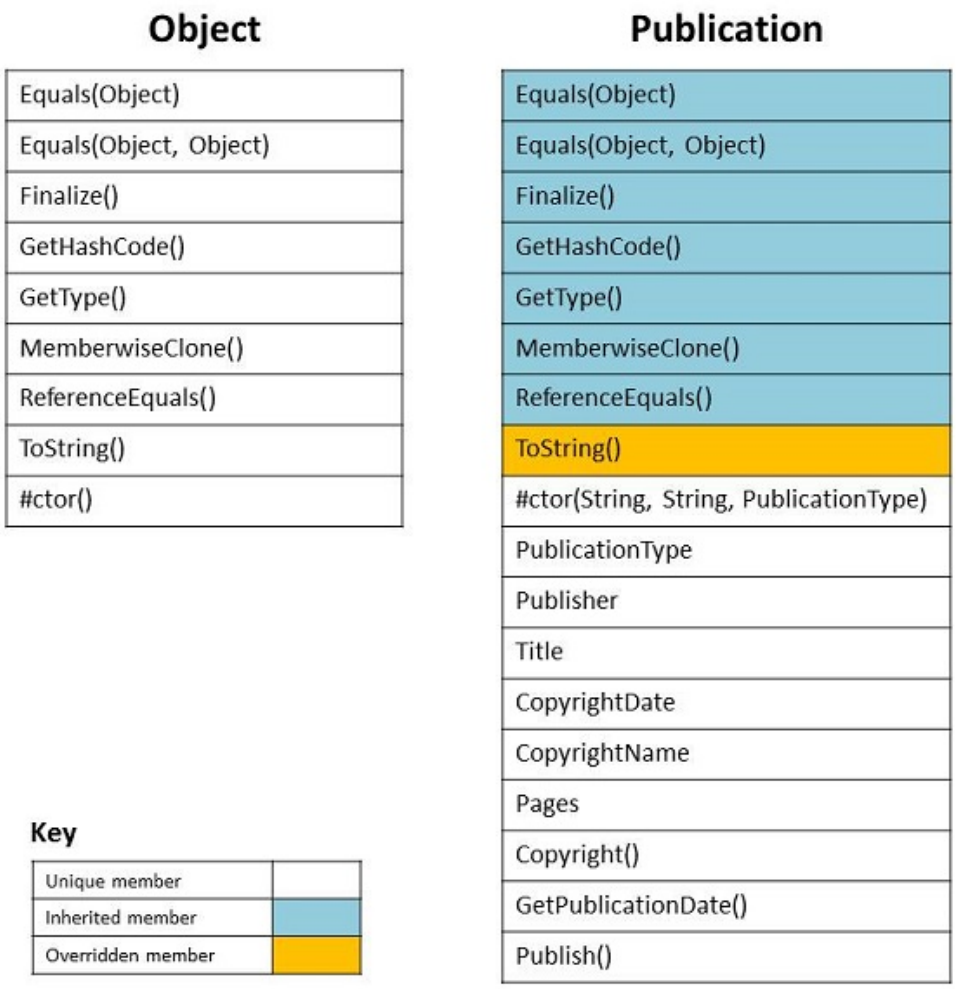
- Membres liés aux droits d'auteur

  La méthode `Copyright` prend comme arguments le nom du titulaire des droits d'auteur et l'année des droits d'auteur et les attribue aux propriétés `CopyrightName` et `CopyrightDate`.

- Une substitution de la méthode `ToString`

  Si un type ne remplace pas la méthode Object.ToString, il retourne le nom qualifié complet du type, ce qui n'aide pas vraiment à faire la différence entre une instance et une autre. La classe `Publication` substitue Object.ToString pour retourner la valeur de la propriété `Title`.

Le schéma suivant illustre la relation entre votre classe `Publication` de base et sa classe Object implicitement héritée.



La classe `Book`

La classe `Book` représente un livre sous la forme d'un type spécialisé de publication. L'exemple suivant montre le code source pour la classe `Book`.

C#

```
using System;

public sealed class Book : Publication
{
    public Book(string title, string author, string publisher) :
           this(title, string.Empty, author, publisher)
    { }

    public Book(string title, string isbn, string author, string publisher)
: base(title, publisher, PublicationType.Book)
    {
        // isbn argument must be a 10- or 13-character numeric string
without "-" characters.
        // We could also determine whether the ISBN is valid by comparing
its checksum digit
        // with a computed checksum.
```

```
        //
        if (!string.IsNullOrEmpty(isbn))
        {
            // Determine if ISBN length is correct.
            if (!(isbn.Length == 10 | isbn.Length == 13))
                throw new ArgumentException("The ISBN must be a 10- or 13-
character numeric string.");
            if (!ulong.TryParse(isbn, out _))
                throw new ArgumentException("The ISBN can consist of numeric
characters only.");
        }
        ISBN = isbn;

        Author = author;
    }

    public string ISBN { get; }

    public string Author { get; }

    public decimal Price { get; private set; }

    // A three-digit ISO currency symbol.
    public string? Currency { get; private set; }

    // Returns the old price, and sets a new price.
    public decimal SetPrice(decimal price, string currency)
    {
        if (price < 0)
            throw new ArgumentOutOfRangeException(nameof(price), "The price
cannot be negative.");
        decimal oldValue = Price;
        Price = price;

        if (currency.Length != 3)
            throw new ArgumentException("The ISO currency symbol is a 3-
character string.");
        Currency = currency;

        return oldValue;
    }

    public override bool Equals(object? obj)
    {
        if (obj is not Book book)
            return false;
        else
            return ISBN == book.ISBN;
    }

    public override int GetHashCode() => ISBN.GetHashCode();

    public override string ToString() => $"{(string.IsNullOrEmpty(Author) ?
"" : Author + ", ")}{Title}";
}
```

Outre les membres qu'elle hérite de `Publication`, la classe `Book` définit les membres uniques et substitutions de membres suivants :

- Deux constructeurs

  Les deux constructeurs `Book` partagent trois paramètres communs. Deux d'entre eux, *title* et *publisher*, correspondent aux paramètres du constructeur `Publication`. Le troisième est *author*, qui est stocké dans une propriété `Author` non modifiable publique. Un constructeur inclut un paramètre *isbn*, qui est stocké dans l'auto-propriété `ISBN`.

  Le premier constructeur utilise le mot-clé this pour appeler l'autre constructeur. Le chaînage de constructeurs est un modèle courant pour la définition de constructeurs. Les constructeurs avec le moins de paramètres fournissent les valeurs par défaut au moment de l'appel du constructeur avec le plus grand nombre de paramètres.

  Le deuxième constructeur utilise le mot-clé base pour transmettre le titre et le nom de l'éditeur au constructeur de classe de base. Si vous n'effectuez pas un appel explicite à un constructeur de classe de base dans votre code source, le compilateur C# fournit automatiquement un appel au constructeur par défaut ou sans paramètres de la classe de base.

- Une propriété `ISBN` en lecture seule qui retourne le numéro ISBN de l'objet `Book`, un numéro unique à 10 ou 13 chiffres. Le numéro ISBN est fourni en tant qu'argument à un des constructeurs `Book`. Le numéro ISBN est stocké dans un champ de stockage privé qui est généré automatiquement par le compilateur.

- Une propriété `Author` en lecture seule. Le nom de l'auteur est fourni en tant qu'argument aux deux constructeurs `Book` et est stocké dans la propriété.

- Deux propriétés en lecture seule relatives au prix, `Price` et `Currency`. Leurs valeurs sont fournies comme arguments dans un appel de méthode `SetPrice`. La propriété `Currency` est le symbole de devise ISO à trois caractères (par exemple, USD pour le dollar américain). Les symboles de devise ISO peuvent être récupérés à partir de la propriété ISOCurrencySymbol. Ces deux propriétés sont en lecture seule en externe, mais peuvent être définies par du code dans la classe `Book`.

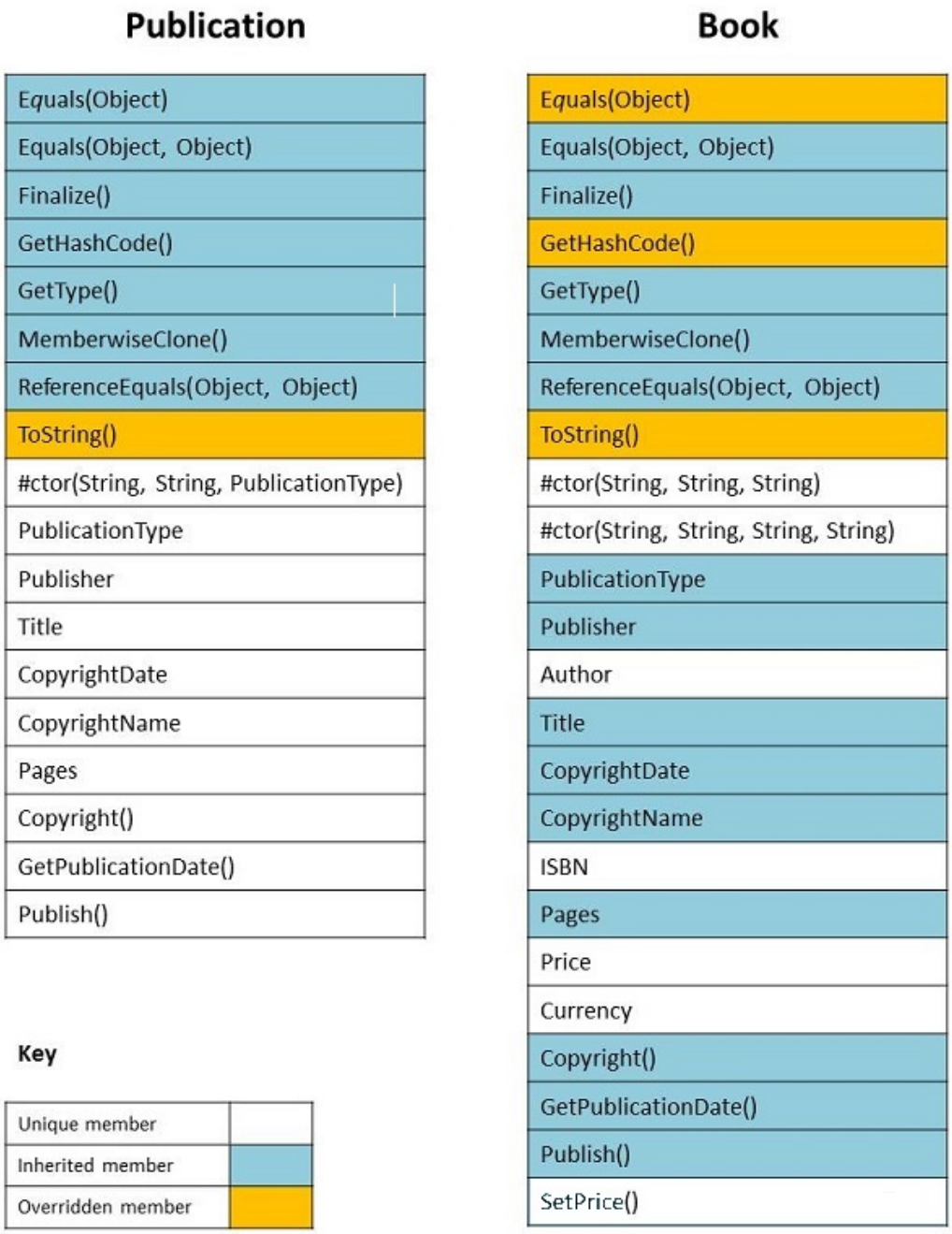
- Une méthode `SetPrice` qui définit les valeurs des propriétés `Price` et `Currency`. Ces valeurs sont retournées par ces mêmes propriétés.

- Se substitue à la méthode `ToString` (héritée de `Publication`) et aux méthodes Object.Equals(Object) et GetHashCode (héritées de Object).

  Sauf si elle est substituée, la méthode Object.Equals(Object) teste l'égalité des références. Autrement dit, deux variables d'objet sont considérées comme égales si elles font référence au même objet. Dans la classe `Book`, en revanche, deux objets `Book` doivent être égaux s'ils ont le même ISBN.

  Lorsque vous substituez la méthode Object.Equals(Object), vous devez également substituer la méthode GetHashCode qui retourne une valeur que le runtime utilise pour stocker les éléments dans les collections hachées pour une récupération efficace. Le code de hachage doit retourner une valeur qui est cohérente avec le test d'égalité. Étant donné que vous avez substitué Object.Equals(Object) pour retourner `true` si les propriétés ISBN de deux objets `Book` sont égales, vous retournez le code de hachage calculé en appelant la méthode GetHashCode de la chaîne retournée par la propriété `ISBN`.

Le schéma suivant illustre la relation entre la base la classe `Book` et `Publication`, sa classe de base.



Vous pouvez maintenant instancier un objet `Book`, appeler ses membres uniques et hérités le passer en tant qu'argument pour une méthode qui attend un paramètre de type `Publication` ou de type `Book`, comme illustré dans l'exemple suivant.

C#

```
public class ClassExample
{
    public static void Main()
    {
        var book = new Book("The Tempest", "0971655819", "Shakespeare,
William",
```

```
                                "Public Domain Press");
        ShowPublicationInfo(book);
        book.Publish(new DateTime(2016, 8, 18));
        ShowPublicationInfo(book);

        var book2 = new Book("The Tempest", "Classic Works Press",
"Shakespeare, William");
        Console.Write($"{book.Title} and {book2.Title} are the same
publication: " +
                $"{((Publication)book).Equals(book2)}");
    }

    public static void ShowPublicationInfo(Publication pub)
    {
        string pubDate = pub.GetPublicationDate();
        Console.WriteLine($"{pub.Title}, " +
                  $"{(pubDate == "NYP" ? "Not Yet Published" : "published on
" + pubDate):d} by {pub.Publisher}");
    }
}
// The example displays the following output:
//       The Tempest, Not Yet Published by Public Domain Press
//       The Tempest, published on 8/18/2016 by Public Domain Press
//       The Tempest and The Tempest are the same publication: False
```

Conception de classes de base abstraites et de leurs classes dérivées

Dans l'exemple précédent, vous avez défini une classe de base qui a fourni une implémentation d'un certain nombre de méthodes pour permettre aux classes dérivées de partager du code. Dans de nombreux cas, toutefois, la classe de base n'est pas censée fournir une implémentation. Au lieu de cela, la classe de base est une *classe abstraite* qui déclare des *méthodes abstraites* ; elle sert de modèle qui définit les membres que chaque classe dérivée doit implémenter. En général, dans une classe de base abstraite, l'implémentation de chaque type dérivé est unique pour ce type. Vous avez marqué la classe avec le mot clé abstract, car il n'était pas judicieux d'instancier un objet `Publication`, même si la classe fournissait des implémentations de fonctionnalités communes aux publications.

Par exemple, chaque forme géométrique bidimensionnelle fermée inclut deux propriétés : l'aire, l'étendue interne de la forme, et le périmètre, ou la distance le long des bords de la forme. La façon de laquelle ces propriétés sont calculées, cependant, dépend entièrement de la forme spécifique. La formule pour calculer le périmètre (ou la circonférence) d'un cercle, par exemple, est différente de celle d'un carré. La classe `Shape` est une classe `abstract` avec des méthodes `abstract`. Ceci indique que les

classes dérivées partagent les mêmes fonctionnalités, mais ces classes dérivées implémentent ces fonctionnalités différemment.

L'exemple suivant définit une classe de base abstraite nommée `Shape` qui définit deux propriétés : `Area` et `Perimeter`. En plus de marquer la classe avec le mot clé abstract, chaque membre de l'instance est également marqué avec le mot clé abstract. Dans ce cas, `Shape` substitue également la méthode Object.ToString pour renvoyer le nom du type, plutôt que son nom qualifié complet. Elle définit aussi deux membres statiques, `GetArea` et `GetPerimeter`, qui permettent aux appelants de récupérer facilement l'aire et le périmètre d'une instance de toute classe dérivée. Lorsque vous passez une instance d'une classe dérivée à l'une de ces méthodes, le runtime appelle la substitution de la méthode de la classe dérivée.

C#

```
public abstract class Shape
{
    public abstract double Area { get; }

    public abstract double Perimeter { get; }

    public override string ToString() => GetType().Name;

    public static double GetArea(Shape shape) => shape.Area;

    public static double GetPerimeter(Shape shape) => shape.Perimeter;
}
```

Vous pouvez ensuite dériver des classes qui représentent des formes spécifiques à partir de `Shape`. L'exemple suivant définit trois classes : `Square`, `Rectangle` et `Circle`. Chaque forme utilise une formule unique pour calculer l'aire et périmètre. Certaines des classes dérivées définissent également des propriétés, telles que `Rectangle.Diagonal` et `Circle.Diameter`, qui sont propres à la forme qu'ils représentent.

C#

```
using System;

public class Square : Shape
{
    public Square(double length)
    {
        Side = length;
    }

    public double Side { get; }
```

```
    public override double Area => Math.Pow(Side, 2);

    public override double Perimeter => Side * 4;

    public double Diagonal => Math.Round(Math.Sqrt(2) * Side, 2);
}

public class Rectangle : Shape
{
    public Rectangle(double length, double width)
    {
        Length = length;
        Width = width;
    }

    public double Length { get; }

    public double Width { get; }

    public override double Area => Length * Width;

    public override double Perimeter => 2 * Length + 2 * Width;

    public bool IsSquare() => Length == Width;

    public double Diagonal => Math.Round(Math.Sqrt(Math.Pow(Length, 2) +
Math.Pow(Width, 2)), 2);
}

public class Circle : Shape
{
    public Circle(double radius)
    {
        Radius = radius;
    }

    public override double Area => Math.Round(Math.PI * Math.Pow(Radius, 2),
2);

    public override double Perimeter => Math.Round(Math.PI * 2 * Radius, 2);

    // Define a circumference, since it's the more familiar term.
    public double Circumference => Perimeter;

    public double Radius { get; }

    public double Diameter => Radius * 2;
}
```

L'exemple suivant utilise les objets dérivés de `Shape`. Elle instancie un tableau d'objets dérivés de `Shape` et appelle les méthodes statiques de la classe `Shape` qui encapsule les valeurs de propriété de retour de `Shape`. Le runtime récupère les valeurs de propriétés

substituées des types dérivés. L'exemple convertit également chaque objet `Shape` dans le tableau en son type dérivé et, si la conversion réussit, récupère les propriétés de cette sous-classe particulière de `Shape`.

C#

```
using System;

public class Example
{
    public static void Main()
    {
        Shape[] shapes = { new Rectangle(10, 12), new Square(5),
                           new Circle(3) };
        foreach (Shape shape in shapes)
        {
            Console.WriteLine($"{shape}: area, {Shape.GetArea(shape)}; " +
                              $"perimeter, {Shape.GetPerimeter(shape)}");
            if (shape is Rectangle rect)
            {
                Console.WriteLine($"   Is Square: {rect.IsSquare()},
Diagonal: {rect.Diagonal}");
                continue;
            }
            if (shape is Square sq)
            {
                Console.WriteLine($"   Diagonal: {sq.Diagonal}");
                continue;
            }
        }
    }
}
// The example displays the following output:
//         Rectangle: area, 120; perimeter, 44
//            Is Square: False, Diagonal: 15.62
//         Square: area, 25; perimeter, 20
//            Diagonal: 7.07
//         Circle: area, 28.27; perimeter, 18.85
```

Guide pratique pour caster de manière sécurisée avec des critères spéciaux, ainsi que les opérateurs is et as

Article • 07/04/2023

Dans la mesure où les objets sont polymorphes, une variable d'un type de classe de base peut contenir un type dérivé. Pour accéder aux membres d'instance du type dérivé, il est nécessaire de réeffectuer un cast de la valeur vers le type dérivé. Toutefois, un cast risque de lever InvalidCastException. C# fournit des instructions relatives aux critères spéciaux, qui effectuent un cast de manière conditionnelle, uniquement en cas de réussite. C# fournit également les opérateurs is et as pour tester l'appartenance d'une valeur à un certain type.

L'exemple suivant montre comment utiliser l'instruction `is` des critères spéciaux :

C#

```
var g = new Giraffe();
var a = new Animal();
FeedMammals(g);
FeedMammals(a);
// Output:
// Eating.
// Animal is not a Mammal

SuperNova sn = new SuperNova();
TestForMammals(g);
TestForMammals(sn);

static void FeedMammals(Animal a)
{
    if (a is Mammal m)
    {
        m.Eat();
    }
    else
    {
        // variable 'm' is not in scope here, and can't be used.
        Console.WriteLine($"{a.GetType().Name} is not a Mammal");
    }
}

static void TestForMammals(object o)
{
    // You also can use the as operator and test for null
    // before referencing the variable.
    var m = o as Mammal;
```

```
    if (m != null)
    {
        Console.WriteLine(m.ToString());
    }
    else
    {
        Console.WriteLine($"{o.GetType().Name} is not a Mammal");
    }
}
// Output:
// I am an animal.
// SuperNova is not a Mammal

class Animal
{
    public void Eat() { Console.WriteLine("Eating."); }
    public override string ToString()
    {
        return "I am an animal.";
    }
}
class Mammal : Animal { }
class Giraffe : Mammal { }

class SuperNova { }
```

L'exemple précédent illustre quelques fonctionnalités de la syntaxe relative aux critères spéciaux. L'instruction `if (a is Mammal m)` combine le test à une assignation d'initialisation. L'assignation n'a lieu qu'en cas de réussite du test. La variable `m` se trouve uniquement dans l'étendue de l'instruction `if` incorporée, à laquelle elle a été assignée. Vous ne pouvez pas accéder à `m` plus tard dans la même méthode. L'exemple précédent montre également comment utiliser l'opérateur as pour convertir un objet en un type spécifié.

Vous pouvez également utiliser la même syntaxe afin de tester l'existence si un type valeur pouvant accepter la valeur Null a une valeur, comme indiqué dans l'exemple de code suivant :

C#

```
int i = 5;
PatternMatchingNullable(i);

int? j = null;
PatternMatchingNullable(j);

double d = 9.78654;
PatternMatchingNullable(d);

PatternMatchingSwitch(i);
```

```
PatternMatchingSwitch(j);
PatternMatchingSwitch(d);

static void PatternMatchingNullable(ValueType? val)
{
    if (val is int j) // Nullable types are not allowed in patterns
    {
        Console.WriteLine(j);
    }
    else if (val is null) // If val is a nullable type with no value, this
expression is true
    {
        Console.WriteLine("val is a nullable type with the null value");
    }
    else
    {
        Console.WriteLine("Could not convert " + val.ToString());
    }
}

static void PatternMatchingSwitch(ValueType? val)
{
    switch (val)
    {
        case int number:
            Console.WriteLine(number);
            break;
        case long number:
            Console.WriteLine(number);
            break;
        case decimal number:
            Console.WriteLine(number);
            break;
        case float number:
            Console.WriteLine(number);
            break;
        case double number:
            Console.WriteLine(number);
            break;
        case null:
            Console.WriteLine("val is a nullable type with the null value");
            break;
        default:
            Console.WriteLine("Could not convert " + val.ToString());
            break;
    }
}
```

L'exemple précédent illustre d'autres fonctionnalités relatives aux critères spéciaux pour les conversions. Vous pouvez tester le modèle null d'une variable en vérifiant spécifiquement la valeur `null`. Quand la valeur runtime de la variable est `null`, une instruction `is` de vérification de type retourne toujours `false`. L'instruction `is` relative

aux critères spéciaux n'autorise aucun type valeur Nullable, par exemple `int?` ou `Nullable<int>`. Toutefois, vous pouvez tester un autre type valeur. Les modèles `is` de l'exemple précédent ne sont pas limités aux types valeur pouvant accepter la valeur Null. Vous pouvez également utiliser ces modèles pour tester si une variable d'un type référence a une valeur ou s'il s'agit de `null`.

L'exemple précédent montre également comment utiliser le type de modèle dans une instruction `switch` où la variable peut avoir plusieurs types différents.

Pour déterminer si une variable a un type donné, mais sans l'assigner pour autant à une nouvelle variable, vous pouvez utiliser les opérateurs `is` et `as` pour les types référence et les types valeur pouvant accepter la valeur Null. Le code suivant montre comment utiliser les instructions `is` et `as`, qui faisaient partie du langage C# avant l'introduction des critères spéciaux permettant de déterminer l'existence ou non d'un type donné pour une variable :

C#

```
// Use the is operator to verify the type.
// before performing a cast.
Giraffe g = new();
UseIsOperator(g);

// Use the as operator and test for null
// before referencing the variable.
UseAsOperator(g);

// Use pattern matching to test for null
// before referencing the variable
UsePatternMatchingIs(g);

// Use the as operator to test
// an incompatible type.
SuperNova sn = new();
UseAsOperator(sn);

// Use the as operator with a value type.
// Note the implicit conversion to int? in
// the method body.
int i = 5;
UseAsWithNullable(i);

double d = 9.78654;
UseAsWithNullable(d);

static void UseIsOperator(Animal a)
{
    if (a is Mammal)
    {
```

```
        Mammal m = (Mammal)a;
        m.Eat();
    }
}

static void UsePatternMatchingIs(Animal a)
{
    if (a is Mammal m)
    {
        m.Eat();
    }
}

static void UseAsOperator(object o)
{
    Mammal? m = o as Mammal;
    if (m is not null)
    {
        Console.WriteLine(m.ToString());
    }
    else
    {
        Console.WriteLine($"{o.GetType().Name} is not a Mammal");
    }
}

static void UseAsWithNullable(System.ValueType val)
{
    int? j = val as int?;
    if (j is not null)
    {
        Console.WriteLine(j);
    }
    else
    {
        Console.WriteLine("Could not convert " + val.ToString());
    }
}
class Animal
{
    public void Eat() => Console.WriteLine("Eating.");
    public override string ToString() => "I am an animal.";
}
class Mammal : Animal { }
class Giraffe : Mammal { }

class SuperNova { }
```

Comme vous pouvez le constater en comparant ce code avec le code utilisant les critères spéciaux, la syntaxe des critères spéciaux fournit des fonctionnalités plus robustes, car elle combine le test et l'assignation dans une seule instruction. Utilisez la syntaxe des critères spéciaux autant que possible.

Tutoriel : Utiliser les critères spéciaux pour générer des algorithmes pilotés par type et pilotés par les données

Article • 10/05/2023

Il est possible d'écrire des fonctionnalités qui se comportent comme si des types provenant potentiellement d'autres bibliothèques avaient été étendus. Une autre utilisation des modèles consiste à créer des fonctionnalités requises par une application qui ne sont pas essentielles pour le type étendu.

Ce didacticiel vous montre comment effectuer les opérations suivantes :

- ✓ Reconnaître les situations dans lesquelles les critères spéciaux sont nécessaires.
- ✓ Utiliser des expressions de critères spéciaux pour implémenter des comportements en fonction des types et des valeurs de propriété.
- ✓ Combiner des critères spéciaux avec d'autres techniques pour créer des algorithmes complets.

Prérequis

- Nous vous recommandons Visual Studio pour Windows ou Mac. Vous pouvez télécharger une version gratuite à partir de la page téléchargements de Visual Studio. Visual Studio inclut le Kit de développement logiciel (SDK) .NET.
- Vous pouvez également utiliser l'éditeur Visual Studio Code. Vous devez installer le dernier SDK .NET séparément.
- Si vous préférez un autre éditeur, vous devez installer le dernier Kit de développement logiciel (SDK) .NET.

Ce tutoriel suppose de connaître C# et .NET, y compris Visual Studio ou l'interface CLI .NET.

Scénarios des critères spéciaux

Le développement moderne passe souvent par l'intégration de données provenant de plusieurs sources et par la présentation d'informations et d'insights tirés de ces données dans une seule et même application. Ni vous ni votre équipe n'aurez accès aux types qui représentent les données entrantes ou ne pourrez les contrôler.

Une conception classique orientée objet exigerait de créer dans l'application des types représentant chacun des types de données de ces multiples sources. L'application s'appuierait alors sur ces nouveaux types pour élaborer des hiérarchies d'héritage, créer des méthodes virtuelles et implémenter des abstractions. Ces techniques fonctionnent et représentent parfois les meilleurs outils disponibles. Il est possible dans d'autres cas d'écrire du code moins long et plus clair, à l'aide de techniques qui séparent les données des opérations qui les manipulent.

Ce tutoriel vise à créer et à explorer une application qui prend en entrée des données provenant de plusieurs sources externes pour un seul scénario. Nous verrons en quoi les **critères spéciaux** constituent un moyen efficace de consommer et de traiter ces données d'une autre manière que ce que prévoit le système d'origine.

Prenons une grande zone métropolitaine qui utilise des péages et des tarifs heures creuses/heures de pointe pour réguler le trafic. Nous allons écrire une application qui calcule le péage en fonction du type de véhicule. Lors d'améliorations ultérieures, nous intégrerons aux tarifs le nombre de passagers du véhicule ainsi que l'heure et le jour de la semaine.

À partir de cette brève description, vous avez peut-être rapidement esquissé une hiérarchie d'objets pour modéliser ce système. Cependant, les données proviennent de plusieurs sources, et notamment d'autres systèmes de gestion de l'immatriculation des véhicules. Ils fournissent des classes différentes pour modéliser ces données, sans proposer de modèle objet unique. Dans ce tutoriel, nous allons utiliser ces classes simplifiées pour modéliser les données sur les véhicules issues de ces systèmes externes, comme dans le code suivant :

C#

```
namespace ConsumerVehicleRegistration
{
    public class Car
    {
        public int Passengers { get; set; }
    }
}

namespace CommercialRegistration
{
    public class DeliveryTruck
    {
        public int GrossWeightClass { get; set; }
    }
}

namespace LiveryRegistration
{
```

```
    public class Taxi
    {
        public int Fares { get; set; }
    }

    public class Bus
    {
        public int Capacity { get; set; }
        public int Riders { get; set; }
    }
}
```

Vous pouvez télécharger l'exemple de démarrage à partir du référentiel GitHub dotnet/samples. Comme on peut le voir, les classes de véhicules proviennent de différents systèmes et utilisent différents espaces de noms. Aucune classe de base commune, autre que `System.Object`, n'est exploitable.

Conception avec les critères spéciaux

Le scénario utilisé dans ce tutoriel met en évidence les types de problèmes que les critères spéciaux peuvent parfaitement résoudre :

- Les objets dont vous avez besoin ne figurent pas dans une hiérarchie d'objets correspondant à vos objectifs. Vous utilisez peut-être des classes faisant partie de systèmes non liés les uns aux autres.
- Les fonctionnalités ajoutées ne font pas partie de l'abstraction de base de ces classes. Le péage payé *change* selon les types de véhicules, mais il ne s'agit pas d'une fonction essentielle du véhicule.

Lorsque la *forme* des données et les *opérations* effectuées sur celles-ci ne sont pas décrites ensemble, les fonctionnalités de critères spéciaux de C# facilitent la tâche.

Implémenter les calculs de péage de base

Le calcul de péage le plus élémentaire s'appuie uniquement sur le type de véhicule :

- `Car` : 2,00 $.
- `Taxi` : 3,50 $.
- `Bus` : 5,00 $.
- `DeliveryTruck` : 10,00 $.

Créez une classe `TollCalculator` et implémentez les critères spéciaux sur le type de véhicule pour obtenir le montant du péage. Le code suivant montre l'implémentation

initiale de `TollCalculator`.

C#

```
using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;

namespace Calculators;

public class TollCalculator
{
    public decimal CalculateToll(object vehicle) =>
        vehicle switch
    {
        Car c           => 2.00m,
        Taxi t          => 3.50m,
        Bus b           => 5.00m,
        DeliveryTruck t => 10.00m,
        { }             => throw new ArgumentException(message: "Not a known
vehicle type", paramName: nameof(vehicle)),
        null            => throw new ArgumentNullException(nameof(vehicle))
    };
}
```

Le code précédent utilise une expression switch (différente d'une instructionswitch) qui teste le modèle de déclaration. Une **expression switch** commence par la variable, `vehicle` dans le code précédent, suivie du mot clé `switch`. Viennent ensuite toutes les **branches switch** entre accolades. L'expression `switch` ajoute d'autres perfectionnements à la syntaxe qui entoure l'instruction `switch`. Le mot clé `case` est omis, et le résultat de chaque branche est une expression. Les deux dernières montrent une nouvelle fonctionnalité du langage. Le cas `{ }` représente n'importe quel objet non Null ne correspondant à aucune des branches antérieures. Il intercepte tous les types incorrects passés à cette méthode. Le cas `{ }` doit suivre les cas de chaque type de véhicule. Si l'ordre était inversé, le cas `{ }` serait prioritaire. Enfin, le `null`modèle constant détecte quand `null` est passé à cette méthode. Le modèle `null` peut être en dernier car les autres modèles correspondent uniquement à un objet non null du type correct.

Pour tester ce code, utilisez le code suivant dans `Program.cs` :

C#

```
using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
```

```
using LiveryRegistration;

using toll_calculator;

var tollCalc = new TollCalculator();

var car = new Car();
var taxi = new Taxi();
var bus = new Bus();
var truck = new DeliveryTruck();

Console.WriteLine($"The toll for a car is {tollCalc.CalculateToll(car)}");
Console.WriteLine($"The toll for a taxi is {tollCalc.CalculateToll(taxi)}");
Console.WriteLine($"The toll for a bus is {tollCalc.CalculateToll(bus)}");
Console.WriteLine($"The toll for a truck is
{tollCalc.CalculateToll(truck)}");

try
{
    tollCalc.CalculateToll("this will fail");
}
catch (ArgumentException e)
{
    Console.WriteLine("Caught an argument exception when using the wrong
type");
}
try
{
    tollCalc.CalculateToll(null!);
}
catch (ArgumentNullException e)
{
    Console.WriteLine("Caught an argument exception when using null");
}
```

Ce code est inclus dans le projet de démarrage, mais il est commenté. Supprimez les commentaires et vous pourrez tester ce que vous avez écrit.

On commence à voir comment les modèles peuvent nous aider à créer des algorithmes dans lesquels le code et les données sont séparés. L' expression `switch` teste le type et produit différentes valeurs en fonction des résultats. Ce n'est que le début.

Ajouter la tarification en fonction du taux d'occupation

L'autorité de péage souhaite inciter les véhicules à circuler à capacité maximale. Elle a décidé de fixer un tarif plus élevé pour les véhicules transportant peu de passagers et un prix inférieur pour les véhicules pleins, de façon à encourager ces derniers :

- Voitures et taxis sans passager : +0,50 $.
- Voitures et taxis avec deux passagers : remise de 0,50 $.
- Voitures et taxis avec trois passagers ou plus : -1,00 $.
- Bus remplis à moins de 50 % : +2,00 $.
- Bus remplis à plus de 90 % : -1,00 $.

Il est possible d'implémenter ces règles avec un modèle de propriété dans la même expression switch. Un modèle de propriété compare une valeur de propriété à une valeur constante. Le modèle de propriété examine les propriétés de l'objet une fois que le type a été déterminé. L'unique cas `Car` est étendu à quatre cas différents :

C#

```
vehicle switch
{
    Car {Passengers: 0} => 2.00m + 0.50m,
    Car {Passengers: 1} => 2.0m,
    Car {Passengers: 2} => 2.0m - 0.50m,
    Car                 => 2.00m - 1.0m,

    // ...
};
```

Les trois premiers cas testent le type `Car`, puis vérifient la valeur de la propriété `Passengers`. Si les deux correspondent, cette expression est évaluée et retournée.

Les cas des taxis sont étendus de la même manière :

C#

```
vehicle switch
{
    // ...

    Taxi {Fares: 0}  => 3.50m + 1.00m,
    Taxi {Fares: 1}  => 3.50m,
    Taxi {Fares: 2}  => 3.50m - 0.50m,
    Taxi             => 3.50m - 1.00m,

    // ...
};
```

Maintenant, implémentons les règles de taux d'occupation en développant les cas des bus, comme dans l'exemple suivant :

C#

```
vehicle switch
{
    // ...

    Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m +
2.00m,
    Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m -
1.00m,
    Bus => 5.00m,

    // ...
};
```

L'autorité de péage ne se soucie pas du nombre de passagers des camions de livraison, et ajuste le montant en fonction du poids du camion, en procédant ainsi :

- Camions de plus de 5 000 lb : +5,00 $.
- Camions de moins de 3000 lb : -2,00 $.

Cette règle est implémentée dans le code suivant :

C#

```
vehicle switch
{
    // ...

    DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
    DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
    DeliveryTruck => 10.00m,
};
```

Le code précédent montre la clause `when` d'une branche switch. Cette clause `when` sert à tester les conditions autres que l'égalité sur une propriété. Une fois que vous avez terminé, vous disposez d'une méthode qui ressemble beaucoup au code suivant :

C#

```
vehicle switch
{
    Car {Passengers: 0}        => 2.00m + 0.50m,
    Car {Passengers: 1}        => 2.0m,
    Car {Passengers: 2}        => 2.0m - 0.50m,
    Car                        => 2.00m - 1.0m,

    Taxi {Fares: 0}  => 3.50m + 1.00m,
    Taxi {Fares: 1}  => 3.50m,
    Taxi {Fares: 2}  => 3.50m - 0.50m,
    Taxi             => 3.50m - 1.00m,
```

```
    Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m +
2.00m,
    Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m -
1.00m,
    Bus => 5.00m,

    DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
    DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
    DeliveryTruck => 10.00m,

    { }     => throw new ArgumentException(message: "Not a known vehicle
type", paramName: nameof(vehicle)),
    null    => throw new ArgumentNullException(nameof(vehicle))
};
```

La plupart de ces branches switch sont des exemples de **modèles récursifs**. Par exemple, `Car { Passengers: 1}` indique un modèle de constante à l'intérieur d'un modèle de propriété.

Il est possible de rendre ce code moins répétitif avec des switch imbriqués. Dans les exemples précédents, `Car` et `Taxi` ont chacun quatre branches différentes. Dans les deux cas, vous pouvez créer un modèle de déclaration qui alimente un modèle constant. Cette technique est illustrée dans le code suivant :

C#

```
public decimal CalculateToll(object vehicle) =>
    vehicle switch
    {
        Car c => c.Passengers switch
        {
            0 => 2.00m + 0.5m,
            1 => 2.0m,
            2 => 2.0m - 0.5m,
            _ => 2.00m - 1.0m
        },

        Taxi t => t.Fares switch
        {
            0 => 3.50m + 1.00m,
            1 => 3.50m,
            2 => 3.50m - 0.50m,
            _ => 3.50m - 1.00m
        },

        Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m +
2.00m,
        Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m -
1.00m,
        Bus b => 5.00m,
```

```
        DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
        DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
        DeliveryTruck t => 10.00m,

        { }  => throw new ArgumentException(message: "Not a known vehicle
type", paramName: nameof(vehicle)),
        null => throw new ArgumentNullException(nameof(vehicle))
    };
```

Dans l'exemple précédent, l'expression récursive évite de répéter les branches `Car` et `Taxi` contenant des branches enfant qui testent la valeur de propriété. Cette technique n'est pas utilisée pour les branches `Bus` et `DeliveryTruck`, car elles correspondent à des plages de test pour la propriété, et non à des valeurs discrètes.

Ajouter la tarification heures creuses/heures de pointe

En guise de fonctionnalité finale, l'autorité de péage souhaite ajouter une tarification soumise à une contrainte horaire. Pendant les heures d'affluence du matin et du soir, les péages sont doublés. Cette règle ne s'applique au trafic que dans un sens : en allant vers la ville le matin et en en sortant le soir. Le reste du temps, pendant la journée de travail, les péages augmentent de 50 %. Tard le soir et tôt le matin, ils sont réduits de 25 %. Le taux normal s'applique le weekend, quelle que soit l'heure. Vous pouvez utiliser une série d'instructions `if` et `else` pour exprimer ceci à l'aide du code suivant :

C#

```
public decimal PeakTimePremiumIfElse(DateTime timeOfToll, bool inbound)
{
    if ((timeOfToll.DayOfWeek == DayOfWeek.Saturday) ||
        (timeOfToll.DayOfWeek == DayOfWeek.Sunday))
    {
        return 1.0m;
    }
    else
    {
        int hour = timeOfToll.Hour;
        if (hour < 6)
        {
            return 0.75m;
        }
        else if (hour < 10)
        {
            if (inbound)
            {
                return 2.0m;
```

```
            }
            else
            {
                return 1.0m;
            }
        }
        else if (hour < 16)
        {
            return 1.5m;
        }
        else if (hour < 20)
        {
            if (inbound)
            {
                return 1.0m;
            }
            else
            {
                return 2.0m;
            }
        }
        else // Overnight
        {
            return 0.75m;
        }
    }
}
```

Le code précédent fonctionne correctement, mais n'est pas lisible. Vous devez chaîner tous les cas d'entrée et les instructions imbriquées `if` pour comprendre le code. Au lieu de cela, vous allez utiliser les critères spéciaux pour cette fonctionnalité, mais vous l'intégrerez à d'autres techniques. Vous pourriez élaborer une seule expression de critères spéciaux qui tiendrait compte de toutes les combinaisons direction/jour de la semaine/heure. Elle serait cependant complexe, difficile à lire, à comprendre et à vérifier. Combinons plutôt ces méthodes afin de créer un tuple de valeurs décrivant de manière concise tous ces états. Ensuite, utilisons les critères spéciaux pour calculer un multiplicateur de péage. Le tuple comporte trois conditions discrètes :

- le jour : semaine / weekend ;
- la tranche horaire ;
- la direction : vers l'intérieur ou l'extérieur de la ville.

Le tableau suivant montre les combinaisons de valeurs d'entrée et le multiplicateur tarifaire :

Jour	Temps	Sens	Premium
Jour de la semaine	Heure de pointe du matin	Vers l'intérieur de la ville	x 2,00

Jour	Temps	Sens	Premium
Jour de la semaine	Heure de pointe du matin	Vers l'extérieur de la ville	x 1,00
Jour de la semaine	Journée	Vers l'intérieur de la ville	x 1,50
Jour de la semaine	Journée	Vers l'extérieur de la ville	x 1,50
Jour de la semaine	Heure de pointe du soir	Vers l'intérieur de la ville	x 1,00
Jour de la semaine	Heure de pointe du soir	Vers l'extérieur de la ville	x 2,00
Jour de la semaine	Nuit	Vers l'intérieur de la ville	x 0,75
Jour de la semaine	Nuit	Vers l'extérieur de la ville	x 0,75
Week-end	Heure de pointe du matin	Vers l'intérieur de la ville	x 1,00
Week-end	Heure de pointe du matin	Vers l'extérieur de la ville	x 1,00
Week-end	Journée	Vers l'intérieur de la ville	x 1,00
Week-end	Journée	Vers l'extérieur de la ville	x 1,00
Week-end	Heure de pointe du soir	Vers l'intérieur de la ville	x 1,00
Week-end	Heure de pointe du soir	Vers l'extérieur de la ville	x 1,00
Week-end	Nuit	Vers l'intérieur de la ville	x 1,00
Week-end	Nuit	Vers l'extérieur de la ville	x 1,00

Les trois variables produisent 16 combinaisons différentes. On peut simplifier l'expression switch finale en associant certaines conditions.

Le système qui collecte les péages utilise une structure DateTime pour indiquer l'heure de collecte. Élaborons les méthodes de membre qui créent des variables à partir du tableau précédent. La fonction suivante utilise une expression switch de critères spéciaux pour exprimer si DateTime représente un jour de weekend ou de semaine :

C#

```
private static bool IsWeekDay(DateTime timeOfToll) =>
    timeOfToll.DayOfWeek switch
    {
        DayOfWeek.Monday    => true,
        DayOfWeek.Tuesday   => true,
        DayOfWeek.Wednesday => true,
        DayOfWeek.Thursday  => true,
        DayOfWeek.Friday    => true,
        DayOfWeek.Saturday  => false,
```

```
        DayOfWeek.Sunday    => false
    };
```

Cette méthode est correcte, mais elle est répétitive. On peut la simplifier comme dans le code suivant :

C#

```
private static bool IsWeekDay(DateTime timeOfToll) =>
    timeOfToll.DayOfWeek switch
    {
        DayOfWeek.Saturday => false,
        DayOfWeek.Sunday => false,
        _ => true
    };
```

Ensuite, ajoutons une fonction similaire pour catégoriser l'heure dans les blocs :

C#

```
private enum TimeBand
{
    MorningRush,
    Daytime,
    EveningRush,
    Overnight
}

private static TimeBand GetTimeBand(DateTime timeOfToll) =>
    timeOfToll.Hour switch
    {
        < 6 or > 19 => TimeBand.Overnight,
        < 10 => TimeBand.MorningRush,
        < 16 => TimeBand.Daytime,
        _ => TimeBand.EveningRush,
    };
```

Ajoutez un `enum` privé pour convertir chaque plage de temps en une valeur discrète. Ensuite, la méthode `GetTimeBand` utilise des modèles relationnels et des modèles or conjonctifs, tous deux ajoutés dans C# 9.0. Un modèle relationnel vous permet de tester une valeur numérique à l'aide de `<`, de `>`, de `<=` ou de `>=`. Le modèle `or` teste si une expression correspond à un ou plusieurs modèles. Vous pouvez également utiliser un modèle `and` pour vous assurer qu'une expression correspond à deux modèles distincts et un modèle `not` pour tester qu'une expression ne correspond pas à un modèle.

Maintenant que nous avons créé ces méthodes, utilisons une autre expression `switch` avec le **modèle de tuple** pour calculer le multiplicateur tarifaire. On pourrait concevoir

une expression `switch` comportant les 16 branches :

C#

```
public decimal PeakTimePremiumFull(DateTime timeOfToll, bool inbound) =>
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
        (true, TimeBand.MorningRush, true) => 2.00m,
        (true, TimeBand.MorningRush, false) => 1.00m,
        (true, TimeBand.Daytime, true) => 1.50m,
        (true, TimeBand.Daytime, false) => 1.50m,
        (true, TimeBand.EveningRush, true) => 1.00m,
        (true, TimeBand.EveningRush, false) => 2.00m,
        (true, TimeBand.Overnight, true) => 0.75m,
        (true, TimeBand.Overnight, false) => 0.75m,
        (false, TimeBand.MorningRush, true) => 1.00m,
        (false, TimeBand.MorningRush, false) => 1.00m,
        (false, TimeBand.Daytime, true) => 1.00m,
        (false, TimeBand.Daytime, false) => 1.00m,
        (false, TimeBand.EveningRush, true) => 1.00m,
        (false, TimeBand.EveningRush, false) => 1.00m,
        (false, TimeBand.Overnight, true) => 1.00m,
        (false, TimeBand.Overnight, false) => 1.00m,
    };
```

Le code ci-dessus fonctionne, mais on peut le simplifier. Les huit combinaisons du weekend correspondent au même péage. Elles sont remplaçables par la ligne suivante :

C#

```
(false, _, _) => 1.0m,
```

Le trafic entrant et le trafic sortant ont le même multiplicateur pendant la journée en semaine et pendant la nuit. Ces quatre branches switch peuvent être remplacées par les deux lignes suivantes :

C#

```
(true, TimeBand.Overnight, _) => 0.75m,
(true, TimeBand.Daytime, _)   => 1.5m,
```

Après ces deux modifications, le code devrait se présenter ainsi :

C#

```
public decimal PeakTimePremium(DateTime timeOfToll, bool inbound) =>
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
```

```
        (true, TimeBand.MorningRush, true)  => 2.00m,
        (true, TimeBand.MorningRush, false) => 1.00m,
        (true, TimeBand.Daytime,     _)     => 1.50m,
        (true, TimeBand.EveningRush, true)  => 1.00m,
        (true, TimeBand.EveningRush, false) => 2.00m,
        (true, TimeBand.Overnight,   _)     => 0.75m,
        (false, _,                   _)     => 1.00m,
    };
```

Enfin, on peut supprimer les deux branches des heures de pointe au tarif normal, ce qui permet de remplacer `false` par un discard (`_`) dans la branche switch finale. On obtient la méthode suivante une fois terminée :

C#

```
public decimal PeakTimePremium(DateTime timeOfToll, bool inbound) =>
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
        (true, TimeBand.Overnight, _) => 0.75m,
        (true, TimeBand.Daytime, _) => 1.5m,
        (true, TimeBand.MorningRush, true) => 2.0m,
        (true, TimeBand.EveningRush, false) => 2.0m,
        _ => 1.0m,
    };
```

Cet exemple met en évidence un des avantages des critères spéciaux : les branches du modèle sont évaluées dans l'ordre. Si vous les réorganisez de telle sorte qu'une branche antérieure gère un des cas suivants, le compilateur vous avertit que le code est inaccessible. Grâce à ces règles de langage, nous avons pu effectuer facilement les simplifications précédentes sans craindre de modifier le code.

Les critères spéciaux rendent certains types de code plus lisibles et offrent une alternative aux techniques orientées objet quand vous ne pouvez pas ajouter de code à vos classes. Le cloud est à l'origine d'une séparation entre les données et les fonctionnalités. La *forme* des données et les *opérations* effectuées sur ces dernières ne sont pas nécessairement décrites ensemble. Dans ce tutoriel, nous avons exploité les données existantes d'une manière totalement différente de leur fonction d'origine. Les critères spéciaux offrent la possibilité d'écrire des fonctionnalités qui remplacent ces types, même sans pouvoir les étendre.

Étapes suivantes

Vous pouvez télécharger le code terminé dans référentiel GitHub dotnet/samples. Explorez les modèles par vous-même et ajoutez cette technique à vos activités de

codage régulières. Ces techniques représentent une autre façon d'aborder les problèmes et de créer de nouvelles fonctionnalités.

Voir aussi

- Modèles
- switch, expression

Comment gérer une exception à l'aide de try-catch

Article • 07/04/2023

L'objectif d'un bloc try-catch est d'intercepter et de gérer une exception générée par du code opérationnel. Certaines exceptions peuvent être gérées dans un bloc `catch` et le problème résolu sans que l'exception soit levée à nouveau, mais le plus souvent la seule chose que vous puissiez faire est de vous assurer que l'exception appropriée est levée.

Exemple

Dans cet exemple, IndexOutOfRangeException n'est pas l'exception la plus appropriée : ArgumentOutOfRangeException a de plus de sens pour la méthode car l'erreur est provoquée par l'argument `index` passé par l'appelant.

C#

```
static int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e)  // CS0168
    {
        Console.WriteLine(e.Message);
        // Set IndexOutOfRangeException to the new exception's
InnerException.
        throw new ArgumentOutOfRangeException("index parameter is out of
range.", e);
    }
}
```

Commentaires

Le code qui provoque une exception est cloisonné dans le bloc `try`. Une instruction `catch` est ajoutée juste après pour gérer `IndexOutOfRangeException`, si elle se produit. Le bloc `catch` gère `IndexOutOfRangeException` et lève l'`ArgumentOutOfRangeException` plus appropriée à la place. Pour fournir à l'appelant autant d'informations que possible, pensez à spécifier l'exception d'origine comme InnerException de la nouvelle exception.

Comme la propriété InnerException a la valeur read-only, vous devez l'affecter dans le constructeur de la nouvelle exception.

Comment exécuter le code de nettoyage à l'aide de finally

Article • 21/03/2023

L'objectif d'une instruction `finally` est de vérifier que le nettoyage nécessaire des objets, généralement ceux contenant des ressources externes, se produit immédiatement, même si une exception est levée. Un exemple de nettoyage de ce type est l'appel à Close sur un FileStream immédiatement après son utilisation au lieu d'attendre que l'objet soit récupéré par garbage collection par le Common Language Runtime, comme suit :

C#

```
static void CodeWithoutCleanup()
{
    FileStream? file = null;
    FileInfo fileInfo = new FileInfo("./file.txt");

    file = fileInfo.OpenWrite();
    file.WriteByte(0xF);

    file.Close();
}
```

Exemple

Pour transformer le code précédent en instruction `try-catch-finally`, le code de nettoyage est séparé du code actif comme suit.

C#

```
static void CodeWithCleanup()
{
    FileStream? file = null;
    FileInfo? fileInfo = null;

    try
    {
        fileInfo = new FileInfo("./file.txt");

        file = fileInfo.OpenWrite();
        file.WriteByte(0xF);
    }
    catch (UnauthorizedAccessException e)
    {
```

```
            Console.WriteLine(e.Message);
        }
        finally
        {
            file?.Close();
        }
    }
```

Comme une exception peut se produire à tout moment dans le bloc `try` avant l'appel à `OpenWrite()`, ou comme l'appel `OpenWrite()` lui-même peut échouer, nous n'avons aucune garantie que le fichier est ouvert quand nous essayons de le fermer. Le bloc `finally` ajoute un contrôle pour garantir que l'objet FileStream n'est pas `null` avant d'appeler la méthode Close. Sans le contrôle `null`, le bloc `finally` pourrait lever sa propre NullReferenceException, mais la levée d'exceptions dans des blocs `finally` doit être évitée, dans la mesure du possible.

Une connexion de base de données est également susceptible d'être fermée dans un bloc `finally`. Le nombre de connexions à un serveur de base de données étant parfois limité, vous devez fermer les connexions de base de données le plus rapidement possible. Si une exception est levée avant que vous puissiez fermer votre connexion, il vaut mieux utiliser le bloc `finally` qu'attendre le nettoyage de la mémoire.

Voir aussi

- using, instruction
- try-catch
- try-finally
- try-catch-finally

Nouveautés de la version 11 de C#

Article • 20/03/2023

Les fonctionnalités suivantes ont été ajoutées dans C# 11 :

- Littéraux de chaîne bruts
- Prise en charge des mathématiques génériques
- Attributs génériques
- Littéraux de chaîne UTF-8
- Nouvelles lignes dans les expressions d'interpolation de chaîne
- Modèles de liste
- Types locaux de fichiers
- Membres nécessaires
- Structs par défaut automatiques
- Correspondance Span<char> de modèle sur une constante string
- Portée de nameof étendue
- IntPtr numérique
- Champs ref et scoped ref
- Amélioration de la conversion de groupe de méthodes en instance déléguée
- Vague d'avertissement 7

Vous pouvez télécharger la dernière version de Visual Studio 2022. Sinon, essayez toutes ces fonctionnalités avec le SDK .NET 7 que vous pouvez télécharger à partir de la page Téléchargements .NET.

> **Notes**
>
> Vos commentaires sur ces fonctionnalités nous intéressent. Si vous rencontrez des problèmes avec l'une de ces nouvelles fonctionnalités, créez un **nouveau problème** dans le référentiel **dotnet/roslyn**.

Attributs génériques

Vous pouvez déclarer une classe générique dont la classe de base est System.Attribute. Cette fonctionnalité fournit une syntaxe plus pratique pour les attributs qui exigent un paramètre System.Type. Auparavant, il était nécessaire de créer un attribut qui prenait un `Type` comme paramètre de constructeur :

```
C#
```

```
// Before C# 11:
public class TypeAttribute : Attribute
{
   public TypeAttribute(Type t) => ParamType = t;

   public Type ParamType { get; }
}
```

Pour appliquer l'attribut, il fallait utiliser l'opérateur typeof :

C#

```
[TypeAttribute(typeof(string))]
public string Method() => default;
```

À l'aide de cette nouvelle fonctionnalité, vous pouvez créer plutôt un attribut générique :

C#

```
// C# 11 feature:
public class GenericAttribute<T> : Attribute { }
```

Ensuite, vous spécifiez le paramètre de type permettant d'utiliser l'attribut :

C#

```
[GenericAttribute<string>()]
public string Method() => default;
```

Vous devez fournir tous les paramètres de type lorsque vous appliquez l'attribut. En d'autres termes, le type générique doit être entièrement construit. Dans l'exemple ci-dessus, les parenthèses vides (`(` et `)`) peuvent être omises car l'attribut n'a pas d'arguments.

C#

```
public class GenericType<T>
{
   [GenericAttribute<T>()] // Not allowed! generic attributes must be fully
constructed types.
   public string Method() => default;
}
```

Les arguments de type doivent respecter les mêmes restrictions que l'opérateur typeof . Les types qui exigent des annotations de métadonnées ne sont pas autorisés comme paramètres de type, par exemple :

- `dynamic`
- `string?` (ou tout type référence Nullable)
- `(int X, int Y)` (ou tout autre type de tuple suivant la syntaxe de tuple C#)

Ces types ne sont pas directement représentés dans les métadonnées. Ils incluent des annotations qui les décrivent. Dans tous les cas, vous pouvez utiliser le type sous-jacent à la place :

- `object` plutôt que `dynamic`
- `string` plutôt que `string?`.
- `ValueTuple<int, int>` plutôt que `(int X, int Y)`.

Prise en charge des mathématiques génériques

Plusieurs fonctionnalités de langage prennent en charge les mathématiques génériques :

- Membres `static virtual` dans les interfaces
- Opérateurs checked définis par l'utilisateur
- Opérateurs de décalage souples
- Opérateur de décalage vers la droite non signé

Vous pouvez ajouter des membres `static abstract` ou `static virtual` dans des interfaces pour définir des interfaces qui incluent des opérateurs surchargeables, d'autres membres statiques et des propriétés statiques. Le scénario principal de cette fonctionnalité consiste à utiliser des opérateurs mathématiques dans des types génériques. Par exemple, vous pouvez implémenter l'interface `System.IAdditionOperators<TSelf, TOther, TResult>` dans un type qui implémente `operator +`. D'autres interfaces définissent d'autres opérations mathématiques ou valeurs bien définies. Pour en savoir plus sur la nouvelle syntaxe, consultez l'article sur les interfaces. Les interfaces qui incluent des méthodes `static virtual` sont généralement des interfaces génériques. De plus, la plupart déclarent une contrainte selon laquelle le paramètre de type implémente l'interface déclarée.

Pour plus d'informations ou pour essayer la fonctionnalité vous-même, consultez le tutoriel Exploration des membres abstraits statiques de l'interface ou le billet de blog Fonctionnalités d'évaluation de .NET 6 – Mathématiques génériques .

Les mathématiques génériques imposent d'autres exigences au langage.

- *Opérateur de décalage vers la droite non signé* : avant C# 11, pour forcer un décalage vers la droite non signé, vous deviez caster tout type entier signé en type non signé, effectuer le décalage, puis recaster le résultat en type signé. À partir de C# 11, vous pouvez utiliser l'*opérateur de décalage non signé* `>>>`.
- *Exigences de l'opérateur de décalage souple* : C# 11 élimine l'exigence selon laquelle le deuxième opérande doit être un `int` ou être implicitement convertible en `int`. Cette modification permet d'utiliser des types qui implémentent des interfaces mathématiques génériques dans ces emplacements.
- *Opérateurs checked et unchecked définis par l'utilisateur* : les développeurs peuvent désormais définir les opérateurs arithmétiques `checked` et `unchecked`. Le compilateur génère des appels à la variante appropriée en fonction du contexte actuel. Pour en savoir plus sur les opérateurs `checked`, consultez l'article sur les opérateurs arithmétiques.

Types numériques `IntPtr` et `UIntPtr`

Les types `nint` et `nuint` sont désormais les alias de System.IntPtr et de System.UIntPtr, respectivement.

Nouvelles lignes dans les interpolations de chaînes

Le texte situé à l'intérieur des caractères `{` et `}` d'une interpolation de chaîne peut désormais s'étendre sur plusieurs lignes. Le texte qui se trouve entre les marqueurs `{` et `}` est analysé comme du code C#. Tout le code C# valide, y compris les nouvelles lignes, est autorisé. Cette fonctionnalité facilite la lecture des interpolations de chaînes qui utilisent de longues expressions C#, comme les expressions `switch` de critères spéciaux et les requêtes LINQ.

Pour plus d'informations sur la fonctionnalité de nouvelles lignes, consultez l'article Interpolations de chaînes dans la documentation de référence du langage.

Modèles de liste

Les *modèles de liste* étendent les critères spéciaux pour intégrer la correspondance avec des séquences d'éléments d'une liste ou d'un tableau. Par exemple, `sequence is [1, 2, 3]` est `true` lorsque `sequence` est un tableau ou une liste de trois entiers (1, 2 et 3). Vous

pouvez faire correspondre des éléments à l'aide de n'importe quel modèle, notamment des modèles constants et relationnels et des modèles de type et de propriété. Le modèle discard (`_`) correspond à n'importe quel élément unique, et le nouveau *modèle de plage* (`..`) à n'importe quelle séquence de zéro, un ou plusieurs éléments.

Pour plus d'informations sur les modèles de liste, consultez l'article Critères spéciaux dans la documentation de référence du langage.

Amélioration de la conversion de groupe de méthodes en instance déléguée

Le standard C# des Conversions de groupes de méthodes inclut désormais l'élément suivant :

> - La conversion est autorisée (mais non obligatoire) pour utiliser une instance déléguée existante qui contient déjà ces références.

Les versions précédentes du standard empêchaient le compilateur de réutiliser l'objet délégué créé pour une conversion de groupe de méthodes. Le compilateur de la version 11 de C# le met en cache et le réutilise. Cette fonctionnalité a été introduite dans Visual Studio 2022 version 17.2 en préversion et dans .NET 7 Preview 2.

Littéraux de chaîne bruts

Les *littéraux de chaîne brute* constituent un nouveau format pour les littéraux de chaîne. Ils peuvent contenir du texte arbitraire, y compris des espaces blancs, de nouvelles lignes, des guillemets incorporés et d'autres caractères spéciaux, sans nécessiter de séquences d'échappement. Un littéral de chaîne brute commence par au moins trois caractères guillemets doubles (""""). Il se termine par le même nombre de guillemets doubles. En règle générale, il utilise trois guillemets doubles sur une seule ligne pour démarrer la chaîne, et trois guillemets doubles sur une ligne distincte pour la terminer. Les nouvelles lignes qui suivent le guillemet ouvrant et qui précèdent le guillemet fermant ne sont pas inclus dans le contenu final :

C#

```
string longMessage = """
    This is a long message.
    It has several lines.
        Some are indented
                more than others.
    Some should start at the first column.
```

```
    Some have "quoted text" in them.
    """;
```

Tout espace blanc situé à gauche des guillemets doubles fermants est supprimé du littéral de chaîne. Les littéraux de chaîne brute peuvent être combinés avec l'interpolation de chaîne pour inclure des accolades dans le texte de sortie. Plusieurs caractères `$` indiquent le nombre d'accolades consécutives qui commencent et terminent l'interpolation :

C#

```
var location = $$"""
   You are at {{{Longitude}}, {{Latitude}}}
   """;
```

L'exemple précédent spécifie que deux accolades commencent et terminent une interpolation. La troisième accolade ouvrante et fermante est incluse dans la chaîne de sortie.

Pour plus d'informations sur les littéraux de chaîne brute, consultez l'article Chaînes dans le guide de programmation, ainsi que les articles Littéraux de chaîne et Chaînes interpolées dans la documentation de référence du langage.

Struct par défaut automatique

Le compilateur C# 11 garantit que tous les champs d'un type `struct` sont initialisés à leur valeur par défaut dans le cadre de l'exécution d'un constructeur. Cette modification signifie que tout champ ou propriété automatique non initialisé par un constructeur est automatiquement initialisé par le compilateur. Les structs dans lesquels le constructeur n'affecte pas définitivement tous les champs sont maintenant compilés, et tous les champs qui ne sont pas initialisés explicitement sont définis avec leur valeur par défaut. Pour en savoir plus sur la façon dont cette modification affecte l'initialisation des structs, consultez l'article sur les structs.

Critères spéciaux avec `Span<char>` ou `ReadOnlySpan<char>` sur une constante `string`

Depuis plusieurs versions, vous pouvez utiliser les critères spéciaux pour tester si un `string` a une valeur constante spécifique. Vous pouvez désormais utiliser la même logique des critères spéciaux avec des variables `Span<char>` ou `ReadOnlySpan<char>`.

Portée nameof étendue

Les noms de paramètre de type et les noms de paramètre sont désormais dans la portée quand ils sont utilisés dans une expression `nameof` dans une déclaration d'attribut sur cette méthode. Cette fonctionnalité signifie que vous pouvez utiliser l'opérateur `nameof` pour spécifier le nom d'un paramètre de méthode dans un attribut sur la déclaration de méthode ou de paramètre. Cette fonctionnalité est le plus souvent utile pour ajouter des attributs à l'analyse de l'état null.

Littéraux de chaîne UTF-8

Vous pouvez spécifier le suffixe `u8` sur un littéral de chaîne pour spécifier le codage de caractères UTF-8. Si votre application a besoin de chaînes UTF-8, pour des constantes de chaîne HTTP ou des protocoles de texte similaires, vous pouvez utiliser cette fonctionnalité pour simplifier la création de chaînes UTF-8.

Pour en savoir plus sur les littéraux de chaîne UTF-8, consultez la section sur les littéraux de chaîne de l'article sur les types de référence intégrés.

Membres nécessaires

Vous pouvez ajouter le requiredmodificateur à des propriétés et à des champs pour obliger les constructeurs et les appelants à initialiser ces valeurs. Vous pouvez ajouter System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute à des constructeurs pour informer le compilateur qu'un constructeur initialise *tous* les membres obligatoires.

Pour plus d'informations sur les membres obligatoires, consultez la section init-only de l'article sur les propriétés.

Champs `ref` et variables `ref scoped`

Vous pouvez déclarer des champs `ref` dans un ref struct. Cela permet de prendre en charge des types tels que System.Span<T> sans attributs spéciaux ou types internes masqués.

Vous pouvez ajouter le modificateur scoped à n'importe quelle déclaration `ref`. Cela limite la portée où la référence peut être échappée.

Types locaux de fichiers

À partir de C# 11, vous pouvez utiliser le modificateur d'accès `file` pour créer un type dont la visibilité est limitée au fichier source dans lequel il est déclaré. Cette fonctionnalité permet aux auteurs de générateurs de source d'éviter les collisions de noms. Pour en savoir plus sur cette fonctionnalité, consultez l'article sur les types inclus dans la portée du fichier dans la référence du langage.

Voir aussi

- Nouveautés de .NET 7

Nouveautés dans C# 10

Article • 10/05/2023

C# 10 ajoute les fonctionnalités et améliorations suivantes au langage C# :

- Structs d'enregistrement
- Améliorations des types de structure
- Gestionnaires de chaînes interpolés
- Directives global using
- Déclaration d'espace de noms dans l'étendue du fichier
- Modèles de propriétés étendus
- Améliorations apportées aux expressions lambda
- Autoriser les chaînes interpolées const
- Les types d'enregistrements peuvent sceller ToString()
- Amélioration de l'affectation définitive
- Autoriser à la fois l'affectation et la déclaration dans la même déconstruction
- Autoriser l'attribut AsyncMethodBuilder sur les méthodes
- Attribut CallerArgumentExpression
- Pragma #line amélioré
- Vague d'avertissements 6

C# 10 est pris en charge sur **.NET 6**. Pour plus d'informations, consultez Contrôle de version du langage C#.

Vous pouvez télécharger la dernière version du kit SDK .NET 6 sur la page de téléchargements .NET. Vous avez également la possibilité de télécharger Visual Studio 2022, qui inclut la version 6 du kit SDK .NET.

> **Notes**
>
> Vos commentaires sur ces fonctionnalités nous intéressent. Si vous rencontrez des problèmes avec l'une de ces nouvelles fonctionnalités, créez un **nouveau problème** dans le référentiel **dotnet/roslyn**.

Structs d'enregistrement

Vous pouvez déclarer des enregistrements de type valeur à l'aide des déclarations record struct ou readonly record struct. Vous pouvez maintenant clarifier que `record` est un type référence avec la déclaration `record class`.

Améliorations des types de structure

C# 10 présente les améliorations suivantes relatives aux types de structure :

- Vous pouvez déclarer un constructeur sans paramètre d'instance dans un type de structure et initialiser un champ ou une propriété d'instance à sa déclaration. Pour plus d'informations, consultez la section Initialisation de struct et valeurs par défaut de l'article Types de structure.
- Un opérande de gauche de l'expression with peut être de n'importe quel type de structure ou d'un type anonyme (référence).

Gestionnaire de chaînes interpolées

Vous pouvez créer un type qui génère la chaîne résultante à partir d'une expression de chaîne interpolée. Les bibliothèques .NET utilisent cette fonctionnalité dans un grand nombre d'API. Vous pouvez en créer un en suivant ce didacticiel.

Directives using globales

Vous pouvez ajouter le modificateur `global` à n'importe quel à l'aide de la directive pour indiquer au compilateur que la directive s'applique à tous les fichiers sources de la compilation. Il s'agit généralement de tous les fichiers sources d'un projet.

Déclaration d'espace de noms délimitée aux fichiers

Vous pouvez utiliser une nouvelle forme de déclaration namespace pour déclarer que toutes les déclarations suivantes sont membres de l'espace de noms déclaré :

```
C#

namespace MyNamespace;
```

Cette nouvelle syntaxe enregistre à la fois l'espace horizontal et vertical pour les déclarations `namespace`.

Modèles de propriétés étendues

À compter de C# 10, vous pouvez référencer des propriétés ou des champs imbriqués dans un modèle de propriété. Par exemple, un modèle du formulaire

```
C#

{ Prop1.Prop2: pattern }
```

est valide en C# 10 et versions ultérieures et équivalent à

```
C#

{ Prop1: { Prop2: pattern } }
```

est valide en C# 8.0 et versions ultérieures.

Pour plus d'informations, consultez la section relative à la proposition de fonctionnalité modèles de propriétés étendues . Pour plus d'informations sur un modèle de propriété, consultez la section Modèle de propriété de l'article Modèles.

Améliorations apportées aux expressions lambda

C# 10 inclut de nombreuses améliorations à la gestion des expressions lambda :

- Les expressions lambda peuvent avoir un type naturel, où le compilateur peut déduire un type délégué à partir du groupe d'expressions lambda ou de méthodes.
- Les expressions lambda peuvent déclarer un type de retour lorsque le compilateur ne peut pas le déduire.
- Les attributs peuvent être appliqués aux expressions lambda.

Grâce à ces fonctionnalités, les expressions ressemblent davantage aux méthodes et aux fonctions locales. Elles facilitent l'utilisation d'expressions lambda sans déclarer une variable d'un type délégué et leur fonctionnement est plus transparent avec les nouvelles API minimales ASP.NET Core.

Chaînes interpolées constantes

En C# 10, les chaînes `const` peuvent être initialisées à l'aide de l'interpolation de chaîne si tous les espaces réservés sont eux-mêmes des chaînes constantes. L'interpolation de chaînes peut créer des chaînes de constantes plus lisibles lorsque vous générez de telles

chaînes pour les utiliser dans votre application. Les expressions d'espace réservé ne peuvent pas être des constantes numériques, car ces constantes sont converties en chaînes au moment de l'exécution. La culture actuelle en la matière peut affecter leur représentation en tant que chaîne. Pour en savoir plus, consultez la référence de langage sur les expressions const.

Les types d'enregistrements peuvent sceller ToString

En C# 10, vous pouvez ajouter le modificateur `sealed` lorsque vous remplacez `ToString` dans un type d'enregistrement. Le scellement de la méthode `ToString` empêche le compilateur de synthétiser une méthode `ToString` pour tous les types d'enregistrements dérivés. Un `sealed ToString` garantit que tous les types d'enregistrements dérivés utilisent la méthode `ToString` définie dans un type d'enregistrement de base commun. Pour plus d'informations sur cette fonctionnalité, consultez l'article sur les enregistrements.

Affectation et déclaration dans la même déconstruction

Ce changement supprime une restriction des versions antérieures de C#. Auparavant, une déconstruction peut affecter toutes les valeurs à des variables existantes ou initialiser des variables nouvellement déclarées :

C#

```
// Initialization:
(int x, int y) = point;

// assignment:
int x1 = 0;
int y1 = 0;
(x1, y1) = point;
```

C# 10 supprime cette restriction :

C#

```
int x = 0;
(x, int y) = point;
```

Amélioration de l'affectation définitive

Avant C# 10, il était fréquent que l'affectation définie et l'analyse de l'état null produisent des avertissements qui étaient des faux positifs. Ces comparaisons impliquent généralement des comparaisons avec des constantes booléennes, l'accès à une variable uniquement dans les instructions `true` ou `false` dans une instruction `if` et les expressions de coalescence null. Ces exemples ont généré des avertissements dans les versions précédentes de C#, mais pas en C# 10 :

C#

```
string representation = "N/A";
if ((c != null && c.GetDependentValue(out object obj)) == true)
{
   representation = obj.ToString(); // undesired error
}

// Or, using ?.
if (c?.GetDependentValue(out object obj) == true)
{
   representation = obj.ToString(); // undesired error
}

// Or, using ??
if (c?.GetDependentValue(out object obj) ?? false)
{
   representation = obj.ToString(); // undesired error
}
```

L'impact principal de cette amélioration est que les avertissements relatifs à l'affectation définie et l'analyse de l'état null sont plus précis.

Autoriser l'attribut AsyncMethodBuilder sur les méthodes

En C# 10 et versions ultérieures, vous pouvez spécifier un générateur de méthodes asynchrone différent pour une seule méthode, en plus de spécifier le type de générateur de méthodes pour toutes les méthodes qui retournent un type de tâche donné. Un générateur de méthode asynchrone personnalisé permet des scénarios avancés de réglage des performances dans lesquels une méthode donnée peut tirer parti d'un générateur personnalisé.

Pour plus d'informations, consultez la section sur AsyncMethodBuilder dans l'article sur les attributs lus par le compilateur.

Diagnostics d'attribut CallerArgumentExpression

Vous pouvez utiliser le paramètre System.Runtime.CompilerServices.CallerArgumentExpressionAttribute pour spécifier un paramètre que le compilateur remplace par la représentation textuelle d'un autre argument. Cette fonctionnalité permet aux bibliothèques de créer des diagnostics plus spécifiques. Le code suivant teste une condition. Si la condition est fausse, le message d'exception contient la représentation textuelle de l'argument passé à `condition` :

C#

```
public static void Validate(bool condition,
[CallerArgumentExpression("condition")] string? message=null)
{
    if (!condition)
    {
        throw new InvalidOperationException($"Argument failed validation:
<{message}>");
    }
}
```

Vous pouvez en savoir plus sur cette fonctionnalité dans l'article sur les attributs d'informations de l'appelant dans la section de référence du langage.

Pragma amélioré #line

C# 10 prend en charge un nouveau format pour le pragma `#line`. Vous n'utiliserez probablement pas le nouveau format, mais vous verrez ses effets. Les améliorations permettent une sortie plus fine dans des langages spécifiques au domaine (DSL) comme Razor. Le moteur Razor utilise ces améliorations pour améliorer l'expérience de débogage. Vous trouverez des débogueurs qui peuvent mettre en évidence votre source Razor plus précisément. Pour en savoir plus sur la nouvelle syntaxe, consultez l'article sur les directives de préprocesseur dans le guide de référence du langage. Vous pouvez également lire les spécifications de fonctionnalités pour les exemples Razor.

Nouveautés dans C# 9.0

Article • 10/05/2023

C# 9.0 ajoute les fonctionnalités et améliorations suivantes au langage C# :

- Enregistrements
- Setter init uniquement
- Instructions de niveau supérieur
- Améliorations des critères spéciaux
- Performances et interopérabilité
  - Entiers dimensionnés natifs
  - Pointeurs fonction
  - Suppression de l'indicateur localsinit émis
- Caractéristiques d'ajustement et de finalisation
  - Expressions new typées cible
  - static fonctions anonymes
  - Expressions conditionnelles typées cible
  - Types de retour covariant
  - Prise en charge des extensions GetEnumerator pour les boucles foreach
  - Paramètres d'abandon lambda
  - Attributs sur des fonctions locales
- Prise en charge des générateurs de code
  - Initialiseurs de module
  - Nouvelles fonctionnalités pour les méthodes partielles
- Vague d'avertissement 5

C# 9.0 est pris en charge par **.NET 5**. Pour plus d'informations, consultez contrôle de version du langage C#.

Vous pouvez télécharger le dernier SDK .NET à partir de la page de téléchargements .NET.

> **Notes**
>
> Vos commentaires sur ces fonctionnalités nous intéressent. Si vous rencontrez des problèmes avec l'une de ces nouvelles fonctionnalités, créez un **nouveau problème** dans le référentiel **dotnet/roslyn**.

Types d'enregistrements

C# 9.0 introduit des ***types d'enregistrements***. Utilisez le mot clé `record` pour définir un type de référence qui fournit des fonctionnalités intégrées pour l'encapsulation de données. Vous pouvez créer des types d'enregistrements avec des propriétés immuables à l'aide de paramètres positionnels ou de la syntaxe de propriété standard :

C#

```
public record Person(string FirstName, string LastName);
```

C#

```
public record Person
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
};
```

Vous pouvez également créer des types d'enregistrements avec des propriétés et des champs mutables :

C#

```
public record Person
{
    public required string FirstName { get; set; }
    public required string LastName { get; set; }
};
```

Bien que les enregistrements puissent être mutables, ils sont principalement destinés à prendre en charge des modèles de données immuables. Le type d'enregistrement offre les fonctionnalités suivantes :

- Syntaxe concise pour la création d'un type de référence avec des propriétés immuables
- Comportement utile pour un type de référence centré sur les données :
  - Égalité des valeurs
  - Syntaxe concise pour les mutations non destructrices
  - Mise en forme intégrée pour l'affichage
- Prise en charge des hiérarchies d'héritage

Vous pouvez utiliser des types de structure pour concevoir des types centrés sur les données qui fournissent une égalité de valeur avec peu ou pas de comportement. Toutefois, pour les modèles de données relativement volumineux, les types de structure ont certains inconvénients :

- Ils ne prennent pas en charge l'héritage.
- Ils sont moins efficaces pour déterminer l'égalité des valeurs. Pour les types valeur, la méthode ValueType.Equals utilise la réflexion pour rechercher tous les champs. Pour les enregistrements, le compilateur génère la méthode `Equals`. En pratique, l'implémentation de l'égalité des valeurs dans les enregistrements est beaucoup plus rapide.
- Ils utilisent plus de mémoire dans certains scénarios, car chaque instance a une copie complète de toutes les données. Les types d'enregistrements sont des types de référence, donc une instance d'enregistrement ne contient qu'une référence aux données.

Syntaxe positionnelle pour la définition de propriété

Vous pouvez utiliser des paramètres positionnels pour déclarer des propriétés d'un enregistrement et initialiser les valeurs de propriété lorsque vous créez une instance :

C#

```
public record Person(string FirstName, string LastName);

public static void Main()
{
    Person person = new("Nancy", "Davolio");
    Console.WriteLine(person);
    // output: Person { FirstName = Nancy, LastName = Davolio }
}
```

Lorsque vous utilisez la syntaxe positionnelle pour la définition de propriété, le compilateur crée :

- Une propriété publique implémentée automatiquement init-only pour chaque paramètre positionnel fourni dans la déclaration d'enregistrement. Une propriété init-only peut être définie dans le constructeur ou à l'aide d'un initialiseur de propriété.
- Un constructeur principal dont les paramètres correspondent aux paramètres positionnels de la déclaration d'enregistrement.
- Une méthode `Deconstruct` avec un paramètre `out` pour chaque paramètre positionnel fourni dans la déclaration d'enregistrement.

Pour plus d'informations, consultez Syntaxe positionnelle dans l'article de référence du langage C# sur les enregistrements.

Immuabilité

Un type d'enregistrement n'est pas nécessairement immuable. Vous pouvez déclarer des propriétés avec des accesseurs `set` et des champs autres que `readonly`. Toutefois, bien que les enregistrements puissent être mutables, ils facilitent la création de modèles de données immuables. Les propriétés que vous créez à l'aide de la syntaxe positionnelle sont immuables.

L'immuabilité peut être utile lorsque vous souhaitez qu'un type centré sur les données soit thread-safe ou qu'un code de hachage reste le même dans une table de hachage. Cela peut empêcher les bogues qui se produisent lorsque vous transmettez un argument par référence à une méthode, et la méthode modifie de façon inattendue la valeur de l'argument.

Les fonctionnalités propres aux types d'enregistrements sont implémentées par des méthodes synthétisées par le compilateur, et aucune de ces méthodes ne compromet l'immuabilité en modifiant l'état de l'objet.

Égalité des valeurs

L'égalité des valeurs signifie que deux variables d'un type d'enregistrement sont égales si les types correspondent et que toutes les valeurs de propriété et de champ correspondent. Pour d'autres types de référence, l'égalité signifie l'identité. Autrement dit, deux variables d'un type référence sont égales si elles font référence au même objet.

L'exemple suivant illustre l'égalité des valeurs des types d'enregistrements :

C#

```
public record Person(string FirstName, string LastName, string[]
PhoneNumbers);

public static void Main()
{
    var phoneNumbers = new string[2];
    Person person1 = new("Nancy", "Davolio", phoneNumbers);
    Person person2 = new("Nancy", "Davolio", phoneNumbers);
    Console.WriteLine(person1 == person2); // output: True

    person1.PhoneNumbers[0] = "555-1234";
    Console.WriteLine(person1 == person2); // output: True

    Console.WriteLine(ReferenceEquals(person1, person2)); // output: False
}
```

Dans les types `class`, vous pouvez remplacer manuellement les méthodes d'égalité et les opérateurs pour atteindre l'égalité des valeurs, mais le développement et le test de

ce code seraient fastidieux et sujets aux erreurs. Le fait d'avoir cette fonctionnalité intégrée empêche les bogues qui résulteraient de l'oubli de mettre à jour le code de remplacement personnalisé lorsque des propriétés ou des champs sont ajoutés ou modifiés.

Pour plus d'informations, consultez Égalité des valeurs dans l'article de référence du langage C# sur les enregistrements.

Mutation non destructrice

Si vous voulez muter les propriétés immuables d'une instance d'enregistrement, vous pouvez utiliser une expression `with` pour obtenir *une mutation non destructrice*. Une expression `with` crée une nouvelle instance d'enregistrement qui est une copie d'une instance d'enregistrement existante, avec des propriétés et des champs spécifiés modifiés. Utilisez la syntaxe d'initialiseur d'objet pour spécifier les valeurs à modifier, comme illustré dans l'exemple suivant :

C#

```
public record Person(string FirstName, string LastName)
{
    public string[] PhoneNumbers { get; init; }
}

public static void Main()
{
    Person person1 = new("Nancy", "Davolio") { PhoneNumbers = new string[1]
};
    Console.WriteLine(person1);
    // output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers
= System.String[] }

    Person person2 = person1 with { FirstName = "John" };
    Console.WriteLine(person2);
    // output: Person { FirstName = John, LastName = Davolio, PhoneNumbers =
System.String[] }
    Console.WriteLine(person1 == person2); // output: False

    person2 = person1 with { PhoneNumbers = new string[1] };
    Console.WriteLine(person2);
    // output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers
= System.String[] }
    Console.WriteLine(person1 == person2); // output: False

    person2 = person1 with { };
    Console.WriteLine(person1 == person2); // output: True
}
```

Pour plus d'informations, consultez Mutation non destructrice dans l'article de référence du langage C# sur les enregistrements.

Mise en forme intégrée pour l'affichage

Les types d'enregistrements ont une méthode ToString générée par le compilateur qui affiche les noms et les valeurs des propriétés publiques et des champs. La méthode `ToString` retourne une chaîne au format suivant :

> <nom du type d'enregistrement> { <nom de propriété> = <valeur>, <nom de propriété> = <valeur>, ...}

Pour les types de référence, le nom de type de l'objet auquel la propriété fait référence est affiché au lieu de la valeur de propriété. Dans l'exemple suivant, le tableau est un type de référence. `System.String[]` est donc affiché au lieu des valeurs d'élément de tableau réelles :

```
Person { FirstName = Nancy, LastName = Davolio, ChildNames = System.String[]
}
```

Pour plus d'informations, consultez Mise en forme intégrée dans l'article de référence du langage C# sur les enregistrements.

Héritage

Un enregistrement peut hériter d'un autre enregistrement. Toutefois, un enregistrement ne peut pas hériter d'une classe et une classe ne peut pas hériter d'un enregistrement.

L'exemple suivant illustre l'héritage avec la syntaxe de propriété positionnelle :

C#

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio, Grade = 3 }
}
```

Pour que deux variables d'enregistrement soient égales, le type d'exécution doit être égal. Les types des variables conteneur peuvent être différents. Ceci est illustré dans l'exemple de code suivant :

C#

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Person student = new Student("Nancy", "Davolio", 3);
    Console.WriteLine(teacher == student); // output: False

    Student student2 = new Student("Nancy", "Davolio", 3);
    Console.WriteLine(student2 == student); // output: True
}
```

Dans l'exemple, toutes les instances ont les mêmes propriétés et les mêmes valeurs de propriété. Mais `student == teacher` retourne `False` bien que les deux soient des variables de type `Person`. Et `student == student2` retourne `True` bien qu'il s'agit d'une variable `Person` et d'une variable `Student`.

Toutes les propriétés publiques et les champs des types dérivés et de base sont inclus dans la sortie `ToString`, comme illustré dans l'exemple suivant :

C#

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);

public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio, Grade = 3 }
}
```

Pour plus d'informations, consultez Héritage dans l'article de référence sur le langage C# sur les enregistrements.

Setter init uniquement

Les setters init-only fournissent une syntaxe cohérente pour initialiser les membres d'un objet. Les initialiseurs de propriété indiquent clairement quelle valeur définit la propriété. L'inconvénient est que ces propriétés doivent être définies. À compter de C# 9.0, vous pouvez créer des accesseurs `init` au lieu d'accesseurs `set` pour les propriétés et les indexeurs. Les appelants peuvent utiliser la syntaxe d'initialiseur de propriété pour définir ces valeurs dans les expressions de création, mais ces propriétés sont en lecture seule une fois la construction terminée. Les setters init-only fournissent uniquement une fenêtre pour modifier l'état. Cette fenêtre se ferme lorsque la phase de construction est terminée. La phase de construction se termine après toutes les initialisations, y compris celles des initialiseurs de propriétés et des expressions « with ».

Vous pouvez déclarer des setters `init`-only dans n'importe quel type que vous écrivez. Par exemple, la structure suivante définit une structure d'observation météorologique :

C#

```
public struct WeatherObservation
{
    public DateTime RecordedAt { get; init; }
    public decimal TemperatureInCelsius { get; init; }
    public decimal PressureInMillibars { get; init; }

    public override string ToString() =>
        $"At {RecordedAt:h:mm tt} on {RecordedAt:M/d/yyyy}: " +
        $"Temp = {TemperatureInCelsius}, with {PressureInMillibars}
pressure";
}
```

Les appelants peuvent utiliser la syntaxe d'initialiseur de propriété pour définir les valeurs tout en conservant l'immuabilité :

C#

```
var now = new WeatherObservation
{
    RecordedAt = DateTime.Now,
    TemperatureInCelsius = 20,
    PressureInMillibars = 998.0m
};
```

Une tentative de modification d'une observation après l'initialisation entraîne une erreur du compilateur :

C#

```
// Error! CS8852.
now.TemperatureInCelsius = 18;
```

Les setters init-only peuvent être utiles pour définir les propriétés de classe de base à partir de classes dérivées. Ils peuvent également définir des propriétés dérivées par le biais d'assistances dans une classe de base. Les enregistrements positionnels déclarent des propriétés à l'aide de setters init-only. Ces setters sont utilisés dans des expressions. Vous pouvez déclarer des setters init-only pour chaque `class`, `struct` ou `record` que vous définissez.

Pour plus d'informations, consultez init (Référence C#).

Instructions de niveau supérieur

Les instructions de niveau supérieur suppriment les procédés inutile de nombreuses applications. Considérez le programme canonique « Hello World! » :

C#

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Il n'y a qu'une seule ligne de code active. Avec les instructions de niveau supérieur, vous pouvez remplacer toutes ces instructions réutilisables par la directive `using` et la seule ligne active :

C#

```
using System;

Console.WriteLine("Hello World!");
```

Si vous vouliez un programme à ligne unique, vous pouvez supprimer la directive `using` et utiliser le nom de type complet :

```
C#
```

```
System.Console.WriteLine("Hello World!");
```

Un seul fichier de votre application peut utiliser des instructions de niveau supérieur. Si le compilateur trouve des instructions de niveau supérieur dans plusieurs fichiers sources, il s'agit d'une erreur. Il s'agit également d'une erreur si vous combinez des instructions de niveau supérieur avec une méthode de point d'entrée de programme déclarée, généralement une méthode `Main`. Dans un sens, vous pouvez penser qu'un fichier contient les instructions qui seraient normalement dans la méthode `Main` d'une classe `Program`.

L'une des utilisations les plus courantes de cette fonctionnalité consiste à créer des supports d'enseignement. Les développeurs C# débutants peuvent écrire le « Hello World! » canonique dans une ou deux lignes de code. Aucun procédé supplémentaire n'est nécessaire. Toutefois, les développeurs expérimentés trouveront de nombreuses utilisations pour cette fonctionnalité. Les instructions de niveau supérieur permettent une expérience de type script pour l'expérimentation similaire à ce que fournissent les notebooks Jupyter. Les instructions de niveau supérieur sont idéales pour les petits programmes et utilitaires de console. Azure Functions est un cas d'usage idéal pour les instructions de niveau supérieur.

Plus important encore, les instructions de niveau supérieur ne limitent pas l'étendue ou la complexité de votre application. Ces instructions peuvent accéder ou utiliser n'importe quelle classe .NET. Elles ne limitent pas non plus votre utilisation d'arguments de ligne de commande ou de valeurs de retour. Les instructions de niveau supérieur peuvent accéder à un tableau de chaînes nommé `args`. Si les instructions de niveau supérieur retournent une valeur entière, cette valeur devient le code de retour entier d'une méthode `Main` synthétisée. Les instructions de niveau supérieur peuvent contenir des expressions asynchrones. Dans ce cas, le point d'entrée synthétisé retourne `Task`, ou `Task<int>`.

Pour plus d'informations, consultez Instructions de niveau supérieur dans le Guide de programmation C#.

Améliorations des critères spéciaux

C# 9 inclut de nouvelles améliorations de correspondance de modèle :

- ***Les modèles de type*** correspondent à un objet qui correspond à un type particulier

- ***Les modèles entre parenthèses*** appliquent ou mettent en évidence la priorité des combinaisons de modèles
- ***Les modèles conjonctifs*** `and` nécessitent que les deux modèles correspondent
- ***Les modèles disjonctifs*** `or` nécessitent une correspondance entre les deux modèles
- ***Les modèles non validés*** `not` nécessitent qu'un modèle ne corresponde pas
- ***Les modèles relationnels*** nécessitent que l'entrée soit inférieure, supérieure ou égale à une constante donnée ou supérieure ou égale à une constante donnée.

Ces modèles enrichissent la syntaxe des modèles. Prenons les exemples suivants :

C#

```
public static bool IsLetter(this char c) =>
    c is >= 'a' and <= 'z' or >= 'A' and <= 'Z';
```

L'ajout de parenthèses facultatives indique clairement que `and` a une priorité plus élevée que `or` :

C#

```
public static bool IsLetterOrSeparator(this char c) =>
    c is (>= 'a' and <= 'z') or (>= 'A' and <= 'Z') or '.' or ',';
```

L'une des utilisations les plus courantes est une nouvelle syntaxe pour une vérification null :

C#

```
if (e is not null)
{
    // ...
}
```

L'un de ces modèles peut être utilisé dans n'importe quel contexte où les modèles sont autorisés : expressions de modèle `is`, expressions `switch`, modèles imbriqués et modèle d'étiquette `case` d'une instruction `switch`.

Pour plus d'informations, consultez Modèles (référence C#).

Pour plus d'informations, consultez les sections Modèles relationnels et Modèles logiques de l'article Modèles.

Performances et interopérabilité

Trois nouvelles fonctionnalités améliorent la prise en charge de l'interopérabilité native et des bibliothèques de bas niveau qui nécessitent une haute performance : les entiers de taille native, les pointeurs de fonction et l'omission d'indicateur `localsinit`.

Les entiers de taille native, `nint` et `nuint`, sont des types entiers. Ils sont exprimés par les types sous-jacents System.IntPtr et System.UIntPtr. Le compilateur affiche des conversions et des opérations supplémentaires pour ces types en tant qu'entiers natifs. Les entiers de taille native définissent des propriétés pour `MaxValue` ou `MinValue`. Ces valeurs ne peuvent pas être exprimées en tant que constantes au moment de la compilation, car elles dépendent de la taille native d'un entier sur l'ordinateur cible. Ces valeurs sont lues en lecture seule au moment de l'exécution. Vous pouvez utiliser des valeurs constantes pour `nint` sur la plage [`int.MinValue` .. `int.MaxValue`]. Vous pouvez utiliser des valeurs constantes pour `nuint` sur la plage [`uint.MinValue` .. `uint.MaxValue`]. Le compilateur effectue un pliage constant pour tous les opérateurs unaires et binaires à l'aide des types System.Int32 et System.UInt32. Si le résultat ne correspond pas à 32 bits, l'opération est exécutée au moment du runtime et n'est pas considérée comme une constante. Les entiers de taille native peuvent augmenter les performances dans les scénarios où les mathématiques entières sont utilisées en grande partie et doivent avoir les performances les plus rapides possibles. Pour plus d'informations, consultez nint et types nuint.

Les pointeurs de fonction fournissent une syntaxe simple pour accéder aux opcodes IL `ldftn` et `calli`. Vous pouvez déclarer des pointeurs de fonction à l'aide de la nouvelle syntaxe `delegate*`. Un type `delegate*` est un type de pointeur. L'appel du type `delegate*` utilise `calli`, contrairement à un délégué qui utilise `callvirt` sur la méthode `Invoke()`. Sur le plan syntaxique, les appels sont identiques. L'appel de pointeur de fonction utilise la convention d'appel `managed`. Ajoutez le mot clé `unmanaged` après la syntaxe `delegate*` pour déclarer que vous souhaitez la convention d'appel `unmanaged`. D'autres conventions d'appel peuvent être spécifiées à l'aide d'attributs sur la déclaration `delegate*`. Pour plus d'informations, consultez Types de code et de pointeur non sécurisés.

Enfin, vous pouvez ajouter System.Runtime.CompilerServices.SkipLocalsInitAttribute pour indiquer au compilateur de ne pas émettre l'indicateur `localsinit`. Cet indicateur indique au CLR d'effectuer l'initialisation zéro de toutes les variables locales. L'indicateur `localsinit` est le comportement par défaut pour C# depuis la version 1.0. Toutefois, l'initialisation zéro peut avoir un impact mesurable sur les performances dans certains scénarios. Surtout lorsque vous utilisez `stackalloc`. Dans ces cas, vous pouvez ajouter

SkipLocalsInitAttribute. Vous pouvez l'ajouter à une seule méthode ou propriété, ou à `class`, `struct` `interface` ou même à un module. Cet attribut n'affecte pas les méthodes `abstract` ; elle affecte le code généré pour l'implémentation. Pour plus d'informations, consultez Attribut SkipLocalsInit.

Ces fonctionnalités peuvent améliorer les performances dans certains scénarios. Elles ne doivent être utilisées qu'après une évaluation minutieuse avant et après l'adoption. Le code impliquant des entiers de taille native doit être testé sur plusieurs plateformes cibles avec des tailles entières différentes. Les autres fonctionnalités nécessitent du code non sécurisé.

Caractéristiques d'ajustement et de finition

De nombreuses autres fonctionnalités peuvent vous aider à écrire du code plus efficacement. Dans C# 9.0, vous pouvez omettre le type dans une expression new lorsque le type de l'objet créé est déjà connu. L'utilisation la plus courante est dans les déclarations de champ :

```
C#

private List<WeatherObservation> _observations = new();
```

Le type cible `new` peut également être utilisé lorsque vous devez créer un objet à passer en tant qu'argument à une méthode. Imaginez une méthode `ForecastFor()` avec la signature suivante :

```
C#

public WeatherForecast ForecastFor(DateTime forecastDate,
WeatherForecastOptions options)
```

Vous pouvez l'appeler comme suit :

```
C#

var forecast = station.ForecastFor(DateTime.Now.AddDays(2), new());
```

Une autre utilisation intéressante pour cette fonctionnalité consiste à la combiner avec des propriétés init-only pour initialiser un nouvel objet :

```
C#
```

```
WeatherStation station = new() { Location = "Seattle, WA" };
```

Vous pouvez retourner une instance créée par le constructeur par défaut à l'aide d'une instruction `return new();`.

Une fonctionnalité similaire améliore la résolution de type cible des expressions conditionnelles. Avec cette modification, les deux expressions n'ont pas besoin d'une conversion implicite de l'une vers l'autre, mais peuvent avoir des conversions implicites vers un type cible. Vous ne remarquerez probablement pas ce changement. Vous remarquerez cependant que désormais, certaines expressions conditionnelles qui nécessitaient des casts ou qui ne se compilaient pas fonctionnent.

À partir de C# 9.0, vous pouvez ajouter le modificateur `static` à des expressions lambda ou à des méthodes anonymes. Les expressions lambda statiques sont analogues aux fonctions locales `static` : une méthode lambda statique ou anonyme ne peut pas capturer les variables locales ou l'état de l'instance. Le modificateur `static` empêche la capture accidentelle d'autres variables.

Les types de retour covariant offrent une flexibilité pour les types de retour de méthodes de remplacement. Une méthode de remplacement peut retourner un type dérivé depuis un type de retour de la méthode de base substituée. Cela peut être utile pour les enregistrements et pour d'autres types qui prennent en charge les méthodes de clone virtuel ou de fabrique.

En outre, la boucle foreach reconnaît et utilise une méthode d'extension `GetEnumerator` alternative pour le modèle `foreach`. Cette modification signifie que `foreach` est cohérent avec d'autres constructions basées sur des modèles, telles que le modèle asynchrone et la déconstruction basée sur des modèles. En pratique, cette modification signifie que vous pouvez ajouter la prise en charge `foreach` à n'importe quel type. Vous devez limiter son utilisation lors de l'énumération d'un objet dans votre conception.

Vous pouvez ensuite utiliser des abandons en tant que paramètres pour les expressions lambda. Cette commodité vous permet d'éviter de nommer l'argument, et le compilateur peut éviter de l'utiliser. Utilisez `_` pour n'importe quel argument. Pour plus d'informations, consultez la section Paramètres d'entrée d'une expression lambda de l'article Expressions lambda.

Vous pouvez désormais appliquer des attributs aux fonctions locales. Par exemple, vous pouvez appliquer des annotations d'attribut nullables aux fonctions locales.

Prise en charge des générateurs de code

Deux fonctionnalités finales prennent en charge les générateurs de code C#. Les générateurs de code C# sont un composant que vous pouvez écrire. Il est similaire à un analyseur Roslyn ou à un correctif de code. La différence est que les générateurs de code analysent le code et écrivent de nouveaux fichiers de code source dans le cadre du processus de compilation. Un générateur de code classique recherche du code pour les attributs ou autres conventions.

Un générateur de code lit des attributs ou autres éléments de code à l'aide des API d'analyse Roslyn. À partir de ces informations, du nouveau code est ajouté à la compilation. Les générateurs sources peuvent uniquement ajouter du code ; ils ne sont pas autorisés à modifier du code existant dans la compilation.

Les deux fonctionnalités ajoutées pour les générateurs de code sont des extensions à la ***syntaxe de méthode partielle*** et aux ***initialiseurs de module***. Tout d'abord, les modifications apportées aux méthodes partielles. Avant C# 9.0, les méthodes partielles sont `private` mais ne peuvent pas spécifier de modificateur d'accès, avoir un retour `void` et ne peuvent pas avoir de paramètres `out`. Ces restrictions signifient que si aucune implémentation de méthode n'est fournie, le compilateur supprime tous les appels à la méthode partielle. C# 9.0 supprime ces restrictions, mais nécessite que les déclarations de méthode partielles aient une implémentation. Les générateurs de code peuvent fournir cette implémentation. Pour éviter d'introduire un changement cassant, le compilateur considère toute méthode partielle sans modificateur d'accès pour suivre les anciennes règles. Si la méthode partielle inclut le modificateur d'accès `private`, les nouvelles règles régissent cette méthode partielle. Pour plus d'informations, consultez Méthode partielles (référence C#).

La deuxième nouvelle fonctionnalité pour les générateurs de code sont ***les initialiseurs de module***. Les initialiseurs de module sont des méthodes qui ont l'attribut ModuleInitializerAttribute attaché à eux. Ces méthodes seront appelées par le runtime avant tout autre appel de champ ou appel de méthode dans l'ensemble du module. Méthode d'initialiseur de module :

- Doit être statique
- Doit être sans paramètre
- Doit retourner void
- Ne doit pas être une méthode générique
- Ne doit pas être contenu dans une classe générique
- Doit être accessible à partir du module conteneur

Ce dernier point de puce signifie que la méthode et sa classe conteneur doivent être internes ou publiques. La méthode ne peut pas être une fonction locale. Pour plus d'informations, consultez Attribut ModuleInitializer.

Histoire du langage C#

Article • 09/03/2023

Cet article fournit un historique de chaque version majeure du langage C#. L'équipe C# continue d'innover et d'ajouter de nouvelles fonctionnalités. L'état détaillé des fonctionnalités de langage, notamment celles prises en compte pour les versions à venir, est disponible dans le dépôt dotnet/roslyn sur GitHub.

> **Important**
>
> Le langage C# s'appuie sur des types et des méthodes dans ce que la spécification C# définit comme une *bibliothèque standard* pour certaines fonctionnalités. La plateforme .NET fournit ces types et ces méthodes dans des packages. Par exemple, le traitement des exceptions. Chaque instruction ou expression `throw` est vérifiée pour s'assurer de l'objet levé est dérivé de **Exception**. De même, chaque `catch` est vérifié pour assurer que le type intercepté est dérivé de **Exception**. Chaque version peut ajouter de nouvelles spécifications. Pour utiliser les dernières fonctionnalités de langage dans les environnements plus anciens, vous devrez peut-être installer des bibliothèques spécifiques. Ces dépendances sont décrites dans la page pour chaque version spécifique. Pour en savoir plus, consultez les **relations entre le langage et la bibliothèque** pour l'arrière-plan sur cette dépendance.

C# version 11

Publication : novembre 2022

Les fonctionnalités suivantes ont été ajoutées dans C# 11 :

- Littéraux de chaîne bruts
- Prise en charge des mathématiques génériques
- Attributs génériques
- Littéraux de chaîne UTF-8
- Nouvelles lignes dans les expressions d'interpolation de chaîne
- Modèles de liste
- Types locaux de fichiers
- Membres nécessaires
- Structs par défaut automatiques
- Correspondance Span<char> de modèle sur une constante string
- Portée de nameof étendue

- IntPtr numérique
- Champs ref et scoped ref
- Amélioration de la conversion de groupe de méthodes en instance déléguée
- Vague d'avertissement 7

C# 11 introduit des *mathématiques génériques* et plusieurs fonctionnalités qui prennent en charge cet objectif. Vous pouvez écrire des algorithmes numériques une fois pour tous les types de nombres. Il existe d'autres fonctionnalités pour faciliter l'utilisation des types `struct`, comme les membres requis et les structs par défaut automatique. L'utilisation de chaînes est plus facile avec les littéraux de chaîne bruts, la nouvelle ligne dans les interpolations de chaîne et les littéraux de chaîne UTF-8. Les fonctionnalités telles que les types locaux de fichiers permettent aux générateurs sources d'être plus simples. Enfin, les modèles de liste ajoutent davantage de prise en charge pour la mise en correspondance des modèles.

C# version 10

Publication : novembre 2021

C# 10 ajoute les fonctionnalités et améliorations suivantes au langage C# :

- Structs d'enregistrement
- Améliorations des types de structure
- Gestionnaires de chaînes interpolés
- Directives global using
- Déclaration d'espace de noms dans l'étendue du fichier
- Modèles de propriétés étendus
- Améliorations apportées aux expressions lambda
- Autoriser les chaînes interpolées const
- Les types d'enregistrements peuvent sceller ToString()
- Amélioration de l'affectation définitive
- Autoriser à la fois l'affectation et la déclaration dans la même déconstruction
- Autoriser l'attribut AsyncMethodBuilder sur les méthodes
- Attribut CallerArgumentExpression
- Pragma #line amélioré

D'autres fonctionnalités étaient disponibles en mode *préversion*. Pour utiliser ces fonctionnalités, vous devez définir <LangVersion> sur Preview dans votre projet :

- Attributs génériques plus loin dans cet article.
- membres abstraits statiques dans les interfaces

C# 10 continue de travailler sur des thèmes de suppression de cérémonie, de séparation des données à partir d'algorithmes et d'amélioration des performances pour le runtime .NET.

La plupart des fonctionnalités impliquent que vous tapez moins de code pour exprimer les mêmes concepts. *Les structs d'enregistrement* synthétisent la plupart des mêmes méthodes que les *classes d'enregistrement*. Les structs et les types anonymes prennent en charge *les expressions*. *Directives à utilisation globale* et *déclarations d'espace de noms délimitées par des fichiers* signifient que vous exprimez plus clairement les dépendances et l'organisation de l'espace de noms. *Les améliorations lambda* facilitent la déclaration d'expressions lambda où elles sont utilisées. Les nouveaux modèles de propriétés et les améliorations de la déconstruction créent du code plus concis.

Les nouveaux gestionnaires de chaînes interpolés et le comportement `AsyncMethodBuilder` peuvent améliorer les performances. Ces fonctionnalités de langage ont été appliquées dans le runtime .NET pour obtenir des améliorations des performances dans .NET 6.

C# 10 marque également plus d'un passage à la cadence annuelle pour les versions de .NET. Étant donné que toutes les fonctionnalités ne peuvent pas être effectuées sur une période annuelle, vous pouvez essayer quelques fonctionnalités « préversion » en C# 10. Les *attributs génériques* et les *membres abstraits statiques dans les interfaces* peuvent être utilisés, mais ces fonctionnalités en préversion peuvent changer avant leur version finale.

C# version 9

Publication : novembre 2020

C# 9 a été publié avec .NET 5. Il s'agit de la version du langage par défaut pour n'importe quel assembly qui cible la version .NET 5. Il contient les fonctionnalités nouvelles et améliorées suivantes :

- Enregistrements
- Setter init uniquement
- Instructions de niveau supérieur
- Améliorations des critères spéciaux
- Performances et interopérabilité
  - Entiers dimensionnés natifs
  - Pointeurs fonction
  - Suppression de l'indicateur localsinit émis
- Caractéristiques d'ajustement et de finalisation

    - Expressions new typées cible
    - static fonctions anonymes
    - Expressions conditionnelles typées cible
    - Types de retour covariant
    - Prise en charge des extensions GetEnumerator pour les boucles foreach
    - Paramètres d'abandon lambda
    - Attributs sur des fonctions locales
- Prise en charge des générateurs de code
    - Initialiseurs de module
    - Nouvelles fonctionnalités pour les méthodes partielles

C# 9 poursuit trois des thèmes des versions précédentes : la suppression de cérémonie, la séparation des données des algorithmes et la fourniture de modèles supplémentaires dans d'autres endroits.

Les instructions de niveau supérieur signifient que votre programme principal est plus simple à lire. Le besoin de cérémonie est moins important : un espace de noms, une classe `Program` et `static void Main()` sont tous inutiles.

L'introduction de records fournit une syntaxe concise pour les types de référence qui suivent la sémantique des valeurs pour l'égalité. Vous utiliserez ces types pour définir des conteneurs de données qui définissent généralement un comportement minimal. Les setters init uniquement offrent la possibilité de mutation non destructrice (expressions `with`) dans les enregistrements. C# 9 ajoute également des types de retour covariants afin que les enregistrements dérivés puissent remplacer les méthodes virtuelles et retourner un type dérivé du type de retour de la méthode de base.

Les fonctionnalités de correspondance des modèles ont été développées de plusieurs façons. Les types numériques prennent désormais en charge *les modèles de plage*. Les modèles peuvent être combinés à l'aide de modèles `and`, `or` et `not`. Des parenthèses peuvent être ajoutées pour clarifier des modèles plus complexes.

Un autre ensemble de fonctionnalités prend en charge le calcul hautes performances en C# :

- Les types `nint` et `nuint` modélisent les types entiers de taille native sur l'UC cible.
- Les pointeurs de fonction fournissent des fonctionnalités de type délégué tout en évitant les allocations nécessaires pour créer un objet délégué.
- Vous pouvez omettre l'instruction `localsinit` pour enregistrer des instructions.

Un autre ensemble d'améliorations prend en charge des scénarios où les *générateurs de code* ajoutent des fonctionnalités :

- Les initialiseurs de module sont des méthodes que le runtime appelle lorsqu'un assembly se charge.
- Les méthodes partielles prennent en charge les nouveaux modificateurs accessibly et les types de retour non void. Dans ce cas, une implémentation doit être fournie.

C# 9 ajoute de nombreuses autres petites fonctionnalités qui améliorent la productivité des développeurs, à la fois pour l'écriture et la lecture du code :

- Expressions `new` de type cible
- `static` fonctions anonymes
- Expressions conditionnelles de type cible
- Prise en charge des extensions `GetEnumerator()` pour les boucles `foreach`
- Les expressions lambda peuvent déclarer des paramètres d'abandon
- Les attributs peuvent être appliqués aux fonctions locales

La version C# 9 poursuit le travail de maintien d'un langage de programmation moderne et universel. Les fonctionnalités continuent de prendre en charge les charges de travail et les types d'applications modernes.

C# version 8.0

Date de publication : septembre 2019

C# 8.0 est la première version principale de C# qui cible spécifiquement .NET Core. Certaines fonctionnalités s'appuient sur de nouvelles fonctionnalités CLR, d'autres sur les types de bibliothèque ajoutés uniquement dans .NET Core. C# 8.0 ajoute les fonctionnalités et améliorations suivantes au langage C# :

- Membres ReadOnly
- Méthodes d'interface par défaut
- Amélioration des critères spéciaux :
  - Expressions switch
  - Modèles de propriétés
  - Modèles de tuples
  - Modèles positionnels
- Déclarations using
- Fonctions locales statiques
- Structs ref jetables
- Types références Nullables
- Flux asynchrones
- Index et plages
- Attribution de fusion Null

- Types construits non managés
- Stackalloc dans des expressions imbriquées
- Amélioration des chaînes verbatim interpolées

Les membres de l'interface par défaut nécessitent des améliorations dans le CLR. Ces fonctionnalités ont été ajoutées dans le CLR pour .NET Core 3.0. Les plages et les index, ainsi que les flux asynchrones nécessitent de nouveaux types dans les bibliothèques .NET Core 3.0. Les types de référence pouvant accepter la valeur Nul, tout en étant implémentés dans le compilateur, sont beaucoup plus utiles lorsque les bibliothèques sont annotées pour fournir des informations sémantiques sur l'état null des arguments et des valeurs de retour. Ces annotations sont ajoutées dans les bibliothèques .NET Core.

C# version 7.3

Publication : mai 2018

Il existe deux thèmes principaux pour la version C# 7.3. Un thème fournit des fonctionnalités permettant au code sécurisé d'être aussi performant que le code non sécurisé. Le second thème fournit des améliorations incrémentielles aux fonctionnalités existantes. De nouvelles options de compilateur ont également été ajoutées dans cette version.

Les nouvelles fonctionnalités suivantes prennent en charge le thème de meilleures performances pour le code sécurisé :

- Vous pouvez accéder à des champs fixes sans épinglage.
- Vous pouvez réaffecter des variables locales `ref`.
- Vous pouvez utiliser des initialiseurs sur les tableaux `stackalloc`.
- Vous pouvez utiliser des instructions `fixed` avec tout type prenant en charge un modèle.
- Vous pouvez utiliser des contraintes plus génériques.

Les améliorations suivantes ont été apportées aux fonctionnalités existantes :

- Vous pouvez tester `==` et `!=` avec des types de tuple.
- Vous pouvez utiliser des variables d'expression dans d'autres emplacements.
- Vous pouvez joindre des attributs à un champ de stockage de propriétés implémentées automatiquement.
- La résolution de la méthode lorsque des arguments diffèrent de `in` a été améliorée.
- La résolution de la surcharge comporte maintenant moins de cas ambigus.

Les nouvelles options du compilateur sont les suivantes :

- -publicsign pour activer la signature d'assemblys Open Source Software (OSS).
- -pathmap pour fournir un mappage des répertoires sources.

C# version 7.2

Publication : novembre 2017

C# 7.2 a ajouté plusieurs fonctionnalités de langage de petite taille :

- Initialiseurs sur des tableaux `stackalloc`.
- Utilisez des instructions `fixed` avec n'importe quel type qui prend en charge un modèle.
- Accéder aux champs fixes sans épinglage.
- Réaffecter des variables `ref` locales.
- Déclarez des types `readonly struct` pour indiquer qu'un struct est immuable et doit être transmis en tant que paramètre `in` à ses méthodes membres.
- Ajoutez le modificateur `in` aux paramètres pour spécifier qu'un argument est passé par référence, mais qu'il n'est pas modifié par la méthode appelée.
- Utilisez le modificateur `ref readonly` sur les retours de méthode, pour indiquer qu'une méthode retourne sa valeur par référence, mais n'autorise pas les écritures dans cet objet.
- Déclarez des types `ref struct`, pour indiquer qu'un type de struct accède directement à la mémoire managée et doit toujours être alloué à la pile.
- Utilisez des contraintes génériques supplémentaires.
- Arguments nommés non placés en position de fin
  - Les arguments nommés peuvent être suivis par des arguments de position.
- Traits de soulignement de début dans les littéraux numériques
  - Les littéraux numériques peuvent maintenant comporter des traits de soulignement de début avant tout chiffre affiché.
- Modificateur d'accès private protected
  - Le modificateur d'accès `private protected` active l'accès pour les classes dérivées dans le même assembly.
- Expressions `ref` conditionnelles
  - Le résultat d'une expression conditionnelle (`?:`) peut maintenant être une référence.

C# version 7.1

Publication : août 2017

C# a commencé à sortir *versions de point* avec C# 7.1. Cette version a ajouté l'élément de configuration de sélection de version de langue, trois nouvelles fonctionnalités de langage et un nouveau comportement du compilateur.

Les nouvelles fonctionnalités de langage de cette version sont :

- Méthode asyncMain
  - Le point d'entrée pour une application peut avoir le modificateur `async`.
- Expressions littérales default
  - Vous pouvez utiliser des expressions littérales default dans les expressions de valeur par défaut quand le type cible peut être inféré.
- Noms des éléments de tuple inférés
  - Les noms des éléments de tuple peuvent être inférés dans de nombreux cas à partir de l'initialisation du tuple.
- Critères spéciaux sur les paramètres de type générique
  - Il est possible d'utiliser des expressions de critères spéciaux sur les variables dont le type est un paramètre de type générique.

Enfin, le compilateur a deux options, -refout et -refonly, qui contrôlent la génération d'assemblys de références

C# version 7.0

Publication : mars 2017

C# version 7.0 a été publié avec Visual Studio 2017. Cette version présente des points évolutifs et très attrayants dans la veine de C# 6.0. Voici quelques-unes des nouvelles fonctionnalités :

- Variables out
- Tuples et déconstruction
- Critères spéciaux
- Fonctions locales
- Membres expression-bodied étendus
- Variables locales ref
- Retours par référence

Autres fonctionnalités disponibles :

- Éléments ignorés
- Littéraux binaires et séparateurs numériques

- Expressions throw

Toutes ces fonctionnalités offrent de nouvelles capacités aux développeurs ainsi que la possibilité d'écrire du code plus propre. Il s'agit notamment de condenser la déclaration des variables à utiliser avec le mot clé `out` et en autorisant plusieurs valeurs de retour par le biais d'un tuple. .NET Core cible à présent n'importe quel système d'exploitation et garde les yeux rivés sur le cloud et la portabilité. Ces nouvelles fonctions occupent sans aucun doute les pensées des concepteurs du langage, en plus des fonctionnalités à venir.

C# version 6.0

Publication : juillet 2015

La version 6.0, sortie avec Visual Studio 2015, proposait de nombreuses fonctionnalités plus petites qui rendent la programmation C# plus productive. En voici quelques-unes :

- Importations statiques
- Filtres d'exceptions
- Initialiseurs de propriétés automatiques
- Membres expression-bodied
- Propagateur Null
- Interpolation de chaîne
- Opérateur nameof

Quelques autres nouvelles fonctions :

- Initialiseurs d'index
- Await dans des blocs catch/finally
- Valeurs par défaut pour les propriétés d'accesseur Get

Chacune de ces fonctionnalités est intéressante individuellement. Mais si vous les examinez dans leur ensemble, un modèle intéressant se dégage. Dans cette version, C# a commencé à éliminer le texte réutilisable du langage pour rendre le code plus laconique et plus lisible. Ainsi, pour les amateurs de code propre et simple, cette version du langage était une énorme victoire.

Une autre nouveauté a été proposée avec cette version, même s'il ne s'agit pas d'une fonctionnalité de langage traditionnelle à proprement dit. Le compilateur Roslyn a été publié en tant que service. Le compilateur C# est à présent écrit en C# et vous pouvez l'utiliser dans le cadre de vos efforts de programmation.

C# version 5.0

Publication : août 2012

C# version 5.0, sortie avec Visual Studio 2012, était une version ciblée du langage. Presque tous les efforts déployés pour cette version portaient sur un autre concept révolutionnaire du langage : le modèle `async` et `await` pour la programmation asynchrone. Voici la liste des principales fonctionnalités :

- Membres asynchrones
- Attributs d'informations de l'appelant
- Projet de code : Attributs des informations de l'appelant en C# 5.0

L'attribut d'informations de l'appelant vous permet de récupérer facilement des informations sur le contexte d'exécution sans avoir recours à une multitude de code de réflexion réutilisable. Nombre de ses usages ont trait aux diagnostics et aux tâches de journalisation.

Mais `async` et `await` sont les véritables vedettes de cette version. Quand ces fonctionnalités sont apparues en 2012, C# a encore modifié la donne en intégrant l'asynchronie en tant que participant de premier ordre dans le langage. Si vous avez déjà eu affaire à de longues opérations et à l'implémentation de rappels, vous avez probablement adoré cette fonctionnalité de langage.

C# version 4.0

Publication : avril 2010

Il fut difficile pour C# version 4.0, sortie avec Visual Studio 2010, d'être à la hauteur du concept révolutionnaire incarné par la version 3.0. Cette version introduisit de nouvelles fonctionnalités intéressantes :

- Liaison dynamique
- Arguments nommés/facultatifs
- Covariance et contravariance génériques
- Types interop incorporés

Les types d'interopérabilité incorporés ont facilité le déploiement de la création d'assemblys COM Interop pour votre application. La covariance et la contravariance génériques vous donnent plus de contrôle sur l'utilisation des génériques, mais elles sont un peu académiques et probablement plus appréciées des auteurs de frameworks et de bibliothèques. Les paramètres nommés et facultatifs vous permettent d'éliminer

les nombreuses surcharges de méthode et s'avèrent plus pratiques. Mais aucune de ces fonctionnalités ne représente un vrai changement de paradigme.

La fonctionnalité majeure était plutôt l'introduction du mot clé `dynamic`. Le mot clé `dynamic` introduisit dans C# version 4.0 la possibilité de remplacer le compilateur lors de la saisie au moment de la compilation. En utilisant le mot clé dynamic, vous pouvez créer des constructions semblables aux langages dynamiquement typés comme JavaScript. Vous pouvez créer un `dynamic x = "a string"`, puis y ajouter six, et ainsi laisser le runtime se débrouiller avec ce qui doit se produire par la suite.

La liaison dynamique offre la possibilité de commettre des erreurs, mais également de contrôler davantage le langage.

C# version 3.0

Publication : novembre 2007

C# version 3.0 est apparu fin 2007, en même temps que Visual Studio 2008, même si l'éventail complet des fonctionnalités du langage n'a réellement voir le jour qu'avec le .NET Framework version 3.5. Cette version a insufflé un changement majeur dans l'évolution de C#. Elle a imposé C# en tant que langage de programmation vraiment formidable. Examinons certaines fonctionnalités importantes dans cette version :

- Propriétés implémentées automatiquement
- Types anonymes
- Expressions de requête
- Expressions lambda
- Arborescences de l'expression
- Méthodes d'extension
- Variables locales implicitement typées
- Méthodes partielles
- Initialiseurs d'objets et de collections

Rétrospectivement, nombre de ces fonctionnalités semblent à la fois inéluctables et inséparables. Elles s'assemblent de façon stratégique. L'expression de requête, également appelée LINQ (Language-Integrated Query), était considérée comme la fonctionnalité remarquable de cette version de C#.

Un point de vue plus nuancé s'intéresse aux arborescences d'expressions, aux expressions lambda et aux types anonymes sur lesquels se construit LINQ. Mais, dans les deux cas, C# 3.0 présentait un concept révolutionnaire. C# 3.0 jetait les bases qui allaient permettre à C# de devenir un langage fonctionnel/orienté objet hybride.

Plus précisément, il devint alors possible d'écrire des requêtes déclaratives de style SQL pour effectuer des opérations sur des collections, entre autres choses. Au lieu d'écrire une boucle `for` pour calculer la moyenne d'une liste d'entiers, vous pouviez alors simplement utiliser `list.Average()`. La combinaison des expressions de requête et des méthodes d'extension a permis à de telles listes d'entiers de devenir beaucoup plus intelligentes.

C# version 2.0

Publication : novembre 2005

Examinons certaines fonctionnalités majeures de C# 2.0, sorti en 2005, en même temps que Visual Studio 2005 :

- Génériques
- Types partiels
- Méthodes anonymes
- Types valeur Nullable
- Itérateurs
- Covariance et contravariance

D'autres fonctionnalités de C# 2.0 ajoutaient des capacités aux fonctionnalités existantes :

- Accessibilité distincte des accesseurs Get/Set
- Conversions de groupes de méthodes (délégués)
- Classes statiques
- Inférence de délégué

Alors que C# avait démarré en tant que langage orienté objet générique, C# version 2.0 a rapidement changé tout ça. Avec les génériques, les types et les méthodes peuvent fonctionner sur un type arbitraire tout en assurant quand même la cohérence des types. À titre d'exemple, un List<T> vous permet d'avoir `List<string>` ou `List<int>` et d'effectuer des opérations qui maintiennent la cohérence des types sur des chaînes ou entiers alors que vous itérez en leur sein. Il vaut mieux utiliser des génériques que créer un type `ListInt` qui dérive de `ArrayList` ou effectuer un cast de `Object` pour chaque opération.

C# version 2.0 a introduit les itérateurs. En bref, les itérateurs permettent d'examiner tous les éléments dans un `List` (ou d'autres types énumérables) avec une boucle `foreach`. Le fait de disposer d'itérateurs comme composants de premier ordre du

langage en a considérablement amélioré la lisibilité et la capacité de raisonnement des utilisateurs vis-à-vis du code.

Pourtant, C# était toujours en train de courir derrière Java. Java avait déjà publié des versions qui incluaient des génériques et des itérateurs. Mais cela allait bientôt changer quand les deux langages poursuivraient des chemins différents.

C# version 1.2

Publication : avril 2003

C# version 1.2 était fourni avec Visual Studio .NET 2003. Cette version contenait quelques améliorations mineures du langage. La principale est que, à compter de cette version, le code était généré dans une boucle `foreach` (appelée Dispose) sur un IEnumerator quand ce IEnumerator implémentait IDisposable.

C# version 1.0

Publication : janvier 2002

Rétrospectivement, le langage C# version 1.0, sortie avec Visual Studio .NET 2002, ressemblait beaucoup à Java. Dans ses objectifs de conception énoncés pour ECMA, il cherchait à être un « langage orienté objet simple, moderne et généraliste ». À l'époque, ressembler à Java signifiait qu'il avait atteint ces objectifs de conception.

Mais si vous repensez à C# 1.0 maintenant, cela peut vous donner le vertige. Il lui manquait des fonctionnalités asynchrones intégrées et certaines des fonctionnalités liées aux génériques qui sont aujourd'hui pour vous une évidence. En fait, il ne proposait pas du tout de génériques. Et LINQ ? Pas encore disponible. Il fallait attendre encore plusieurs années.

C# version 1.0 semblait dénué de fonctionnalités, par rapport à aujourd'hui. Vous vous retrouviez à écrire plus ou moins du code détaillé. Mais il faut bien commencer quelque part. C# version 1.0 était une alternative viable à Java sur la plateforme Windows.

Les principales fonctionnalités du langage C# 1.0 étaient les suivantes :

- Classes
- Structures
- Interfaces
- Événements
- Propriétés

- Délégués
- Opérateurs et expressions
- Instructions
- Attributs

Article[initialement publié sur le blog NDepend], avec l'aimable autorisation d'Erik Dietrich et de Patrick Smacchia.

Découvrir les changements cassants dans le compilateur C#

Article • 10/05/2023

Vous trouverez les changements cassants depuis la version C# 10 ici.

L'équipe Roslyn tient à jour une liste des changements cassants dans les compilateurs C# et Visual Basic. Vous trouverez des informations sur ces changements en suivant ces liens sur leur dépôt GitHub :

- Changements cassants dans Roslyn dans C# 10.0/.NET 6
- Changements cassants dans Roslyn après .NET 5
- Changements cassants dans VS2019 version 16.8 introduits dans .NET 5 et C# 9.0
- Changements cassants dans VS2019 Update 1 et ultérieur par rapport à VS2019
- Changements cassants depuis VS2017 (C# 7)
- Changements cassants dans Roslyn 3.0 (VS2019) par rapport à Roslyn 2.* (VS2017)
- Changements cassants dans Roslyn 2.0 (VS2017) à partir de Roslyn 1.* (VS2015) et du compilateur C# natif (VS2013 et antérieur).
- Changements cassants dans Roslyn 1.0 (VS2015) à partir du compilateur C# natif (VS2013 et antérieur).
- Changement de version Unicode dans C# 6

Histoire du langage C#

Article • 09/03/2023

Cet article fournit un historique de chaque version majeure du langage C#. L'équipe C# continue d'innover et d'ajouter de nouvelles fonctionnalités. L'état détaillé des fonctionnalités de langage, notamment celles prises en compte pour les versions à venir, est disponible dans le dépôt dotnet/roslyn sur GitHub.

> **Important**
>
> Le langage C# s'appuie sur des types et des méthodes dans ce que la spécification C# définit comme une *bibliothèque standard* pour certaines fonctionnalités. La plateforme .NET fournit ces types et ces méthodes dans des packages. Par exemple, le traitement des exceptions. Chaque instruction ou expression `throw` est vérifiée pour s'assurer de l'objet levé est dérivé de **Exception**. De même, chaque `catch` est vérifié pour assurer que le type intercepté est dérivé de **Exception**. Chaque version peut ajouter de nouvelles spécifications. Pour utiliser les dernières fonctionnalités de langage dans les environnements plus anciens, vous devrez peut-être installer des bibliothèques spécifiques. Ces dépendances sont décrites dans la page pour chaque version spécifique. Pour en savoir plus, consultez les **relations entre le langage et la bibliothèque** pour l'arrière-plan sur cette dépendance.

C# version 11

Publication : novembre 2022

Les fonctionnalités suivantes ont été ajoutées dans C# 11 :

- Littéraux de chaîne bruts
- Prise en charge des mathématiques génériques
- Attributs génériques
- Littéraux de chaîne UTF-8
- Nouvelles lignes dans les expressions d'interpolation de chaîne
- Modèles de liste
- Types locaux de fichiers
- Membres nécessaires
- Structs par défaut automatiques
- Correspondance Span<char> de modèle sur une constante string
- Portée de nameof étendue

- IntPtr numérique
- Champs ref et scoped ref
- Amélioration de la conversion de groupe de méthodes en instance déléguée
- Vague d'avertissement 7

C# 11 introduit des *mathématiques génériques* et plusieurs fonctionnalités qui prennent en charge cet objectif. Vous pouvez écrire des algorithmes numériques une fois pour tous les types de nombres. Il existe d'autres fonctionnalités pour faciliter l'utilisation des types `struct`, comme les membres requis et les structs par défaut automatique. L'utilisation de chaînes est plus facile avec les littéraux de chaîne bruts, la nouvelle ligne dans les interpolations de chaîne et les littéraux de chaîne UTF-8. Les fonctionnalités telles que les types locaux de fichiers permettent aux générateurs sources d'être plus simples. Enfin, les modèles de liste ajoutent davantage de prise en charge pour la mise en correspondance des modèles.

C# version 10

Publication : novembre 2021

C# 10 ajoute les fonctionnalités et améliorations suivantes au langage C# :

- Structs d'enregistrement
- Améliorations des types de structure
- Gestionnaires de chaînes interpolés
- Directives global using
- Déclaration d'espace de noms dans l'étendue du fichier
- Modèles de propriétés étendus
- Améliorations apportées aux expressions lambda
- Autoriser les chaînes interpolées const
- Les types d'enregistrements peuvent sceller ToString()
- Amélioration de l'affectation définitive
- Autoriser à la fois l'affectation et la déclaration dans la même déconstruction
- Autoriser l'attribut AsyncMethodBuilder sur les méthodes
- Attribut CallerArgumentExpression
- Pragma #line amélioré

D'autres fonctionnalités étaient disponibles en mode *préversion*. Pour utiliser ces fonctionnalités, vous devez définir <LangVersion> sur Preview dans votre projet :

- Attributs génériques plus loin dans cet article.
- membres abstraits statiques dans les interfaces

C# 10 continue de travailler sur des thèmes de suppression de cérémonie, de séparation des données à partir d'algorithmes et d'amélioration des performances pour le runtime .NET.

La plupart des fonctionnalités impliquent que vous tapez moins de code pour exprimer les mêmes concepts. *Les structs d'enregistrement* synthétisent la plupart des mêmes méthodes que les *classes d'enregistrement*. Les structs et les types anonymes prennent en charge *les expressions*. *Directives à utilisation globale* et *déclarations d'espace de noms délimitées par des fichiers* signifient que vous exprimez plus clairement les dépendances et l'organisation de l'espace de noms. *Les améliorations lambda* facilitent la déclaration d'expressions lambda où elles sont utilisées. Les nouveaux modèles de propriétés et les améliorations de la déconstruction créent du code plus concis.

Les nouveaux gestionnaires de chaînes interpolés et le comportement `AsyncMethodBuilder` peuvent améliorer les performances. Ces fonctionnalités de langage ont été appliquées dans le runtime .NET pour obtenir des améliorations des performances dans .NET 6.

C# 10 marque également plus d'un passage à la cadence annuelle pour les versions de .NET. Étant donné que toutes les fonctionnalités ne peuvent pas être effectuées sur une période annuelle, vous pouvez essayer quelques fonctionnalités « préversion » en C# 10. Les *attributs génériques* et les *membres abstraits statiques dans les interfaces* peuvent être utilisés, mais ces fonctionnalités en préversion peuvent changer avant leur version finale.

C# version 9

Publication : novembre 2020

C# 9 a été publié avec .NET 5. Il s'agit de la version du langage par défaut pour n'importe quel assembly qui cible la version .NET 5. Il contient les fonctionnalités nouvelles et améliorées suivantes :

- Enregistrements
- Setter init uniquement
- Instructions de niveau supérieur
- Améliorations des critères spéciaux
- Performances et interopérabilité
  - Entiers dimensionnés natifs
  - Pointeurs fonction
  - Suppression de l'indicateur localsinit émis
- Caractéristiques d'ajustement et de finalisation

    - Expressions new typées cible
    - static fonctions anonymes
    - Expressions conditionnelles typées cible
    - Types de retour covariant
    - Prise en charge des extensions GetEnumerator pour les boucles foreach
    - Paramètres d'abandon lambda
    - Attributs sur des fonctions locales
- Prise en charge des générateurs de code
    - Initialiseurs de module
    - Nouvelles fonctionnalités pour les méthodes partielles

C# 9 poursuit trois des thèmes des versions précédentes : la suppression de cérémonie, la séparation des données des algorithmes et la fourniture de modèles supplémentaires dans d'autres endroits.

Les instructions de niveau supérieur signifient que votre programme principal est plus simple à lire. Le besoin de cérémonie est moins important : un espace de noms, une classe `Program` et `static void Main()` sont tous inutiles.

L'introduction de records fournit une syntaxe concise pour les types de référence qui suivent la sémantique des valeurs pour l'égalité. Vous utiliserez ces types pour définir des conteneurs de données qui définissent généralement un comportement minimal. Les setters init uniquement offrent la possibilité de mutation non destructrice (expressions `with`) dans les enregistrements. C# 9 ajoute également des types de retour covariants afin que les enregistrements dérivés puissent remplacer les méthodes virtuelles et retourner un type dérivé du type de retour de la méthode de base.

Les fonctionnalités de correspondance des modèles ont été développées de plusieurs façons. Les types numériques prennent désormais en charge *les modèles de plage*. Les modèles peuvent être combinés à l'aide de modèles `and`, `or` et `not`. Des parenthèses peuvent être ajoutées pour clarifier des modèles plus complexes.

Un autre ensemble de fonctionnalités prend en charge le calcul hautes performances en C# :

- Les types `nint` et `nuint` modélisent les types entiers de taille native sur l'UC cible.
- Les pointeurs de fonction fournissent des fonctionnalités de type délégué tout en évitant les allocations nécessaires pour créer un objet délégué.
- Vous pouvez omettre l'instruction `localsinit` pour enregistrer des instructions.

Un autre ensemble d'améliorations prend en charge des scénarios où les *générateurs de code* ajoutent des fonctionnalités :

- Les initialiseurs de module sont des méthodes que le runtime appelle lorsqu'un assembly se charge.
- Les méthodes partielles prennent en charge les nouveaux modificateurs accessibly et les types de retour non void. Dans ce cas, une implémentation doit être fournie.

C# 9 ajoute de nombreuses autres petites fonctionnalités qui améliorent la productivité des développeurs, à la fois pour l'écriture et la lecture du code :

- Expressions `new` de type cible
- `static` fonctions anonymes
- Expressions conditionnelles de type cible
- Prise en charge des extensions `GetEnumerator()` pour les boucles `foreach`
- Les expressions lambda peuvent déclarer des paramètres d'abandon
- Les attributs peuvent être appliqués aux fonctions locales

La version C# 9 poursuit le travail de maintien d'un langage de programmation moderne et universel. Les fonctionnalités continuent de prendre en charge les charges de travail et les types d'applications modernes.

C# version 8.0

Date de publication : septembre 2019

C# 8.0 est la première version principale de C# qui cible spécifiquement .NET Core. Certaines fonctionnalités s'appuient sur de nouvelles fonctionnalités CLR, d'autres sur les types de bibliothèque ajoutés uniquement dans .NET Core. C# 8.0 ajoute les fonctionnalités et améliorations suivantes au langage C# :

- Membres ReadOnly
- Méthodes d'interface par défaut
- Amélioration des critères spéciaux :
  - Expressions switch
  - Modèles de propriétés
  - Modèles de tuples
  - Modèles positionnels
- Déclarations using
- Fonctions locales statiques
- Structs ref jetables
- Types références Nullables
- Flux asynchrones
- Index et plages
- Attribution de fusion Null

- Types construits non managés
- Stackalloc dans des expressions imbriquées
- Amélioration des chaînes verbatim interpolées

Les membres de l'interface par défaut nécessitent des améliorations dans le CLR. Ces fonctionnalités ont été ajoutées dans le CLR pour .NET Core 3.0. Les plages et les index, ainsi que les flux asynchrones nécessitent de nouveaux types dans les bibliothèques .NET Core 3.0. Les types de référence pouvant accepter la valeur Nul, tout en étant implémentés dans le compilateur, sont beaucoup plus utiles lorsque les bibliothèques sont annotées pour fournir des informations sémantiques sur l'état null des arguments et des valeurs de retour. Ces annotations sont ajoutées dans les bibliothèques .NET Core.

C# version 7.3

Publication : mai 2018

Il existe deux thèmes principaux pour la version C# 7.3. Un thème fournit des fonctionnalités permettant au code sécurisé d'être aussi performant que le code non sécurisé. Le second thème fournit des améliorations incrémentielles aux fonctionnalités existantes. De nouvelles options de compilateur ont également été ajoutées dans cette version.

Les nouvelles fonctionnalités suivantes prennent en charge le thème de meilleures performances pour le code sécurisé :

- Vous pouvez accéder à des champs fixes sans épinglage.
- Vous pouvez réaffecter des variables locales `ref`.
- Vous pouvez utiliser des initialiseurs sur les tableaux `stackalloc`.
- Vous pouvez utiliser des instructions `fixed` avec tout type prenant en charge un modèle.
- Vous pouvez utiliser des contraintes plus génériques.

Les améliorations suivantes ont été apportées aux fonctionnalités existantes :

- Vous pouvez tester `==` et `!=` avec des types de tuple.
- Vous pouvez utiliser des variables d'expression dans d'autres emplacements.
- Vous pouvez joindre des attributs à un champ de stockage de propriétés implémentées automatiquement.
- La résolution de la méthode lorsque des arguments diffèrent de `in` a été améliorée.
- La résolution de la surcharge comporte maintenant moins de cas ambigus.

Les nouvelles options du compilateur sont les suivantes :

- -publicsign pour activer la signature d'assemblys Open Source Software (OSS).
- -pathmap pour fournir un mappage des répertoires sources.

C# version 7.2

Publication : novembre 2017

C# 7.2 a ajouté plusieurs fonctionnalités de langage de petite taille :

- Initialiseurs sur des tableaux `stackalloc`.
- Utilisez des instructions `fixed` avec n'importe quel type qui prend en charge un modèle.
- Accéder aux champs fixes sans épinglage.
- Réaffecter des variables `ref` locales.
- Déclarez des types `readonly struct` pour indiquer qu'un struct est immuable et doit être transmis en tant que paramètre `in` à ses méthodes membres.
- Ajoutez le modificateur `in` aux paramètres pour spécifier qu'un argument est passé par référence, mais qu'il n'est pas modifié par la méthode appelée.
- Utilisez le modificateur `ref readonly` sur les retours de méthode, pour indiquer qu'une méthode retourne sa valeur par référence, mais n'autorise pas les écritures dans cet objet.
- Déclarez des types `ref struct`, pour indiquer qu'un type de struct accède directement à la mémoire managée et doit toujours être alloué à la pile.
- Utilisez des contraintes génériques supplémentaires.
- Arguments nommés non placés en position de fin
  - Les arguments nommés peuvent être suivis par des arguments de position.
- Traits de soulignement de début dans les littéraux numériques
  - Les littéraux numériques peuvent maintenant comporter des traits de soulignement de début avant tout chiffre affiché.
- Modificateur d'accès private protected
  - Le modificateur d'accès `private protected` active l'accès pour les classes dérivées dans le même assembly.
- Expressions `ref` conditionnelles
  - Le résultat d'une expression conditionnelle (`?:`) peut maintenant être une référence.

C# version 7.1

Publication : août 2017

C# a commencé à sortir *versions de point* avec C# 7.1. Cette version a ajouté l'élément de configuration de sélection de version de langue, trois nouvelles fonctionnalités de langage et un nouveau comportement du compilateur.

Les nouvelles fonctionnalités de langage de cette version sont :

- Méthode asyncMain
  - Le point d'entrée pour une application peut avoir le modificateur `async`.
- Expressions littérales default
  - Vous pouvez utiliser des expressions littérales default dans les expressions de valeur par défaut quand le type cible peut être inféré.
- Noms des éléments de tuple inférés
  - Les noms des éléments de tuple peuvent être inférés dans de nombreux cas à partir de l'initialisation du tuple.
- Critères spéciaux sur les paramètres de type générique
  - Il est possible d'utiliser des expressions de critères spéciaux sur les variables dont le type est un paramètre de type générique.

Enfin, le compilateur a deux options, -refout et -refonly, qui contrôlent la génération d'assemblys de références

C# version 7.0

Publication : mars 2017

C# version 7.0 a été publié avec Visual Studio 2017. Cette version présente des points évolutifs et très attrayants dans la veine de C# 6.0. Voici quelques-unes des nouvelles fonctionnalités :

- Variables out
- Tuples et déconstruction
- Critères spéciaux
- Fonctions locales
- Membres expression-bodied étendus
- Variables locales ref
- Retours par référence

Autres fonctionnalités disponibles :

- Éléments ignorés
- Littéraux binaires et séparateurs numériques

- Expressions throw

Toutes ces fonctionnalités offrent de nouvelles capacités aux développeurs ainsi que la possibilité d'écrire du code plus propre. Il s'agit notamment de condenser la déclaration des variables à utiliser avec le mot clé `out` et en autorisant plusieurs valeurs de retour par le biais d'un tuple. .NET Core cible à présent n'importe quel système d'exploitation et garde les yeux rivés sur le cloud et la portabilité. Ces nouvelles fonctions occupent sans aucun doute les pensées des concepteurs du langage, en plus des fonctionnalités à venir.

C# version 6.0

Publication : juillet 2015

La version 6.0, sortie avec Visual Studio 2015, proposait de nombreuses fonctionnalités plus petites qui rendent la programmation C# plus productive. En voici quelques-unes :

- Importations statiques
- Filtres d'exceptions
- Initialiseurs de propriétés automatiques
- Membres expression-bodied
- Propagateur Null
- Interpolation de chaîne
- Opérateur nameof

Quelques autres nouvelles fonctions :

- Initialiseurs d'index
- Await dans des blocs catch/finally
- Valeurs par défaut pour les propriétés d'accesseur Get

Chacune de ces fonctionnalités est intéressante individuellement. Mais si vous les examinez dans leur ensemble, un modèle intéressant se dégage. Dans cette version, C# a commencé à éliminer le texte réutilisable du langage pour rendre le code plus laconique et plus lisible. Ainsi, pour les amateurs de code propre et simple, cette version du langage était une énorme victoire.

Une autre nouveauté a été proposée avec cette version, même s'il ne s'agit pas d'une fonctionnalité de langage traditionnelle à proprement dit. Le compilateur Roslyn a été publié en tant que service. Le compilateur C# est à présent écrit en C# et vous pouvez l'utiliser dans le cadre de vos efforts de programmation.

C# version 5.0

Publication : août 2012

C# version 5.0, sortie avec Visual Studio 2012, était une version ciblée du langage. Presque tous les efforts déployés pour cette version portaient sur un autre concept révolutionnaire du langage : le modèle `async` et `await` pour la programmation asynchrone. Voici la liste des principales fonctionnalités :

- Membres asynchrones
- Attributs d'informations de l'appelant
- Projet de code : Attributs des informations de l'appelant en C# 5.0

L'attribut d'informations de l'appelant vous permet de récupérer facilement des informations sur le contexte d'exécution sans avoir recours à une multitude de code de réflexion réutilisable. Nombre de ses usages ont trait aux diagnostics et aux tâches de journalisation.

Mais `async` et `await` sont les véritables vedettes de cette version. Quand ces fonctionnalités sont apparues en 2012, C# a encore modifié la donne en intégrant l'asynchronie en tant que participant de premier ordre dans le langage. Si vous avez déjà eu affaire à de longues opérations et à l'implémentation de rappels, vous avez probablement adoré cette fonctionnalité de langage.

C# version 4.0

Publication : avril 2010

Il fut difficile pour C# version 4.0, sortie avec Visual Studio 2010, d'être à la hauteur du concept révolutionnaire incarné par la version 3.0. Cette version introduisit de nouvelles fonctionnalités intéressantes :

- Liaison dynamique
- Arguments nommés/facultatifs
- Covariance et contravariance génériques
- Types interop incorporés

Les types d'interopérabilité incorporés ont facilité le déploiement de la création d'assemblys COM Interop pour votre application. La covariance et la contravariance génériques vous donnent plus de contrôle sur l'utilisation des génériques, mais elles sont un peu académiques et probablement plus appréciées des auteurs de frameworks et de bibliothèques. Les paramètres nommés et facultatifs vous permettent d'éliminer

les nombreuses surcharges de méthode et s'avèrent plus pratiques. Mais aucune de ces fonctionnalités ne représente un vrai changement de paradigme.

La fonctionnalité majeure était plutôt l'introduction du mot clé `dynamic`. Le mot clé `dynamic` introduisit dans C# version 4.0 la possibilité de remplacer le compilateur lors de la saisie au moment de la compilation. En utilisant le mot clé dynamic, vous pouvez créer des constructions semblables aux langages dynamiquement typés comme JavaScript. Vous pouvez créer un `dynamic x = "a string"`, puis y ajouter six, et ainsi laisser le runtime se débrouiller avec ce qui doit se produire par la suite.

La liaison dynamique offre la possibilité de commettre des erreurs, mais également de contrôler davantage le langage.

C# version 3.0

Publication : novembre 2007

C# version 3.0 est apparu fin 2007, en même temps que Visual Studio 2008, même si l'éventail complet des fonctionnalités du langage n'a réellement voir le jour qu'avec le .NET Framework version 3.5. Cette version a insufflé un changement majeur dans l'évolution de C#. Elle a imposé C# en tant que langage de programmation vraiment formidable. Examinons certaines fonctionnalités importantes dans cette version :

- Propriétés implémentées automatiquement
- Types anonymes
- Expressions de requête
- Expressions lambda
- Arborescences de l'expression
- Méthodes d'extension
- Variables locales implicitement typées
- Méthodes partielles
- Initialiseurs d'objets et de collections

Rétrospectivement, nombre de ces fonctionnalités semblent à la fois inéluctables et inséparables. Elles s'assemblent de façon stratégique. L'expression de requête, également appelée LINQ (Language-Integrated Query), était considérée comme la fonctionnalité remarquable de cette version de C#.

Un point de vue plus nuancé s'intéresse aux arborescences d'expressions, aux expressions lambda et aux types anonymes sur lesquels se construit LINQ. Mais, dans les deux cas, C# 3.0 présentait un concept révolutionnaire. C# 3.0 jetait les bases qui allaient permettre à C# de devenir un langage fonctionnel/orienté objet hybride.

Plus précisément, il devint alors possible d'écrire des requêtes déclaratives de style SQL pour effectuer des opérations sur des collections, entre autres choses. Au lieu d'écrire une boucle `for` pour calculer la moyenne d'une liste d'entiers, vous pouviez alors simplement utiliser `list.Average()`. La combinaison des expressions de requête et des méthodes d'extension a permis à de telles listes d'entiers de devenir beaucoup plus intelligentes.

C# version 2.0

Publication : novembre 2005

Examinons certaines fonctionnalités majeures de C# 2.0, sorti en 2005, en même temps que Visual Studio 2005 :

- Génériques
- Types partiels
- Méthodes anonymes
- Types valeur Nullable
- Itérateurs
- Covariance et contravariance

D'autres fonctionnalités de C# 2.0 ajoutaient des capacités aux fonctionnalités existantes :

- Accessibilité distincte des accesseurs Get/Set
- Conversions de groupes de méthodes (délégués)
- Classes statiques
- Inférence de délégué

Alors que C# avait démarré en tant que langage orienté objet générique, C# version 2.0 a rapidement changé tout ça. Avec les génériques, les types et les méthodes peuvent fonctionner sur un type arbitraire tout en assurant quand même la cohérence des types. À titre d'exemple, un List<T> vous permet d'avoir `List<string>` ou `List<int>` et d'effectuer des opérations qui maintiennent la cohérence des types sur des chaînes ou entiers alors que vous itérez en leur sein. Il vaut mieux utiliser des génériques que créer un type `ListInt` qui dérive de `ArrayList` ou effectuer un cast de `Object` pour chaque opération.

C# version 2.0 a introduit les itérateurs. En bref, les itérateurs permettent d'examiner tous les éléments dans un `List` (ou d'autres types énumérables) avec une boucle `foreach`. Le fait de disposer d'itérateurs comme composants de premier ordre du

langage en a considérablement amélioré la lisibilité et la capacité de raisonnement des utilisateurs vis-à-vis du code.

Pourtant, C# était toujours en train de courir derrière Java. Java avait déjà publié des versions qui incluaient des génériques et des itérateurs. Mais cela allait bientôt changer quand les deux langages poursuivraient des chemins différents.

C# version 1.2

Publication : avril 2003

C# version 1.2 était fourni avec Visual Studio .NET 2003. Cette version contenait quelques améliorations mineures du langage. La principale est que, à compter de cette version, le code était généré dans une boucle `foreach` (appelée Dispose) sur un IEnumerator quand ce IEnumerator implémentait IDisposable.

C# version 1.0

Publication : janvier 2002

Rétrospectivement, le langage C# version 1.0, sortie avec Visual Studio .NET 2002, ressemblait beaucoup à Java. Dans ses objectifs de conception énoncés pour ECMA, il cherchait à être un « langage orienté objet simple, moderne et généraliste ». À l'époque, ressembler à Java signifiait qu'il avait atteint ces objectifs de conception.

Mais si vous repensez à C# 1.0 maintenant, cela peut vous donner le vertige. Il lui manquait des fonctionnalités asynchrones intégrées et certaines des fonctionnalités liées aux génériques qui sont aujourd'hui pour vous une évidence. En fait, il ne proposait pas du tout de génériques. Et LINQ ? Pas encore disponible. Il fallait attendre encore plusieurs années.

C# version 1.0 semblait dénué de fonctionnalités, par rapport à aujourd'hui. Vous vous retrouviez à écrire plus ou moins du code détaillé. Mais il faut bien commencer quelque part. C# version 1.0 était une alternative viable à Java sur la plateforme Windows.

Les principales fonctionnalités du langage C# 1.0 étaient les suivantes :

- Classes
- Structures
- Interfaces
- Événements
- Propriétés

- Délégués
- Opérateurs et expressions
- Instructions
- Attributs

Article[initialement publié sur le blog NDepend], avec l'aimable autorisation d'Erik Dietrich et de Patrick Smacchia.

Relations entre les fonctionnalités de langage et les types de bibliothèque

Article • 10/05/2023

La définition du langage C# nécessite une bibliothèque standard avec certains types et certains membres accessibles sur ces types. Le compilateur génère du code qui a besoin d'utiliser ces types et membres pour de nombreuses fonctionnalités de langage différentes. Quand cela est nécessaire, il existe des packages NuGet qui contiennent tous les types à utiliser pour des versions plus récentes du langage si vous écrivez du code destiné à des environnements où ces types ou membres n'ont pas encore été déployés.

Cette dépendance sur les fonctionnalités de la bibliothèque standard fait partie du langage C# depuis le début. Dans la première version, il y avait les types suivants :

- Exception : utilisé pour toutes les exceptions générées par le compilateur.
- String : le type `string` C# est un synonyme de String.
- Int32 : synonyme de `int`.

Cette première version était simple : le compilateur et la bibliothèque standard étaient fournis ensemble, et il n'y avait qu'une seule version de chacun.

Dans les versions ultérieures de C#, de nouveaux types ou membres ont parfois été ajoutés aux dépendances. Par exemple, INotifyCompletion, CallerFilePathAttribute et CallerMemberNameAttribute. C# 7.0 poursuit dans ce sens en ajoutant une dépendance sur ValueTuple pour implémenter la fonctionnalité de langage tuples.

L'équipe de conception du langage s'efforce de réduire la surface d'exposition des types et des membres requis dans une bibliothèque standard conforme. Cet objectif s'ajoute à la nécessité de conserver une conception épurée à mesure que de nouvelles fonctionnalités de bibliothèque sont incorporées au langage. De nouvelles fonctionnalités seront ajoutées dans les futures versions du langage C# qui nécessiteront de nouveaux types et membres dans une bibliothèque standard. Il est donc important de bien comprendre comment gérer ces dépendances dans votre travail.

Gestion des dépendances

Les outils du compilateur C# sont maintenant dissociés du cycle de mise en production des bibliothèques .NET sur les plateformes prises en charge. En effet, les bibliothèques

.NET n’ont pas toutes les mêmes cycles de mise en production : le .NET Framework sur Windows est fourni en tant que mise à jour Windows Update, .NET Core est fourni selon une planification distincte, et les versions Xamarin des mises à jour de bibliothèque sont fournies avec les outils Xamarin pour chaque plateforme cible.

La plupart du temps, ces changements passeront inaperçus pour les utilisateurs. Toutefois, si vous utilisez une version du langage plus récente qui fait appel à des fonctionnalités actuellement non disponibles dans les bibliothèques .NET sur cette plateforme, vous devez référencer les packages NuGet pour fournir ces nouveaux types nécessaires. Une fois que les plateformes prises en charge par votre application auront été mises à jour avec les nouvelles versions de framework, vous pourrez supprimer la référence ajoutée.

Grâce à cette dissociation, vous pouvez utiliser les nouvelles fonctionnalités de langage même quand vous ciblez des machines qui n’ont pas le framework approprié.

Considérations relatives à la version et la mise à jour pour les développeurs C#

Article • 30/06/2023

La compatibilité est un objectif important quand de nouvelles fonctionnalités sont ajoutées au langage C#. Dans presque tous les cas, le code existant peut être recompilé avec une nouvelle version du compilateur sans problème. L'équipe du runtime .NET a également pour objectif de garantir la compatibilité des bibliothèques mises à jour. Dans presque tous les cas, lorsque votre application est lancée à partir d'un runtime mis à jour avec des bibliothèques mises à jour, le comportement reste exactement le même qu'avec les versions précédentes.

La version du langage utilisée pour compiler l'application correspond généralement au moniker de framework cible (TFM) du runtime référencé dans votre projet. Pour plus d'informations sur le changement de la version du langage par défaut, consultez l'article intitulé Configurer la version du langage. Ce comportement par défaut garantit une compatibilité maximale.

Quand des *changements cassants* sont introduits, ils sont classés de la façon suivante :

- *Changement cassant binaire* : un changement cassant binaire entraîne un comportement différent, y compris potentiellement un plantage, dans votre application ou bibliothèque qui est lancée à partir d'un nouveau runtime. Vous devez recompiler votre application pour incorporer ces changements. Le binaire existant ne fonctionnera pas correctement.
- *Changement cassant source* : un changement cassant source modifie la signification de votre code source. Vous devez modifier le code source avant de compiler votre application avec la dernière version du langage. Votre binaire existant s'exécutera correctement avec la version plus récente de l'hôte et du runtime. Notez que pour la syntaxe du langage, un *changement cassant source* est également un *changement de comportement*, comme défini dans les changements cassants de runtime.

Quand un changement cassant binaire impacte votre application, vous devez recompiler l'application, mais vous n'avez pas besoin de modifier le code source. Quand un changement cassant source impacte votre application, le binaire existant continue de s'exécuter correctement dans les environnements avec le runtime et les bibliothèques mis à jour. Toutefois, vous devez apporter les modifications sources à recompiler avec la nouvelle version du langage et du runtime. Si une modification est à la fois un

changement cassant source et un changement cassant binaire, vous devez recompiler votre application avec la dernière version et effectuer les mises à jour sources.

En raison de l'objectif recherché d'éviter les changements cassants par l'équipe du langage C# et l'équipe du runtime, la mise à jour de votre application consiste généralement à mettre à jour le TFM et à recompiler l'application. Toutefois, pour les bibliothèques qui sont distribuées publiquement, vous devez soigneusement évaluer votre stratégie de prise en charge des versions des TFM et des langages. Vous créez peut-être une nouvelle bibliothèque avec des fonctionnalités incluses dans la version la plus récente et avez besoin de vous assurer que les applications créées à l'aide de versions précédentes du compilateur peuvent l'utiliser. Ou vous mettez peut-être à niveau une bibliothèque existante et la plupart de vos utilisateurs ne disposent pas encore des versions mises à niveau.

Introduction de changements cassants dans vos bibliothèques

Lorsque vous adoptez de nouvelles fonctionnalités de langage dans l'API publique de votre bibliothèque, vous devez évaluer si l'adoption d'une fonctionnalité va introduire un changement cassant binaire ou source pour les utilisateurs de votre bibliothèque. Toutes les modifications apportées à votre implémentation interne qui n'apparaissent pas dans les interfaces `public` ou `protected` sont compatibles.

> ⓘ **Notes**
>
> Si vous utilisez **System.Runtime.CompilerServices.InternalsVisibleToAttribute** pour permettre aux types de voir les membres internes, les membres internes risquent d'introduire des changements cassants.

En présence d'un *changement cassant binaire*, les utilisateurs doivent recompilent leur code afin d'utiliser la nouvelle version. Par exemple, examinez cette méthode publique :

C#

```
public double CalculateSquare(double value) => value * value;
```

Si vous ajoutez le modificateur `in` à la méthode, il s'agit d'un changement cassant binaire :

C#

```
public double CalculateSquare(in double value) => value * value;
```

Les utilisateurs doivent alors recompiler toute application utilisant la méthode `CalculateSquare` pour que la nouvelle bibliothèque fonctionne correctement.

Un *changement cassant source* oblige les utilisateurs à modifier leur code avant de le recompiler. Par exemple, examinez ce type :

C#

```
public class Person
{
    public string FirstName { get; }
    public string LastName { get; }

    public Person(string firstName, string lastName) => (FirstName,
LastName) = (firstName, lastName);

    // other details omitted
}
```

Dans une version plus récente, vous souhaitez tirer parti des membres synthétisés générés pour les types `record`. Vous apportez la modification suivante :

C#

```
public record class Person(string FirstName, string LastName);
```

La modification précédente nécessite de modifier aussi tous les types dérivés de `Person`. Toutes ces déclarations doivent ajouter le modificateur `record` à leurs déclarations.

Impact des changements cassants

Lorsque vous introduisez un *changement cassant binaire* dans votre bibliothèque, vous forcez la recompilation de tous les projets qui utilisent cette bibliothèque. Toutefois, aucun code source dans ces projets n'a besoin d'être modifié. L'impact du changement cassant est donc relativement faible pour chaque projet.

Quand vous introduisez un *changement cassant source* dans votre bibliothèque, vous forcez tous les projets à apporter les modifications sources afin d'utiliser votre nouvelle bibliothèque. Si la modification à faire nécessite de nouvelles fonctionnalités de langage, vous forcez ces projets à effectuer une mise à niveau vers la même version de langage

et le même TFM que vous utilisez actuellement. Vous avez accru la charge de travail de vos utilisateurs, et vous les avez éventuellement forcés à effectuer une mise à niveau.

L'impact réel de l'introduction d'un changement cassant dépend du nombre de projets qui dépendent de votre bibliothèque. Si votre bibliothèque est utilisée en interne par quelques applications seulement, vous pouvez traiter les changements cassants dans tous les projets impactés. En revanche, si votre bibliothèque est téléchargée publiquement, vous devez évaluer l'impact potentiel et envisager des alternatives :

- Vous pouvez ajouter de nouvelles API en parallèle aux API existantes.
- Vous pouvez envisager des builds parallèles pour différents TPM.
- Vous pouvez envisager le multiciblage.

Tutoriel : Explorer les fonctionnalités de C# 11 - membres virtuels statiques dans les interfaces

Article • 05/06/2023

C# 11 et .NET 7 incluent des *membres virtuels statiques dans les interfaces*. Cette fonctionnalité vous permet de définir des interfaces qui incluent des opérateurs surchargés ou d'autres membres statiques. Une fois que vous avez défini des interfaces avec des membres statiques, vous pouvez utiliser ces interfaces comme contraintes pour créer des types génériques qui utilisent des opérateurs ou d'autres méthodes statiques. Même si vous ne créez pas d'interfaces avec des opérateurs surchargés, vous tirerez probablement parti de cette fonctionnalité et des classes mathématiques génériques activées par la mise à jour du langage.

Ce didacticiel vous montre comment effectuer les opérations suivantes :

- ✓ Définissez des interfaces avec des membres statiques.
- ✓ Utilisez des interfaces pour définir des classes qui implémentent des interfaces avec des opérateurs définis.
- ✓ Créez des algorithmes génériques qui s'appuient sur des méthodes d'interface statiques.

Prérequis

Vous devez configurer votre ordinateur pour exécuter .NET 7, qui prend en charge C# 11. Le compilateur C# 11 est disponible à partir de Visual Studio 2022 version 17.3 ou du SDK .NET 7.

Méthodes d'interface abstraite statiques

Commençons avec un exemple. La méthode suivante retourne le point médian de deux nombres `double` :

C#

```
public static double MidPoint(double left, double right) =>
    (left + right) / (2.0);
```

La même logique fonctionne pour n'importe quel type numérique : `int`, `short`, `long`, `float` `decimal` ou tout type qui représente un nombre. Vous devez disposer d'un moyen d'utiliser les opérateurs `+` et `/` et de définir une valeur pour `2`. Vous pouvez utiliser l'interface System.Numerics.INumber<TSelf> pour écrire la méthode précédente en tant que méthode générique suivante :

```
C#
```

```
public static T MidPoint<T>(T left, T right)
    where T : INumber<T> => (left + right) / T.CreateChecked(2);  // note:
the addition of left and right may overflow here; it's just for
demonstration purposes
```

Tout type qui implémente l'interface INumber<TSelf> doit inclure une définition pour `operator +` et pour `operator /`. Le dénominateur est défini par `T.CreateChecked(2)` pour créer la valeur `2` pour n'importe quel type numérique, ce qui force le dénominateur à être du même type que les deux paramètres. INumberBase<TSelf>.CreateChecked<TOther>(TOther) crée une instance du type à partir de la valeur spécifiée et lève une OverflowException si la valeur se situe en dehors de la plage représentable. (Cette implémentation a le potentiel de dépassement si `left` et `right` sont toutes les deux des valeurs suffisamment grandes. Il existe d'autres algorithmes qui peuvent éviter ce problème potentiel.)

Vous définissez des membres abstraits statiques dans une interface à l'aide de la syntaxe familière : vous ajoutez les modificateurs `static` et `abstract` à tout membre statique qui ne fournit pas d'implémentation. L'exemple suivant définit une interface `IGetNext<T>` qui peut être appliquée à n'importe quel type qui remplace `operator ++` :

```
C#
```

```
public interface IGetNext<T> where T : IGetNext<T>
{
    static abstract T operator ++(T other);
}
```

La contrainte que l'argument de type, `T`, implémente `IGetNext<T>` garantit que la signature de l'opérateur inclut le type contenant ou son argument de type. De nombreux opérateurs imposent que ses paramètres correspondent au type ou soient le paramètre de type contraint d'implémenter le type contenant. Sans cette contrainte, l'opérateur `++` n'a pas pu être défini dans l'interface `IGetNext<T>`.

Vous pouvez créer une structure qui crée une chaîne de caractères « A » où chaque incrément ajoute un autre caractère à la chaîne à l'aide du code suivant :

C#

```
public struct RepeatSequence : IGetNext<RepeatSequence>
{
    private const char Ch = 'A';
    public string Text = new string(Ch, 1);

    public RepeatSequence() {}

    public static RepeatSequence operator ++(RepeatSequence other)
        => other with { Text = other.Text + Ch };

    public override string ToString() => Text;
}
```

Plus généralement, vous pouvez créer n'importe quel algorithme dans lequel vous pouvez définir que `++` signifie « produire la valeur suivante de ce type ». L'utilisation de cette interface produit du code et des résultats clairs :

C#

```
var str = new RepeatSequence();

for (int i = 0; i < 10; i++)
    Console.WriteLine(str++);
```

L'exemple précédent génère la sortie suivante :

PowerShell

```
A
AA
AAA
AAAA
AAAAA
AAAAAA
AAAAAAA
AAAAAAAA
AAAAAAAAA
AAAAAAAAAA
```

Ce petit exemple illustre la motivation de cette fonctionnalité. Vous pouvez utiliser la syntaxe naturelle pour les opérateurs, les valeurs constantes et d'autres opérations statiques. Vous pouvez explorer ces techniques lorsque vous créez plusieurs types qui s'appuient sur des membres statiques, y compris des opérateurs surchargés. Définissez les interfaces qui correspondent aux capacités de vos types, puis déclarez la prise en charge de ces types pour la nouvelle interface.

Mathématiques génériques

Le scénario le plus intéressant pour autoriser les méthodes statiques, y compris les opérateurs, dans les interfaces est de prendre en charge les algorithmes mathématiques génériques. La bibliothèque de classes de base .NET 7 contient des définitions d'interface pour de nombreux opérateurs arithmétiques et des interfaces dérivées qui combinent de nombreux opérateurs arithmétiques dans une interface `INumber<T>`. Appliquons ces types pour créer un enregistrement `Point<T>` qui peut utiliser n'importe quel type numérique pour `T`. Le point peut être déplacé par certains `XOffset` et `YOffset` à l'aide de l'opérateur `+`.

Commencez par créer une application Console, à l'aide de `dotnet new` ou de Visual Studio. Définissez la version du langage C# sur « préversion », ce qui active les fonctionnalités d'évaluation de C# 11. Ajoutez l'élément suivant à votre fichier *csproj* à l'intérieur d'un élément `<PropertyGroup>` :

XML

```xml
<LangVersion>preview</LangVersion>
```

> ⓘ Notes
>
> Cet élément ne peut pas être défini à l'aide de l'interface utilisateur de Visual Studio. Vous devez modifier le fichier projet directement.

L'interface publique pour `Translation<T>` et `Point<T>` doit ressembler au code suivant :

C#

```csharp
// Note: Not complete. This won't compile yet.
public record Translation<T>(T XOffset, T YOffset);

public record Point<T>(T X, T Y)
{
    public static Point<T> operator +(Point<T> left, Translation<T> right);
}
```

Vous utilisez le type `record` pour les types `Translation<T>` et `Point<T>` : les deux valeurs stockent deux valeurs, et elles représentent le stockage des données plutôt que le comportement sophistiqué. L'implémentation de `operator +` ressemblerait au code suivant :

C#

```
public static Point<T> operator +(Point<T> left, Translation<T> right) =>
    left with { X = left.X + right.XOffset, Y = left.Y + right.YOffset };
```

Pour que le code précédent soit compilé, vous devez déclarer que `T` prend en charge l'interface `IAdditionOperators<TSelf, TOther, TResult>`. Cette interface inclut la méthode statique `operator +`. Elle déclare trois paramètres de type : un pour l'opérande gauche, un pour l'opérande droit et un pour le résultat. Certains types implémentent `+` pour différents types d'opérandes et de résultats. Ajoutez une déclaration que l'argument de type, `T` implémente `IAdditionOperators<T, T, T>` :

C#

```
public record Point<T>(T X, T Y) where T : IAdditionOperators<T, T, T>
```

Après avoir ajouté cette contrainte, votre classe `Point<T>` peut utiliser le `+` pour son opérateur d'ajout. Ajoutez la même contrainte à la déclaration `Translation<T>` :

C#

```
public record Translation<T>(T XOffset, T YOffset) where T :
IAdditionOperators<T, T, T>;
```

La contrainte `IAdditionOperators<T, T, T>` empêche un développeur qui utilise votre classe de créer un `Translation` à l'aide d'un type qui ne respecte pas la contrainte pour l'ajout à un point. Vous avez ajouté les contraintes nécessaires au paramètre de type pour `Translation<T>` et `Point<T>` pour que ce code fonctionne. Vous pouvez effectuer des tests en ajoutant du code comme ci-dessus en suivant les déclarations de `Translation` et `Point` dans votre fichier *Program.cs* :

C#

```
var pt = new Point<int>(3, 4);

var translate = new Translation<int>(5, 10);

var final = pt + translate;

Console.WriteLine(pt);
Console.WriteLine(translate);
Console.WriteLine(final);
```

Vous pouvez rendre ce code plus réutilisable en déclarant que ces types implémentent les interfaces arithmétiques appropriées. La première modification à apporter consiste à déclarer qui `Point<T, T>` implémente l'interface `IAdditionOperators<Point<T>, Translation<T>, Point<T>>`. Le type `Point` utilise différents types pour les opérandes et le résultat. Le type `Point` implémente déjà un `operator +` avec cette signature. L'ajout de l'interface à la déclaration est donc tout ce dont vous avez besoin :

C#

```
public record Point<T>(T X, T Y) : IAdditionOperators<Point<T>,
Translation<T>, Point<T>>
    where T : IAdditionOperators<T, T, T>
```

Enfin, lorsque vous effectuez un ajout, il est utile d'avoir une propriété qui définit la valeur d'identité additive pour ce type. Il existe une nouvelle interface pour cette fonctionnalité : IAdditiveIdentity<TSelf,TResult>. Une traduction de `{0, 0}` est l'identité additive : le point résultant est le même que l'opérande gauche. L'interface `IAdditiveIdentity<TSelf, TResult>` définit une propriété en lecture seule, `AdditiveIdentity`, qui retourne la valeur d'identité. La `Translation<T>` a besoin de quelques modifications pour implémenter cette interface :

C#

```
using System.Numerics;

public record Translation<T>(T XOffset, T YOffset) :
IAdditiveIdentity<Translation<T>, Translation<T>>
    where T : IAdditionOperators<T, T, T>, IAdditiveIdentity<T, T>
{
    public static Translation<T> AdditiveIdentity =>
        new Translation<T>(XOffset: T.AdditiveIdentity, YOffset:
T.AdditiveIdentity);
}
```

Il y a quelques changements ici. Nous allons donc les parcourir un par un. Tout d'abord, vous déclarez que le type `Translation` implémente l'interface `IAdditiveIdentity` :

C#

```
public record Translation<T>(T XOffset, T YOffset) :
IAdditiveIdentity<Translation<T>, Translation<T>>
```

Vous pouvez ensuite essayer d'implémenter le membre d'interface comme indiqué dans le code suivant :

C#

```
public static Translation<T> AdditiveIdentity =>
    new Translation<T>(XOffset: 0, YOffset: 0);
```

Le code précédent ne sera pas compilé, car `0` dépend du type. Réponse : Utilisez `IAdditiveIdentity<T>.AdditiveIdentity` pour `0`. Ce changement signifie que vos contraintes doivent maintenant inclure que `T` implémente `IAdditiveIdentity<T>`. Cela entraîne l'implémentation suivante :

C#

```
public static Translation<T> AdditiveIdentity =>
    new Translation<T>(XOffset: T.AdditiveIdentity, YOffset:
T.AdditiveIdentity);
```

Maintenant que vous avez ajouté cette contrainte sur `Translation<T>`, vous devez ajouter la même contrainte à `Point<T>` :

C#

```
using System.Numerics;

public record Point<T>(T X, T Y) : IAdditionOperators<Point<T>,
Translation<T>, Point<T>>
    where T : IAdditionOperators<T, T, T>, IAdditiveIdentity<T, T>
{
    public static Point<T> operator +(Point<T> left, Translation<T> right)
=>
        left with { X = left.X + right.XOffset, Y = left.Y + right.YOffset
};
}
```

Cet exemple vous a donné un aperçu de la composition des interfaces pour les mathématiques génériques. Vous avez appris à :

- ✓ Écrivez une méthode qui s'appuie sur l'interface `INumber<T>` afin qu'elle puisse être utilisée avec n'importe quel type numérique.
- ✓ Créez un type qui s'appuie sur les interfaces d'ajout pour implémenter un type qui ne prend en charge qu'une seule opération mathématique. Ce type déclare sa prise en charge pour ces mêmes interfaces afin qu'il puisse être composé d'autres manières. Les algorithmes sont écrits à l'aide de la syntaxe la plus naturelle des opérateurs mathématiques.

Testez ces fonctionnalités et inscrivez des commentaires. Vous pouvez utiliser l'élément de menu *Envoyer des commentaires* dans Visual Studio ou créer un problème dans le référentiel roslyn sur GitHub. Générez des algorithmes génériques qui fonctionnent avec n'importe quel type numérique. Générez des algorithmes à l'aide de ces interfaces où l'argument de type ne peut implémenter qu'un sous-ensemble de capacités de type nombre. Même si vous ne créez pas de nouvelles interfaces qui utilisent ces capacités, vous pouvez expérimenter leur utilisation dans vos algorithmes.

Voir aussi

- Mathématiques génériques

Créer des types d'enregistrements

Article • 09/05/2023

C# 9 introduit des *enregistrements*, un nouveau type de référence que vous pouvez créer au lieu de classes ou de structs. C# 10 ajoute des *structs d'enregistrement* pour que vous puissiez définir des enregistrements en tant que types valeur. Les enregistrements distincts des classes dans ce type d'enregistrement utilisent *l'égalité basée sur la valeur*. Deux variables d'un type d'enregistrement sont égales si les définitions de type d'enregistrement sont identiques et, si pour chaque champ, les valeurs des deux enregistrements sont égales. Deux variables d'un type de classe sont égales si les objets référencés sont le même type de classe et que les variables font référence au même objet. L'égalité basée sur la valeur implique d'autres fonctionnalités que vous souhaiterez probablement dans les types d'enregistrements. Le compilateur génère un grand nombre de ces membres lorsque vous déclarez un `record` au lieu d'une `class`. Le compilateur génère ces mêmes méthodes pour les types `record struct`.

Dans ce tutoriel, vous apprendrez à :

- ✓ Déterminez si vous souhaitez ajouter le modificateur `record` à un type `class`.
- ✓ Déclarez les types d'enregistrements et les types d'enregistrements positionnels.
- ✓ Remplacez vos méthodes par les méthodes générées par le compilateur dans les enregistrements.

Prérequis

Vous devez configurer votre ordinateur pour exécuter .NET 6 ou version ultérieure, avec le compilateur C# 10 ou version ultérieure. Le compilateur C# 10 est disponible à partir de Visual Studio 2022 ou du Kit de développement logiciel (SDK).NET 6.

Caractéristiques des enregistrements

On définit un *enregistrement* en déclarant un type avec le mot clé `record`, en modifiant une déclaration de `class` ou de `struct`. Si vous le souhaitez, vous pouvez omettre le mot clé `class` pour créer une `record class`. Un enregistrement suit la sémantique d'égalité basée sur la valeur. Pour appliquer la sémantique des valeurs, le compilateur génère plusieurs méthodes pour votre type d'enregistrement (à la fois pour les types `record class` et les types `record struct`) :

- Un remplacement de Object.Equals(Object).

- Une méthode virtuelle `Equals` dont le paramètre est le type d'enregistrement.
- Un remplacement de Object.GetHashCode().
- Des méthodes pour `operator ==` et `operator !=`.
- Les types d'enregistrements implémentent System.IEquatable<T>.

Les enregistrements fournissent également un remplacement de Object.ToString(). Le compilateur synthétise les méthodes d'affichage des enregistrements à l'aide de Object.ToString(). Vous allez explorer ces membres lorsque vous écrivez le code de ce didacticiel. Les enregistrements prennent en charge les expressions `with` pour activer la mutation non destructrice des enregistrements.

Vous pouvez également déclarer des *enregistrements positionnels* à l'aide d'une syntaxe plus concise. Le compilateur synthétise plus de méthodes pour vous lorsque vous déclarez des enregistrements positionnels :

- Un constructeur principal dont les paramètres correspondent aux paramètres positionnels sur la déclaration d'enregistrement.
- Des propriétés publiques pour chaque paramètre d'un constructeur principal. Ces propriétés sont *init uniquement* pour les types `record class` et les types `readonly record struct`. Pour les types `record struct`, elles sont en *lecture-écriture*.
- Une méthode `Deconstruct` permettant d'extraire les propriétés de l'enregistrement.

Générer des données de température

Les données et les statistiques font partie des scénarios dans lesquels vous souhaitez utiliser des enregistrements. Pour ce didacticiel, vous allez créer une application qui calcule les *degrés jours* pour différentes utilisations. *Les degrés jours* sont une mesure de la chaleur (ou de l'absence de chaleur) sur une période de jours, de semaines ou de mois. Les degrés jours permettent de suivre et de prévoir la consommation d'énergie. Des jours plus chauds signifient plus de climatisation, et des jours plus froids signifient plus d'utilisation de la chaudière. Les degrés jours aident à gérer les populations de plantes et à mettre en corrélation la croissance des plantes au fil des saisons. Les degrés jours permettent de suivre les migrations animales pour les espèces qui se déplacent en fonction du climat.

La formule est basée sur la température moyenne sur un jour donné et une température de référence. Pour calculer les degrés jours au fil du temps, vous aurez besoin de la température maximale et minimale quotidienne pendant une période de temps. Commençons par créer une application. Créer une application console. Créez un type d'enregistrement dans un nouveau fichier nommé « DailyTemperature.cs » :

C#

```
public readonly record struct DailyTemperature(double HighTemp, double
LowTemp);
```

Le code précédent définit un *enregistrement positionnel*. L'enregistrement `DailyTemperature` est une `readonly record struct`, car vous n'avez pas l'intention d'en hériter et il doit être immuable. Les propriétés `HighTemp` et `LowTemp` sont *des propriétés init uniquement*, ce qui signifie qu'elles peuvent être définies dans le constructeur ou à l'aide d'un initialiseur de propriété. Si vous souhaitez que les paramètres positionnels soient en lecture-écriture, vous déclarez une `record struct` au lieu d'une `readonly record struct`. Le type `DailyTemperature` a également un *constructeur principal* qui a deux paramètres qui correspondent aux deux propriétés. Vous utilisez le constructeur principal pour initialiser un enregistrement de `DailyTemperature`. Le code suivant crée et initialise plusieurs enregistrements de `DailyTemperature`. Le premier utilise des paramètres nommés pour clarifier le `HighTemp` et `LowTemp`. Les initialiseurs restants utilisent des paramètres positionnels pour initialiser la `HighTemp` et la `LowTemp` :

C#

```
private static DailyTemperature[] data = new DailyTemperature[]
{
    new DailyTemperature(HighTemp: 57, LowTemp: 30),
    new DailyTemperature(60, 35),
    new DailyTemperature(63, 33),
    new DailyTemperature(68, 29),
    new DailyTemperature(72, 47),
    new DailyTemperature(75, 55),
    new DailyTemperature(77, 55),
    new DailyTemperature(72, 58),
    new DailyTemperature(70, 47),
    new DailyTemperature(77, 59),
    new DailyTemperature(85, 65),
    new DailyTemperature(87, 65),
    new DailyTemperature(85, 72),
    new DailyTemperature(83, 68),
    new DailyTemperature(77, 65),
    new DailyTemperature(72, 58),
    new DailyTemperature(77, 55),
    new DailyTemperature(76, 53),
    new DailyTemperature(80, 60),
    new DailyTemperature(85, 66)
};
```

Vous pouvez ajouter vos propres propriétés ou méthodes aux enregistrements, y compris des enregistrements positionnels. Vous devez calculer la température moyenne

pour chaque jour. Vous pouvez ajouter cette propriété à l'enregistrement `DailyTemperature` :

C#

```
public readonly record struct DailyTemperature(double HighTemp, double
LowTemp)
{
    public double Mean => (HighTemp + LowTemp) / 2.0;
}
```

Assurons-nous que vous pouvez utiliser ces données. Ajoutez le code suivant à votre méthode `Main` :

C#

```
foreach (var item in data)
    Console.WriteLine(item);
```

Exécutez votre application et vous verrez un résultat semblable à l'affichage suivant (plusieurs lignes supprimées pour l'espace) :

CLI .NET

```
DailyTemperature { HighTemp = 57, LowTemp = 30, Mean = 43.5 }
DailyTemperature { HighTemp = 60, LowTemp = 35, Mean = 47.5 }


DailyTemperature { HighTemp = 80, LowTemp = 60, Mean = 70 }
DailyTemperature { HighTemp = 85, LowTemp = 66, Mean = 75.5 }
```

Le code précédent montre le résultat du remplacement de `ToString` synthétisé par le compilateur. Si vous préférez un texte différent, vous pouvez écrire votre propre version de `ToString` qui empêche le compilateur de synthétiser une version pour vous.

Calculer les degrés jours

Pour calculer les degrés jours, vous prenez la différence entre une température de référence et la température moyenne d'un jour donné. Pour mesurer la chaleur au fil du temps, vous ignorez tous les jours où la température moyenne est inférieure à la référence. Pour mesurer le froid au fil du temps, vous ignorez tous les jours où la température moyenne est supérieure à la référence. Par exemple, les États-Unis utilisent 65 F (18,3° C) comme référence pour les degrés jours de chauffage et de climatisation. C'est la température à laquelle il n'est pas nécessaire d'utiliser le chauffage ou la

climatisation. Si un jour a une température moyenne de 70 F (21,1° C), ce jour nécessite cinq degrés jours de climatisation et zéro degré jour de chauffage. À l'inverse, si la température moyenne est de 55 F (12,7° C), ce jour nécessite 10 degrés jours de chauffage et 0 degré jour de climatisation.

Vous pouvez exprimer ces formules sous la forme d'une petite hiérarchie de types d'enregistrements : un type de degré jour abstrait et deux types de degré jour concrets pour les degrés jours de chauffage et les degrés jours de climatisation. Ces types peuvent également être des enregistrements positionnels. Ils prennent une température de référence et une séquence d'enregistrements de température quotidiens comme arguments pour le constructeur principal :

C#

```
public abstract record DegreeDays(double BaseTemperature,
IEnumerable<DailyTemperature> TempRecords);

public sealed record HeatingDegreeDays(double BaseTemperature,
IEnumerable<DailyTemperature> TempRecords)
    : DegreeDays(BaseTemperature, TempRecords)
{
    public double DegreeDays => TempRecords.Where(s => s.Mean <
BaseTemperature).Sum(s => BaseTemperature - s.Mean);
}

public sealed record CoolingDegreeDays(double BaseTemperature,
IEnumerable<DailyTemperature> TempRecords)
    : DegreeDays(BaseTemperature, TempRecords)
{
    public double DegreeDays => TempRecords.Where(s => s.Mean >
BaseTemperature).Sum(s => s.Mean - BaseTemperature);
}
```

L'enregistrement de `DegreeDays` abstrait est la classe de base partagée pour à la fois les enregistrements de `HeatingDegreeDays` et les enregistrements de `CoolingDegreeDays`. Les déclarations du constructeur principal sur les enregistrements dérivés montrent comment gérer l'initialisation des enregistrements de référence. Votre enregistrement dérivé déclare des paramètres pour tous les paramètres du constructeur principal de l'enregistrement de référence. L'enregistrement de référence déclare et initialise ces propriétés. L'enregistrement dérivé ne les masque pas, mais crée et initialise uniquement les propriétés des paramètres qui ne sont pas déclarés dans son enregistrement de référence. Dans cet exemple, les enregistrements dérivés n'ajoutent pas de nouveaux paramètres de constructeur principal. Testez votre code en ajoutant le code suivant à votre méthode `Main` :

C#

```
var heatingDegreeDays = new HeatingDegreeDays(65, data);
Console.WriteLine(heatingDegreeDays);

var coolingDegreeDays = new CoolingDegreeDays(65, data);
Console.WriteLine(coolingDegreeDays);
```

Vous obtiendrez un résultat similaire à l'affichage suivant :

CLI .NET

```
HeatingDegreeDays { BaseTemperature = 65, TempRecords =
record_types.DailyTemperature[], DegreeDays = 85 }
CoolingDegreeDays { BaseTemperature = 65, TempRecords =
record_types.DailyTemperature[], DegreeDays = 71.5 }
```

Définir des méthodes synthétisées par le compilateur

Votre code calcule le nombre correct de degrés jours de chauffage et de climatisation au cours de cette période. Toutefois, cet exemple montre pourquoi vous souhaiterez peut-être remplacer certaines des méthodes synthétisées pour les enregistrements. Vous pouvez déclarer votre propre version de l'une des méthodes synthétisées par le compilateur dans un type d'enregistrement, à l'exception de la méthode clone. La méthode clone a un nom généré par le compilateur et vous ne pouvez pas fournir d'implémentation différente. Ces méthodes synthétisées incluent un constructeur de copie, les membres de l'interface System.IEquatable<T>, l'égalité et les tests d'inégalité, et GetHashCode(). À cet effet, vous allez synthétiser `PrintMembers`. Vous pouvez également déclarer votre propre `ToString`, mais `PrintMembers` offre une meilleure option pour les scénarios d'héritage. Pour fournir votre propre version d'une méthode synthétisée, la signature doit correspondre à la méthode synthétisée.

L'élément `TempRecords` du résultat de la console n'est pas utile. Il affiche le type, mais rien d'autre. Vous pouvez modifier ce comportement en fournissant votre propre implémentation de la méthode synthétisée `PrintMembers`. La signature dépend des modificateurs appliqués à la déclaration `record` :

- Si un type d'enregistrement est `sealed` ou une `record struct`, la signature est `private bool PrintMembers(StringBuilder builder);`
- Si un type d'enregistrement n'est pas `sealed` et dérive de `object` (autrement dit, il ne déclare pas d'enregistrement de référence), la signature est `protected virtual bool PrintMembers(StringBuilder builder);`

- Si un type d'enregistrement n'est pas `sealed` et dérive d'un autre enregistrement, la signature est `protected override bool PrintMembers(StringBuilder builder);`

Ces règles sont plus faciles à comprendre par le biais de la compréhension de l'objectif de `PrintMembers`. `PrintMembers` ajoute des informations sur chaque propriété d'un type d'enregistrement à une chaîne. Le contrat nécessite des enregistrements de référence pour ajouter leurs membres à l'affichage et suppose que les membres dérivés ajouteront leurs membres. Chaque type d'enregistrement synthétise un remplacement de `ToString` semblable à l'exemple suivant pour `HeatingDegreeDays` :

C#

```
public override string ToString()
{
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.Append("HeatingDegreeDays");
    stringBuilder.Append(" { ");
    if (PrintMembers(stringBuilder))
    {
        stringBuilder.Append(" ");
    }
    stringBuilder.Append("}");
    return stringBuilder.ToString();
}
```

Vous déclarez une méthode `PrintMembers` dans l'enregistrement `DegreeDays` qui n'imprime pas le type de la collection :

C#

```
protected virtual bool PrintMembers(StringBuilder stringBuilder)
{
    stringBuilder.Append($"BaseTemperature = {BaseTemperature}");
    return true;
}
```

La signature déclare une méthode `virtual protected` pour correspondre à la version du compilateur. Ne vous inquiétez pas si vous recevez mal les accesseurs ; le langage applique la signature correcte. Si vous oubliez les modificateurs corrects pour n'importe quelle méthode synthétisée, le compilateur émet des avertissements ou des erreurs qui vous aident à obtenir la bonne signature.

En C# 10 et versions ultérieures, vous pouvez déclarer la méthode `ToString` comme `sealed` dans un type d'enregistrement. Cela empêche les enregistrements dérivés de fournir une nouvelle implémentation. Les enregistrements dérivés contiendront toujours

le remplacement de `PrintMembers`. Vous devez sceller `ToString` si vous ne souhaitez pas qu'il affiche le type d'exécution de l'enregistrement. Dans l'exemple précédent, vous perdez les informations sur l'endroit où l'enregistrement mesure les degrés jours de chauffage ou de climatisation.

Mutation non destructrice

Les membres synthétisés dans une classe d'enregistrement positionnel ne modifient pas l'état de l'enregistrement. L'objectif est que vous pouvez créer plus facilement des enregistrements immuables. N'oubliez pas que vous déclarez une `readonly record struct` pour créer une struct d'enregistrement immuable. Examinez à nouveau les déclarations précédentes pour `HeatingDegreeDays` et `CoolingDegreeDays`. Les membres ajoutés effectuent des calculs sur les valeurs de l'enregistrement, mais ne mutent pas l'état. Les enregistrements positionnels facilitent la création de types de référence immuables.

La création de types de référence immuables signifie que vous souhaiterez utiliser une mutation non destructrice. Vous créez de nouvelles instances d'enregistrement similaires aux instances d'enregistrement existantes à l'aide d'expressions with. Ces expressions sont une construction de copie avec des affectations supplémentaires qui modifient la copie. Le résultat est une nouvelle instance d'enregistrement où chaque propriété a été copiée à partir de l'enregistrement existant et éventuellement modifiée. L'enregistrement d'origine n'est pas modifié.

Ajoutons quelques fonctionnalités à votre programme qui illustrent les expressions `with`. Tout d'abord, créons un enregistrement pour calculer des degrés jours de croissance à l'aide des mêmes données. *Les degrés jours* de croissance utilisent généralement 41 F (5° C) comme base de référence et mesure les températures au-dessus de la référence. Pour utiliser les mêmes données, vous pouvez créer un enregistrement similaire aux `coolingDegreeDays`, mais avec une température de référence différente :

C#

```
// Growing degree days measure warming to determine plant growing rates
var growingDegreeDays = coolingDegreeDays with { BaseTemperature = 41 };
Console.WriteLine(growingDegreeDays);
```

Vous pouvez comparer le nombre de degrés calculés aux nombres générés avec une température de référence plus élevée. N'oubliez pas que les enregistrements sont des *types de référence* et que ces copies sont des copies superficielles. Le tableau des

données n'est pas copié, mais les deux enregistrements font référence aux mêmes données. Il s'agit d'un avantage dans un autre scénario. Pour les degrés jours de croissance, il est utile de suivre le total des cinq jours précédents. Vous pouvez créer de nouveaux enregistrements avec différentes données sources à l'aide d'expressions `with`. Le code suivant génère une collection de ces accumulations, puis affiche les valeurs :

C#

```
// showing moving accumulation of 5 days using range syntax
List<CoolingDegreeDays> movingAccumulation = new();
int rangeSize = (data.Length > 5) ? 5 : data.Length;
for (int start = 0; start < data.Length - rangeSize; start++)
{
    var fiveDayTotal = growingDegreeDays with { TempRecords = data[start..
(start + rangeSize)] };
    movingAccumulation.Add(fiveDayTotal);
}
Console.WriteLine();
Console.WriteLine("Total degree days in the last five days");
foreach(var item in movingAccumulation)
{
    Console.WriteLine(item);
}
```

Vous pouvez également utiliser des expressions `with` pour créer des copies d'enregistrements. Ne spécifiez aucune propriété entre les accolades de l'expression `with`. Cela signifie que vous devez créer une copie et ne pas modifier de propriétés :

C#

```
var growingDegreeDaysCopy = growingDegreeDays with { };
```

Exécutez l'application terminée pour afficher les résultats.

Résumé

Ce didacticiel a montré plusieurs aspects d'enregistrements. Les enregistrements fournissent une syntaxe concise pour les types où l'utilisation fondamentale stocke des données. Pour les classes orientées objet, l'utilisation fondamentale définit les responsabilités. Ce didacticiel s'est concentré sur les *enregistrements positionnels*, où vous pouvez utiliser une syntaxe concise pour déclarer les propriétés d'un enregistrement. Le compilateur synthétise plusieurs membres de l'enregistrement pour copier et comparer les enregistrements. Vous pouvez ajouter tous les autres membres dont vous avez besoin pour vos types d'enregistrements. Vous pouvez créer des types

d'enregistrements immuables sachant qu'aucun des membres générés par le compilateur ne mutera l'état. Et les expressions `with` facilitent la prise en charge des mutations non destructrices.

Les enregistrements ajoutent une autre façon de définir des types. Vous utilisez des définitions `class` pour créer des hiérarchies orientées objet qui se concentrent sur les responsabilités et le comportement des objets. Vous créez des types de `struct` pour les structures de données qui stockent les données et sont suffisamment petits pour les copier efficacement. Vous créez des types d'`record` lorsque vous souhaitez une égalité et une comparaison basées sur des valeurs, utiliser des variables de référence et ne souhaitez pas copier de valeurs. Vous créez des types de `record struct` lorsque vous souhaitez que les fonctionnalités des enregistrements pour un type qui est assez petit pour être copié efficacement.

Vous pouvez en savoir plus sur les enregistrements dans l'article de référence du langage C# pour le type d'enregistrement et la spécification de type d'enregistrement proposée et la spécification de struct d'enregistrement.

Didacticiel : explorez les idées à l'aide d'instructions de niveau supérieur pour générer du code au fur et à mesure que vous apprenez

Article • 10/05/2023

Dans ce tutoriel, vous apprendrez à :

- ✓ Découvrez les règles qui régissent votre utilisation des instructions de niveau supérieur.
- ✓ Utilisez des instructions de niveau supérieur pour explorer les algorithmes.
- ✓ Refactorisez des explorations en composants réutilisables.

Prérequis

Vous devez configurer votre ordinateur pour exécuter .NET 6, qui inclut le compilateur C# 10. Le compilateur C# 10 est disponible à partir de Visual Studio 2022 ou du SDK .NET 6.

Ce tutoriel suppose de connaître C# et .NET, y compris Visual Studio ou l'interface CLI .NET.

Commencez à explorer

Les instructions de niveau supérieur vous permettent d'éviter les étapes supplémentaires requises en plaçant le point d'entrée de votre programme dans une méthode statique dans une classe. Le point de départ classique d'une nouvelle application console ressemble au code suivant :

C#

```
using System;

namespace Application
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
```

```
        }
    }
}
```

Le code précédent est le résultat de l'exécution de la commande `dotnet new console` et de la création d'une application console. Ces 11 lignes ne contiennent qu'une seule ligne de code exécutable. Vous pouvez simplifier ce programme avec la nouvelle fonctionnalité d'instructions de niveau supérieur. Cela vous permet de supprimer toutes les lignes de ce programme, à l'exception de deux :

C#

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

> **ⓘ Important**
>
> Les modèles C# pour .NET 6 utilisent des *instructions de niveau supérieur*. Votre application ne correspond peut-être pas au code de cet article, si vous avez déjà effectué une mise à niveau vers .NET 6. Pour plus d'informations, consultez l'article **Les nouveaux modèles C# génèrent des instructions de niveau supérieur**.
>
> Le SDK .NET 6 ajoute également un ensemble de directives *implicites* `global using` pour les projets qui utilisent les kits SDK suivants :
>
> - Microsoft.NET.Sdk
> - Microsoft.NET.Sdk.Web
> - Microsoft.NET.Sdk.Worker
>
> Ces directives implicites `global using` incluent les espaces de noms les plus courants pour le type de projet.

Cette fonctionnalité simplifie ce qui est nécessaire pour commencer à explorer de nouvelles idées. Vous pouvez utiliser des instructions de niveau supérieur pour des scénarios de script ou pour explorer. Une fois que les principes de base fonctionnent, vous pouvez commencer à refactoriser le code et créer des méthodes, des classes ou d'autres assemblys pour les composants réutilisables que vous avez créés. Les instructions de niveau supérieur permettent une expérimentation rapide et des didacticiels pour débutants. Ils offrent également un parcours fluide de l'expérimentation, jusqu'aux programmes complets.

Les instructions de niveau supérieur sont exécutées dans leur ordre d'apparition dans le fichier. Les instructions de niveau supérieur ne peuvent être utilisées que dans un seul fichier source dans votre application. Le compilateur génère une erreur si vous les utilisez dans plusieurs fichiers.

Créer un répondeur .NET magique

Pour ce tutoriel, créons une application console qui répond à une question « oui » ou « non » par une réponse aléatoire. Vous allez générer la fonctionnalité étape par étape. Vous pouvez vous concentrer sur votre tâche plutôt que sur les étapes nécessaires à la structure d'un programme classique. Ensuite, une fois que vous êtes satisfait(e) de la fonctionnalité, vous pouvez refactoriser l'application comme bon vous semble.

Un bon point de départ consiste à réécrire la question dans la console. Vous pouvez commencer par écrire le code suivant :

C#

```
Console.WriteLine(args);
```

Vous ne déclarez pas de variable `args`. Pour le fichier source unique qui contient vos instructions de niveau supérieur, le compilateur reconnaît qu'il s'agit des arguments `args` de ligne de commande. Le type d'args est un `string[]`, comme dans tous les programmes C#.

Vous pouvez tester votre code en exécutant la commande `dotnet run` suivante :

CLI .NET

```
dotnet run -- Should I use top level statements in all my programs?
```

Les arguments après le `--` sur la ligne de commande sont passés au programme. Vous pouvez voir le type de la variable `args`, car il s'agit de ce qui est imprimé sur la console :

Console

```
System.String[]
```

Pour écrire la question dans la console, vous devez énumérer les arguments et les séparer avec un espace. Remplacez l'appel `WriteLine` par le code suivant :

C#

```
Console.WriteLine();
foreach(var s in args)
{
    Console.Write(s);
    Console.Write(' ');
}
Console.WriteLine();
```

Maintenant, lorsque vous exécutez le programme, il affiche correctement la question sous la forme d'une chaîne d'arguments.

Répondre avec une réponse aléatoire

Après avoir renvoyé en écho la question, vous pouvez ajouter le code pour générer la réponse aléatoire. Commencez par ajouter un tableau de réponses possibles :

C#

```
string[] answers =
{
    "It is certain.",       "Reply hazy, try again.",     "Don't count on
it.",
    "It is decidedly so.",  "Ask again later.",           "My reply is no.",
    "Without a doubt.",     "Better not tell you now.",   "My sources say
no.",
    "Yes – definitely.",    "Cannot predict now.",        "Outlook not so
good.",
    "You may rely on it.",  "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
};
```

Ce tableau contient dix réponses affirmatives, cinq sans engagement et cinq qui sont négatives. Ensuite, ajoutez le code suivant pour générer et afficher une réponse aléatoire à partir du tableau :

C#

```
var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

Vous pouvez réexécuter l'application pour afficher les résultats. Vous devriez voir une sortie semblable à la suivante :

CLI .NET

```
dotnet run -- Should I use top level statements in all my programs?

Should I use top level statements in all my programs?
Better not tell you now.
```

Ce code répond aux questions, mais ajoutons une fonctionnalité supplémentaire. Vous souhaitez que votre application de questions simule la réflexion sur la réponse. Pour ce faire, ajoutez un peu d'animation ASCII et une pause pendant le traitement. Ajoutez le code suivant après la ligne qui renvoie en écho la question :

C#

```
for (int i = 0; i < 20; i++)
{
    Console.Write("| -");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("/ \\");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("- |");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("\\ /");
    await Task.Delay(50);
    Console.Write("\b\b\b");
}
Console.WriteLine();
```

Vous devez également ajouter une instruction `using` en haut du fichier source :

C#

```
using System.Threading.Tasks;
```

Les instructions `using` doivent être antérieures à toutes les autres instructions du fichier. Sinon, il s'agit d'une erreur du compilateur. Vous pouvez réexécuter le programme et voir l'animation. Cela crée une meilleure expérience. Expérimentez la longueur du délai en fonction de vos goûts.

Le code précédent crée un ensemble de lignes tournantes séparées par un espace. L'ajout du mot clé `await` indique au compilateur de générer le point d'entrée du programme en tant que méthode qui a le modificateur `async` et retourne un System.Threading.Tasks.Task. Ce programme ne retourne pas de valeur, de sorte que le

point d'entrée du programme retourne un `Task`. Si votre programme retourne une valeur entière, vous ajoutez une instruction de retour à la fin de vos instructions de niveau supérieur. Cette instruction de retour spécifie la valeur entière à retourner. Si vos instructions de niveau supérieur incluent une expression `await`, le type de retour devient System.Threading.Tasks.Task<TResult>.

Refactorisation pour l'avenir

Votre programme doit ressembler au code suivant :

C#

```
Console.WriteLine();
foreach(var s in args)
{
    Console.Write(s);
    Console.Write(' ');
}
Console.WriteLine();

for (int i = 0; i < 20; i++)
{
    Console.Write("| -");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("/ \\");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("- |");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("\\ /");
    await Task.Delay(50);
    Console.Write("\b\b\b");
}
Console.WriteLine();

string[] answers =
{
    "It is certain.",       "Reply hazy, try again.",     "Don't count on
it.",
    "It is decidedly so.",  "Ask again later.",           "My reply is no.",
    "Without a doubt.",     "Better not tell you now.",   "My sources say
no.",
    "Yes - definitely.",    "Cannot predict now.",        "Outlook not so
good.",
    "You may rely on it.",  "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
```

```
    "Signs point to yes.",
};

var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

Le code précédent est raisonnable. Il fonctionne. Mais il n'est pas réutilisable. Maintenant que l'application fonctionne, il est temps de retirer des parties réutilisables.

L'un des candidats est le code qui affiche l'animation d'attente. Cet extrait de code peut devenir une méthode :

Vous pouvez commencer en créant une fonction locale dans votre fichier. Remplacez l'animation actuelle par le code suivant :

C#

```
await ShowConsoleAnimation();

static async Task ShowConsoleAnimation()
{
    for (int i = 0; i < 20; i++)
    {
        Console.Write("| -");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("/ \\");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("- |");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("\\ /");
        await Task.Delay(50);
        Console.Write("\b\b\b");
    }
    Console.WriteLine();
}
```

Le code précédent crée une fonction locale à l'intérieur de votre méthode principale. Ceci n'est toujours pas réutilisable. Par conséquent, extrayez ce code dans une classe. Créez un nouveau fichier nommé *utilities.cs* , puis ajoutez le code suivant :

C#

```
namespace MyNamespace
{
    public static class Utilities
    {
        public static async Task ShowConsoleAnimation()
```

```
        {
            for (int i = 0; i < 20; i++)
            {
                Console.Write("| -");
                await Task.Delay(50);
                Console.Write("\b\b\b");
                Console.Write("/ \\");
                await Task.Delay(50);
                Console.Write("\b\b\b");
                Console.Write("- |");
                await Task.Delay(50);
                Console.Write("\b\b\b");
                Console.Write("\\ /");
                await Task.Delay(50);
                Console.Write("\b\b\b");
            }
            Console.WriteLine();
        }
    }
}
```

Un fichier qui a des instructions de niveau supérieur peut également contenir des espaces de noms et des types à la fin du fichier, après les instructions de niveau supérieur. Toutefois, pour ce didacticiel, vous placez la méthode d'animation dans un fichier distinct pour la rendre plus facilement réutilisable.

Enfin, vous pouvez nettoyer le code d'animation pour supprimer une duplication :

C#

```
foreach (string s in new[] { "| -", "/ \\", "- |", "\\ /", })
{
    Console.Write(s);
    await Task.Delay(50);
    Console.Write("\b\b\b");
}
```

Vous disposez maintenant d'une application complète et vous avez refactorisé les parties réutilisables pour une utilisation ultérieure. Vous pouvez appeler la nouvelle méthode utilitaire à partir de vos instructions de niveau supérieur, comme indiqué ci-dessous dans la version terminée du programme principal :

C#

```
using MyNamespace;

Console.WriteLine();
foreach(var s in args)
{
```

```
    Console.Write(s);
    Console.Write(' ');
}
Console.WriteLine();

await Utilities.ShowConsoleAnimation();

string[] answers =
{
    "It is certain.",       "Reply hazy, try again.",     "Don't count on
it.",
    "It is decidedly so.",  "Ask again later.",           "My reply is no.",
    "Without a doubt.",     "Better not tell you now.",   "My sources say
no.",
    "Yes – definitely.",    "Cannot predict now.",        "Outlook not so
good.",
    "You may rely on it.",  "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
};

var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

L'exemple précédent ajoute l'appel à `Utilities.ShowConsoleAnimation` et ajoute une instruction supplémentaire `using`.

Résumé

Les instructions de niveau supérieur facilitent la création de programmes simples à utiliser pour explorer de nouveaux algorithmes. Vous pouvez expérimenter des algorithmes en essayant différents extraits de code. Une fois que vous avez appris ce qui fonctionne, vous pouvez refactoriser le code pour qu'il soit plus facile à gérer.

Les instructions de niveau supérieur simplifient les programmes qui sont basés sur des applications console. Il s'agit notamment des fonctions Azure, des actions GitHub et d'autres petits utilitaires. Pour plus d'informations, consultez Instructions de niveau supérieur (Guide de programmation C#).

Utiliser la correspondance de modèle pour générer votre comportement de classe, afin d'obtenir un meilleur code

Article • 09/05/2023

Les fonctionnalités de correspondance de modèle en C# fournissent une syntaxe pour exprimer vos algorithmes. Vous pouvez utiliser ces techniques pour implémenter le comportement dans vos classes. Vous pouvez combiner la conception de classes orientée objet avec une implémentation orientée données, afin de fournir du code concis lors de la modélisation d'objets réels.

Dans ce tutoriel, vous apprendrez à :

- ✓ Exprimez vos classes orientées objet à l'aide de modèles de données.
- ✓ Implémentez ces modèles à l'aide des fonctionnalités de correspondance de modèle de C#.
- ✓ Tirez parti des diagnostics du compilateur pour valider votre implémentation.

Prérequis

Vous devrez configurer votre ordinateur de sorte qu'il exécute .NET 5, avec le compilateur C# 9. Le compilateur C# 9 est disponible à partir de Visual Studio 2019 version 16.8 ou du SDK .NET 5.

Générer une simulation d'une écluse de canal

Dans ce didacticiel, vous allez générer une classe C# qui simule une écluse de canal. En bref, une écluse de canal est un dispositif qui élève et abaisse les bateaux qui se déplacent entre deux étendues d'eau à différents niveaux. Une écluse possède deux portes et un mécanisme permettant de changer le niveau de l'eau.

En fonctionnement normal, un bateau entre dans l'une des portes, tandis que le niveau de l'eau dans l'écluse correspond au niveau de l'eau sur le côté où le bateau entre. Une fois dans l'écluse, le niveau de l'eau est modifié pour correspondre au niveau d'eau là où le bateau quittera l'écluse. Une fois que le niveau de l'eau correspond à ce côté, la porte du côté de la sortie s'ouvre. Les mesures de sécurité permettent de s'assurer qu'un opérateur ne peut pas créer de situation dangereuse dans le canal. Le niveau de l'eau ne peut être modifié que lorsque les deux portes sont fermées. Au plus, seule une porte

peut être ouverte. Pour ouvrir une porte, le niveau de l'eau dans l'écluse doit correspondre au niveau de l'eau à l'extérieur de la porte ouverte.

Vous pouvez générer une classe C# pour modéliser ce comportement. Une classe `CanalLock` prend en charge les commandes permettant d'ouvrir ou de fermer l'une ou l'autre des portes. Elle aurait d'autres commandes pour élever ou abaisser l'eau. La classe doit également prendre en charge les propriétés pour lire l'état actuel des deux portes et du niveau d'eau. Vos méthodes implémentent les mesures de sécurité.

Définir une classe

Vous allez générer une application console pour tester votre classe `CanalLock`. Créez un nouveau projet de console pour .NET 5 à l'aide de Visual Studio ou de CLI .NET. Puis, ajoutez une nouvelle classe et nommez-la `CanalLock`. Ensuite, concevez votre API publique, mais laissez les méthodes non implémentées :

C#

```
public enum WaterLevel
{
    Low,
    High
}
public class CanalLock
{
    // Query canal lock state:
    public WaterLevel CanalLockWaterLevel { get; private set; } =
WaterLevel.Low;
    public bool HighWaterGateOpen { get; private set; } = false;
    public bool LowWaterGateOpen { get; private set; } = false;

    // Change the upper gate.
    public void SetHighGate(bool open)
    {
        throw new NotImplementedException();
    }

    // Change the lower gate.
    public void SetLowGate(bool open)
    {
        throw new NotImplementedException();
    }

    // Change water level.
    public void SetWaterLevel(WaterLevel newLevel)
    {
        throw new NotImplementedException();
    }
```

```
    public override string ToString() =>
        $"The lower gate is {(LowWaterGateOpen ? "Open" : "Closed")}. " +
        $"The upper gate is {(HighWaterGateOpen ? "Open" : "Closed")}. " +
        $"The water level is {CanalLockWaterLevel}.";
}
```

Le code précédent initialise l'objet afin que les deux portes soient fermées et que le niveau d'eau soit bas. Ensuite, écrivez le code de test suivant dans votre méthode `Main` pour vous guider lors de la création d'une première implémentation de la classe :

C#

```
// Create a new canal lock:
var canalGate = new CanalLock();

// State should be doors closed, water level low:
Console.WriteLine(canalGate);

canalGate.SetLowGate(open: true);
Console.WriteLine($"Open the lower gate:  {canalGate}");

Console.WriteLine("Boat enters lock from lower gate");

canalGate.SetLowGate(open: false);
Console.WriteLine($"Close the lower gate:  {canalGate}");

canalGate.SetWaterLevel(WaterLevel.High);
Console.WriteLine($"Raise the water level: {canalGate}");

canalGate.SetHighGate(open: true);
Console.WriteLine($"Open the higher gate:  {canalGate}");

Console.WriteLine("Boat exits lock at upper gate");
Console.WriteLine("Boat enters lock from upper gate");

canalGate.SetHighGate(open: false);
Console.WriteLine($"Close the higher gate: {canalGate}");

canalGate.SetWaterLevel(WaterLevel.Low);
Console.WriteLine($"Lower the water level: {canalGate}");

canalGate.SetLowGate(open: true);
Console.WriteLine($"Open the lower gate:  {canalGate}");

Console.WriteLine("Boat exits lock at upper gate");

canalGate.SetLowGate(open: false);
Console.WriteLine($"Close the lower gate:  {canalGate}");
```

Ensuite, ajoutez une première implémentation de chaque méthode dans la classe `CanalLock`. Le code suivant implémente les méthodes de la classe sans se soucier des

règles de sécurité. Vous ajouterez des tests de sécurité ultérieurement :

C#

```
// Change the upper gate.
public void SetHighGate(bool open)
{
    HighWaterGateOpen = open;
}

// Change the lower gate.
public void SetLowGate(bool open)
{
    LowWaterGateOpen = open;
}

// Change water level.
public void SetWaterLevel(WaterLevel newLevel)
{
    CanalLockWaterLevel = newLevel;
}
```

Les tests que vous avez écrits jusqu'à présent réussissent. Vous avez implémenté les principes de base. À présent, écrivez un test pour la première condition d'échec. À la fin des essais précédents, les deux portes sont fermées et le niveau de l'eau est défini sur bas. Ajoutez un test pour essayer d'ouvrir la porte supérieure :

C#

```
Console.WriteLine("=============================================");
Console.WriteLine("     Test invalid commands");
// Open "wrong" gate (2 tests)
try
{
    canalGate = new CanalLock();
    canalGate.SetHighGate(open: true);
}
catch (InvalidOperationException)
{
    Console.WriteLine("Invalid operation: Can't open the high gate. Water is
low.");
}
Console.WriteLine($"Try to open upper gate: {canalGate}");
```

Ce test échoue, car la porte s'ouvre. En tant que première implémentation, vous pouvez la corriger avec le code suivant :

C#

```
// Change the upper gate.
public void SetHighGate(bool open)
{
    if (open && (CanalLockWaterLevel == WaterLevel.High))
        HighWaterGateOpen = true;
    else if (open && (CanalLockWaterLevel == WaterLevel.Low))
        throw new InvalidOperationException("Cannot open high gate when the
water is low");
}
```

Vos tests réussissent. Toutefois, au fur et à mesure que vous ajouterez d'autres tests, vous ajouterez de plus en plus de `if` clauses et vous testerez différentes propriétés. Rapidement, ces méthodes deviendront trop compliquées à mesure que vous ajouterez des instructions conditionnelles.

Implémenter les commandes avec des modèles

Il vaut mieux utiliser *des modèles* pour déterminer si l'objet est dans un état valide pour exécuter une commande. Vous pouvez exprimer si une commande est autorisée en fonction de trois variables : l'état de la porte, le niveau de l'eau et le nouveau paramètre :

Nouveau paramètre	État de la porte	Niveau de l'eau	Résultats
Fermé	Fermé	Élevé	Fermé
Fermé	Fermé	Faible	Fermé
Fermé	Ouvrir	Élevé	Fermés
~~Fermés~~	~~Ouvrir~~	~~Low~~	~~Fermés~~
Ouvrir	Fermés	Élevé	Ouvrir
Ouvrir	Fermés	Faible	Fermé (erreur)
Ouvrir	Ouvrir	Élevé	Ouvrir
~~Ouvrir~~	~~Ouvrir~~	~~Low~~	~~Fermé (erreur)~~

Les quatrième et dernière lignes du tableau ont été barrées dans le texte, car elles sont non valides. Le code que vous ajoutez maintenant doit s'assurer que la porte d'eau haute n'est jamais ouverte lorsque l'eau est basse. Ces états peuvent être codés en tant qu'expression de commutateur unique (n'oubliez pas que `false` indique « Fermé ») :

```
C#
```

```
HighWaterGateOpen = (open, HighWaterGateOpen, CanalLockWaterLevel) switch
{
    (false, false, WaterLevel.High) => false,
    (false, false, WaterLevel.Low) => false,
    (false, true, WaterLevel.High) => false,
    (false, true, WaterLevel.Low) => false, // should never happen
    (true, false, WaterLevel.High) => true,
    (true, false, WaterLevel.Low) => throw new
InvalidOperationException("Cannot open high gate when the water is low"),
    (true, true, WaterLevel.High) => true,
    (true, true, WaterLevel.Low) => false, // should never happen
};
```

Essayez cette version. Vos tests réussissent, ce qui valide le code. Le tableau complet présente les combinaisons possibles d'entrées et de résultats. Cela signifie que vous et les autres développeurs pouvez rapidement consulter le tableau et voir que vous avez couvert toutes les entrées possibles. Encore plus facile, le compilateur peut également vous aider. Après avoir ajouté le code précédent, vous pouvez voir que le compilateur génère un avertissement : *CS8524* indique que l'expression de commutateur ne couvre pas toutes les entrées possibles. La raison de cet avertissement est que l'une des entrées est un type `enum`. Le compilateur interprète « toutes les entrées possibles » comme toutes les entrées du type sous-jacent, généralement une `int`. Cette expression `switch` vérifie uniquement les valeurs déclarées dans l'`enum`. Pour supprimer l'avertissement, vous pouvez ajouter un modèle d'abandon fourre-tout pour la dernière branche de l'expression. Cette condition lève une exception, car elle indique une entrée non valide :

```
C#

_  => throw new InvalidOperationException("Invalid internal state"),
```

La branche de commutateur précédente doit être la dernière de votre expression `switch`, car elle correspond à toutes les entrées. Expérimentez en le déplaçant plus tôt dans l'ordre. Cela provoque une erreur du compilateur *CS8510* pour le code inaccessible dans un modèle. La structure naturelle des expressions de commutateur permet au compilateur de générer des erreurs et des avertissements en cas d'erreurs possibles. Le « filet de sécurité » du compilateur vous permet de créer plus facilement du code correct en moins d'itérations, et vous donne la liberté de combiner des branches de commutateur avec des caractères génériques. Le compilateur émet des erreurs si votre combinaison entraîne des branches inaccessibles que vous n'attendiez pas, et des avertissements si vous supprimez une branche nécessaire.

Le premier changement consiste à combiner toutes les branches où la commande consiste à fermer la porte ; ceci est toujours autorisé. Ajoutez le code suivant comme

première branche dans votre expression de commutateur :

```
C#
```

```
(false, _, _) => false,
```

Après avoir ajouté la branche de commutateur précédente, vous obtenez quatre erreurs du compilateur, une sur chacune des branche où la commande est `false`. Ces branches sont déjà couvertes par la branche nouvellement ajoutée. Vous pouvez supprimer ces quatre lignes en toute sécurité. Vous avez prévu que cette nouvelle branche de commutateur remplace ces conditions.

Ensuite, vous pouvez simplifier les quatre branche où la commande est d'ouvrir la porte. Dans les deux cas où le niveau d'eau est haut, la porte peut être ouverte. (Dans l'un des cas, la porte est déjà ouverte.) Un cas où le niveau d'eau est bas lève une exception, et l'autre ne devrait pas se produire. Lever la même exception devrait être sûr, si l'écluse est déjà dans un état non valide. Vous pouvez effectuer les simplifications suivantes pour ces branches :

```
C#
```

```
(true, _, WaterLevel.High) => true,
(true, false, WaterLevel.Low) => throw new InvalidOperationException("Cannot
open high gate when the water is low"),
_ => throw new InvalidOperationException("Invalid internal state"),
```

Exécutez à nouveau vos tests, et ils réussissent. Voici la version finale de la méthode `SetHighGate` :

```
C#
```

```
// Change the upper gate.
public void SetHighGate(bool open)
{
    HighWaterGateOpen = (open, HighWaterGateOpen, CanalLockWaterLevel)
switch
    {
        (false, _,     _)               => false,
        (true, _,      WaterLevel.High) => true,
        (true, false, WaterLevel.Low)   => throw new
InvalidOperationException("Cannot open high gate when the water is low"),
        _                               => throw new
InvalidOperationException("Invalid internal state"),
    };
}
```

Implémenter des modèles vous-même

Maintenant que vous avez vu la technique, remplissez les méthodes `SetLowGate` et `SetWaterLevel` vous-même. Commencez par ajouter le code suivant pour tester des opérations non valides sur ces méthodes :

C#

```
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetWaterLevel(WaterLevel.High);
    canalGate.SetLowGate(open: true);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't open the lower gate. Water
is high.");
}
Console.WriteLine($"Try to open lower gate: {canalGate}");
// change water level with gate open (2 tests)
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetLowGate(open: true);
    canalGate.SetWaterLevel(WaterLevel.High);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't raise water when the lower
gate is open.");
}
Console.WriteLine($"Try to raise water with lower gate open: {canalGate}");
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetWaterLevel(WaterLevel.High);
    canalGate.SetHighGate(open: true);
    canalGate.SetWaterLevel(WaterLevel.Low);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't lower water when the high
gate is open.");
}
Console.WriteLine($"Try to lower water with high gate open: {canalGate}");
```

Exécutez de nouveau votre application. Vous pouvez voir que les nouveaux tests échouent et que l'écluse de canal est dans un état non valide. Essayez d'implémenter les méthodes restantes vous-même. La méthode permettant de définir la porte inférieure doit être similaire à la méthode permettant de définir la porte supérieure. La méthode qui modifie le niveau d'eau a des vérifications différentes, mais doit suivre une structure similaire. Il pourrait être utile d'utiliser le même processus pour la méthode qui définit le niveau d'eau. Commencez par les quatre entrées : l'état des deux portes, l'état actuel du niveau d'eau et le nouveau niveau d'eau demandé. L'expression de commutateur doit commencer par :

C#

```
CanalLockWaterLevel = (newLevel, CanalLockWaterLevel, LowWaterGateOpen,
HighWaterGateOpen) switch
{
    // elided
};
```

Vous aurez 16 branches de commutateur à remplir en tout. Ensuite, testez et simplifiez.

Avez-vous réalisé des méthodes ressemblant à cela ?

C#

```
// Change the lower gate.
public void SetLowGate(bool open)
{
    LowWaterGateOpen = (open, LowWaterGateOpen, CanalLockWaterLevel) switch
    {
        (false, _, _) => false,
        (true, _, WaterLevel.Low) => true,
        (true, false, WaterLevel.High) => throw new
InvalidOperationException("Cannot open high gate when the water is low"),
        _ => throw new InvalidOperationException("Invalid internal state"),
    };
}

// Change water level.
public void SetWaterLevel(WaterLevel newLevel)
{
    CanalLockWaterLevel = (newLevel, CanalLockWaterLevel, LowWaterGateOpen,
HighWaterGateOpen) switch
    {
        (WaterLevel.Low, WaterLevel.Low, true, false) => WaterLevel.Low,
        (WaterLevel.High, WaterLevel.High, false, true) => WaterLevel.High,
        (WaterLevel.Low, _, false, false) => WaterLevel.Low,
        (WaterLevel.High, _, false, false) => WaterLevel.High,
        (WaterLevel.Low, WaterLevel.High, false, true) => throw new
```

```
InvalidOperationException("Cannot lower water when the high gate is open"),
        (WaterLevel.High, WaterLevel.Low, true, false) => throw new
InvalidOperationException("Cannot raise water when the low gate is open"),
        _ => throw new InvalidOperationException("Invalid internal state"),
    };
}
```

Vos tests doivent réussir et l'écluse du canal doit fonctionner en toute sécurité.

Résumé

Dans ce didacticiel, vous avez appris à utiliser la correspondance de modèle pour vérifier l'état interne d'un objet avant d'appliquer des modifications à cet état. Vous pouvez vérifier les combinaisons de propriétés. Une fois que vous avez créé des tableaux pour l'une de ces transitions, vous testez votre code, puis simplifiez la lisibilité et la maintenance. Ces refactorisations initiales peuvent suggérer d'autres refactorisations qui valident l'état interne ou gèrent d'autres modifications d'API. Ce didacticiel a combiné des classes et des objets avec une approche plus orientée données, basée sur des modèles, pour implémenter ces classes.

Tutoriel : Mettre à jour des interfaces avec les méthodes d'interface par défaut

Article • 24/03/2023

Vous pouvez définir une implémentation quand vous déclarez un membre d'une interface. Le scénario le plus courant consiste à ajouter de manière sécurisée des membres à une interface déjà publiée et utilisée par de nombreux clients.

Ce didacticiel vous montre comment effectuer les opérations suivantes :

- ✓ Étendre des interfaces de manière sécurisée en ajoutant des méthodes avec des implémentations
- ✓ Créer des implémentations paramétrables pour fournir une plus grande flexibilité
- ✓ Permettre aux implémenteurs de fournir une implémentation plus spécifique sous la forme d'un remplacement

Prérequis

Vous devez configurer votre machine pour qu'elle exécute .NET, y compris le compilateur C#. Le compilateur C# est accessible via Visual Studio 2022 ou le SDK .NET.

Présentation du scénario

Ce tutoriel commence par la version 1 d'une bibliothèque de relation client. Vous pouvez obtenir l'application de démarrage à partir de notre dépôt samples sur GitHub. L'entreprise qui a créé cette bibliothèque espérait la voir adoptée par les clients ayant des applications existantes. Elle a fourni des définitions d'interface minimales destinées à être implémentées par les utilisateurs de sa bibliothèque. Voici la définition d'interface pour un client :

C#

```
public interface ICustomer
{
    IEnumerable<IOrder> PreviousOrders { get; }

    DateTime DateJoined { get; }
    DateTime? LastOrder { get; }
```

```
    string Name { get; }
    IDictionary<DateTime, string> Reminders { get; }
}
```

Elle a défini une deuxième interface qui représente une commande :

```
C#

public interface IOrder
{
    DateTime Purchased { get; }
    decimal Cost { get; }
}
```

À partir de ces interfaces, l'équipe a pu générer une bibliothèque pour les utilisateurs permettant de créer une meilleure expérience pour ses clients. Son objectif était de créer une relation plus étroite avec les clients existants et d'améliorer ses relations avec les nouveaux clients.

À présent, il est temps de mettre à niveau la bibliothèque pour la prochaine version. L'une des fonctionnalités demandées permet une remise de fidélité pour les clients qui ont un grand nombre de commandes. Cette nouvelle remise de fidélité est appliquée chaque fois qu'un client passe une commande. La remise spécifique est une propriété de chaque client. Chaque implémentation de `ICustomer` peut définir des règles différentes pour la remise de fidélité.

La façon la plus naturelle d'ajouter cette fonctionnalité consiste à améliorer l'interface `ICustomer` avec une méthode pour appliquer une remise de fidélité. Cette suggestion de conception suscitait l'inquiétude des développeurs expérimentés : « Les interfaces sont immuables une fois qu'elles ont été publiées ! Ne faites pas de changement cassant ! » *Implémentations d'interface par défaut* pour la mise à niveau des interfaces. Les auteurs de bibliothèque peuvent ajouter de nouveaux membres à l'interface et fournir une implémentation par défaut pour ces membres.

Les implémentations d'interface par défaut permettent aux développeurs de mettre à niveau une interface tout en laissant la possibilité aux implémenteurs de substituer cette implémentation. Les utilisateurs de la bibliothèque peuvent accepter l'implémentation par défaut en tant que modification sans rupture. Si leurs règles métier sont différentes, ils peuvent opérer un remplacement.

Effectuer une mise à niveau avec des méthodes d'interface par défaut

L'équipe s'est mise d'accord sur l'implémentation par défaut la plus vraisemblable : une remise de fidélité pour les clients.

La mise à niveau doit fournir la fonctionnalité permettant de définir deux propriétés : le nombre de commandes nécessaires pour pouvoir bénéficier de la remise et le pourcentage de celle-ci. Ces fonctionnalités en font un scénario idéal pour les méthodes d'interface par défaut. Vous pouvez ajouter une méthode à l'interface `ICustomer` et fournir l'implémentation la plus probable. Toutes les implémentations existantes et nouvelles peuvent utiliser l'implémentation par défaut ou fournir la leur.

Tout d'abord, ajoutez la nouvelle méthode à l'interface, y compris le corps de la méthode :

C#

```
// Version 1:
public decimal ComputeLoyaltyDiscount()
{
    DateTime TwoYearsAgo = DateTime.Now.AddYears(-2);
    if ((DateJoined < TwoYearsAgo) && (PreviousOrders.Count() > 10))
    {
        return 0.10m;
    }
    return 0;
}
```

L'auteur de la bibliothèque a écrit un premier test pour vérifier l'implémentation :

C#

```
SampleCustomer c = new SampleCustomer("customer one", new DateTime(2010, 5,
31))
{
    Reminders =
    {
        { new DateTime(2010, 08, 12), "childs's birthday" },
        { new DateTime(1012, 11, 15), "anniversary" }
    }
};

SampleOrder o = new SampleOrder(new DateTime(2012, 6, 1), 5m);
c.AddOrder(o);

o = new SampleOrder(new DateTime(2103, 7, 4), 25m);
c.AddOrder(o);

// Check the discount:
ICustomer theCustomer = c;
```

```
Console.WriteLine($"Current discount:
{theCustomer.ComputeLoyaltyDiscount()}");
```

Notez la partie suivante du test :

C#

```
// Check the discount:
ICustomer theCustomer = c;
Console.WriteLine($"Current discount:
{theCustomer.ComputeLoyaltyDiscount()}");
```

Ce cast de `SampleCustomer` en `ICustomer` est nécessaire. La classe `SampleCustomer` n'a pas besoin de fournir une implémentation pour `ComputeLoyaltyDiscount` ; elle est fournie par l'interface `ICustomer`. Toutefois, la classe `SampleCustomer` n'hérite pas les membres de ses interfaces. Cette règle n'a pas changé. Pour qu'il soit possible d'appeler n'importe quelle méthode déclarée et implémentée dans l'interface, la variable doit être du même type que l'interface, `ICustomer` dans cet exemple.

Fournir le paramétrage

L'implémentation par défaut est trop restrictive. De nombreux consommateurs de ce système peuvent choisir des seuils différents pour le nombre d'achats, une durée différente pour la qualité de membre ou un pourcentage de remise différent. Vous pouvez procurer une meilleure expérience de mise à niveau à davantage de clients en fournissant un moyen de définir ces paramètres. Nous allons ajouter une méthode statique qui définit ces trois paramètres de contrôle de l'implémentation par défaut :

C#

```
// Version 2:
public static void SetLoyaltyThresholds(
    TimeSpan ago,
    int minimumOrders = 10,
    decimal percentageDiscount = 0.10m)
{
    length = ago;
    orderCount = minimumOrders;
    discountPercent = percentageDiscount;
}
private static TimeSpan length = new TimeSpan(365 * 2, 0,0,0); // two years
private static int orderCount = 10;
private static decimal discountPercent = 0.10m;

public decimal ComputeLoyaltyDiscount()
{
```

```
    DateTime start = DateTime.Now - length;

    if ((DateJoined < start) && (PreviousOrders.Count() > orderCount))
    {
        return discountPercent;
    }
    return 0;
}
```

Ce petit fragment de code montre de nombreuses nouvelles fonctionnalités de langage. Les interfaces peuvent maintenant inclure des membres statiques, notamment des champs et des méthodes. Différents modificateurs d'accès sont également activés. Les autres champs sont privés, la nouvelle méthode est publique. Tout modificateur est autorisé sur les membres d'interface.

Les applications qui utilisent la formule générale pour le calcul de la remise de fidélité, mais des paramètres différents, n'ont pas besoin de fournir une implémentation personnalisée ; elles peuvent définir les arguments par le biais d'une méthode statique. Par exemple, le code suivant définit une « évaluation du client » qui récompense tout client membre depuis plus d'un mois :

C#

```
ICustomer.SetLoyaltyThresholds(new TimeSpan(30, 0, 0, 0), 1, 0.25m);
Console.WriteLine($"Current discount:
{theCustomer.ComputeLoyaltyDiscount()}");
```

Étendre l'implémentation par défaut

Le code que vous avez ajouté jusqu'à présent a fourni une implémentation pratique pour les scénarios où les utilisateurs veulent quelque chose comme l'implémentation par défaut ou pour fournir un ensemble de règles non lié. En guise de dernière fonctionnalité, nous allons refactoriser le code un peu pour permettre les scénarios où les utilisateurs sont susceptibles de vouloir générer l'implémentation par défaut.

Imaginez une start-up qui souhaite attirer de nouveaux clients. Elle offre une remise de 50 % sur la première commande d'un nouveau client. Pour leur part, les clients existants bénéficient de la remise standard. L'auteur de la bibliothèque doit déplacer l'implémentation par défaut vers une méthode `protected static` afin que toute classe qui implémente cette interface puisse réutiliser le code dans son implémentation. L'implémentation par défaut du membre d'interface appelle également cette méthode partagée :

C#

```
public decimal ComputeLoyaltyDiscount() => DefaultLoyaltyDiscount(this);
protected static decimal DefaultLoyaltyDiscount(ICustomer c)
{
    DateTime start = DateTime.Now - length;

    if ((c.DateJoined < start) && (c.PreviousOrders.Count() > orderCount))
    {
        return discountPercent;
    }
    return 0;
}
```

Dans une implémentation d'une classe qui implémente cette interface, le remplacement peut appeler la méthode d'assistance statique et étendre cette logique pour fournir la remise « nouveau client » :

C#

```
public decimal ComputeLoyaltyDiscount()
{
   if (PreviousOrders.Any() == false)
        return 0.50m;
    else
        return ICustomer.DefaultLoyaltyDiscount(this);
}
```

Vous pouvez voir l'intégralité du code terminé dans notre dépôt d'exemples sur GitHub. Vous pouvez obtenir l'application de démarrage à partir de notre dépôt samples sur GitHub.

Ces nouvelles fonctionnalités signifient que les interfaces peuvent être mises à jour de manière sécurisée quand il existe une implémentation par défaut raisonnable pour les nouveaux membres. Concevez soigneusement les interfaces pour exprimer des idées fonctionnelles uniques implémentées par plusieurs classes. Cela facilite la mise à niveau de ces définitions d'interface quand de nouvelles exigences sont découvertes pour cette même idée fonctionnelle.

Tutoriel : Combiner les fonctionnalités lors de la création de classes à l'aide d'interfaces avec des méthodes d'interface par défaut

Article • 24/03/2023

Vous pouvez définir une implémentation quand vous déclarez un membre d'une interface. Cette fonctionnalité fournit de nouvelles fonctionnalités dans lesquelles vous pouvez définir des implémentations par défaut pour les fonctionnalités déclarées dans les interfaces. Les classes peuvent choisir quand remplacer les fonctionnalités, quand utiliser la fonctionnalité par défaut et quand ne pas déclarer la prise en charge des fonctionnalités discrètes.

Ce didacticiel vous montre comment effectuer les opérations suivantes :

- ✓ Créez des interfaces avec des implémentations qui décrivent des fonctionnalités discrètes.
- ✓ Créez des classes qui utilisent les implémentations par défaut.
- ✓ Créez des classes qui remplacent tout ou partie des implémentations par défaut.

Prérequis

Vous devez configurer votre machine pour qu'elle exécute .NET, y compris le compilateur C#. Le compilateur C# est accessible via Visual Studio 2022 ou le SDK .NET.

Limitations des méthodes d'extension

Une façon d'implémenter un comportement qui apparaît dans le cadre d'une interface consiste à définir des méthodes d'extension qui fournissent le comportement par défaut. Les interfaces déclarent un ensemble minimal de membres tout en fournissant une plus grande surface d'exposition pour toute classe qui implémente cette interface. Par exemple, les méthodes d'extension dans Enumerable fournissent l'implémentation pour que n'importe quelle séquence soit la source d'une requête LINQ.

Les méthodes d'extension sont résolues au moment de la compilation, à l'aide du type déclaré de la variable. Les classes qui implémentent l'interface peuvent fournir une meilleure implémentation pour n'importe quelle méthode d'extension. Les déclarations

de variables doivent correspondre au type d'implémentation pour permettre au compilateur de choisir cette implémentation. Lorsque le type au moment de la compilation correspond à l'interface, les appels de méthode sont résolus à la méthode d'extension. Une autre préoccupation avec les méthodes d'extension est que ces méthodes sont accessibles partout où la classe contenant les méthodes d'extension est accessible. Les classes ne peuvent pas déclarer si elles doivent ou non fournir des fonctionnalités déclarées dans les méthodes d'extension.

Vous pouvez déclarer les implémentations par défaut en tant que méthodes d'interface. Ensuite, chaque classe utilise automatiquement l'implémentation par défaut. Toute classe qui peut fournir une meilleure implémentation peut remplacer la définition de méthode d'interface par un meilleur algorithme. Dans un sens, cette technique ressemble à la façon dont vous pouvez utiliser des méthodes d'extension.

Dans cet article, vous allez découvrir comment les implémentations d'interface par défaut permettent de nouveaux scénarios.

Concevoir l'application

Envisagez une application domotique. Vous avez probablement de nombreux types différents de lumières et d'indicateurs qui pourraient être utilisés dans toute la maison. Chaque lumière doit prendre en charge les API pour les activer et les désactiver, et pour signaler l'état actuel. Certains voyants peuvent prendre en charge d'autres fonctionnalités, comme :

- Allumer la lumière, puis l'éteindre après une minuterie.
- Faire clignoter la lumière pendant un certain temps.

Certaines de ces fonctionnalités étendues peuvent être émulées dans les appareils qui prennent en charge l'ensemble minimal. Cela indique la fourniture d'une implémentation par défaut. Pour les appareils qui ont plus de fonctionnalités intégrées, le logiciel de l'appareil utilise les fonctionnalités natives. Pour les autres voyants, ils peuvent choisir d'implémenter l'interface et d'utiliser l'implémentation par défaut.

Les membres d'interface par défaut sont une meilleure solution pour ce scénario que les méthodes d'extension. Les auteurs de classes peuvent contrôler les interfaces qu'ils choisissent d'implémenter. Les interfaces qu'ils choisissent sont disponibles en tant que méthodes. En outre, étant donné que les méthodes d'interface par défaut sont virtuelles par défaut, la répartition de la méthode choisit toujours l'implémentation dans la classe.

Créons le code pour illustrer ces différences.

Créer des interfaces

Commencez par créer l'interface qui définit le comportement de toutes les lumières :

C#

```
public interface ILight
{
    void SwitchOn();
    void SwitchOff();
    bool IsOn();
}
```

Un luminaire de base peut implémenter cette interface, comme indiqué dans le code suivant :

C#

```
public class OverheadLight : ILight
{
    private bool isOn;
    public bool IsOn() => isOn;
    public void SwitchOff() => isOn = false;
    public void SwitchOn() => isOn = true;

    public override string ToString() => $"The light is {(isOn ? "on" :
"off")}";
}
```

Dans ce tutoriel, le code ne pilote pas les appareils IoT, mais émule ces activités en écrivant des messages dans la console. Vous pouvez explorer le code sans automatiser votre maison.

Ensuite, définissons l'interface d'une lumière qui peut s'éteindre automatiquement après un délai d'expiration :

C#

```
public interface ITimerLight : ILight
{
    Task TurnOnFor(int duration);
}
```

Vous pouvez ajouter une implémentation de base au luminaire, mais une meilleure solution consiste à modifier cette définition d'interface pour fournir une implémentation `virtual` par défaut :

C#

```
public interface ITimerLight : ILight
{
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Using the default interface method for the
ITimerLight.TurnOnFor.");
        SwitchOn();
        await Task.Delay(duration);
        SwitchOff();
        Console.WriteLine("Completed ITimerLight.TurnOnFor sequence.");
    }
}
```

La classe `OverheadLight` peut implémenter la fonction de minuterie en déclarant la prise en charge de l'interface :

C#

```
public class OverheadLight : ITimerLight { }
```

Un autre type de lumière peut prendre en charge un protocole plus sophistiqué. Il peut fournir sa propre implémentation pour `TurnOnFor`, comme illustré dans le code suivant :

C#

```
public class HalogenLight : ITimerLight
{
    private enum HalogenLightState
    {
        Off,
        On,
        TimerModeOn
    }

    private HalogenLightState state;
    public void SwitchOn() => state = HalogenLightState.On;
    public void SwitchOff() => state = HalogenLightState.Off;
    public bool IsOn() => state != HalogenLightState.Off;
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Halogen light starting timer function.");
        state = HalogenLightState.TimerModeOn;
        await Task.Delay(duration);
        state = HalogenLightState.Off;
        Console.WriteLine("Halogen light finished custom timer function");
    }
```

```
    public override string ToString() => $"The light is {state}";
}
```

Contrairement au remplacement de méthodes de classe virtuelle, la déclaration de `TurnOnFor` dans la classe `HalogenLight` n'utilise pas le mot clé `override`.

Combiner les fonctionnalités

Les avantages des méthodes d'interface par défaut deviennent plus clairs à mesure que vous introduisez des fonctionnalités plus avancées. L'utilisation d'interfaces vous permet de combiner des fonctionnalités. Cela permet également à chaque auteur de classe de choisir entre l'implémentation par défaut et une implémentation personnalisée. Ajoutons une interface avec une implémentation par défaut pour un voyant clignotant :

C#

```
public interface IBlinkingLight : ILight
{
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("Using the default interface method for
IBlinkingLight.Blink.");
        for (int count = 0; count < repeatCount; count++)
        {
            SwitchOn();
            await Task.Delay(duration);
            SwitchOff();
            await Task.Delay(duration);
        }
        Console.WriteLine("Done with the default interface method for
IBlinkingLight.Blink.");
    }
}
```

L'implémentation par défaut permet à n'importe quel voyant de clignoter. Le luminaire peut ajouter des fonctionnalités de minuterie et de clignotement à l'aide de l'implémentation par défaut :

C#

```
public class OverheadLight : ILight, ITimerLight, IBlinkingLight
{
    private bool isOn;
    public bool IsOn() => isOn;
    public void SwitchOff() => isOn = false;
    public void SwitchOn() => isOn = true;
```

```
    public override string ToString() => $"The light is {(isOn ? "on" :
"off")}";
}
```

Un nouveau type de lumière, le `LEDLight`, prend en charge directement la fonction de minuterie et la fonction de clignotement. Ce style clair implémente les interfaces `ITimerLight` et `IBlinkingLight` et remplace la méthode `Blink` :

C#

```
public class LEDLight : IBlinkingLight, ITimerLight, ILight
{
    private bool isOn;
    public void SwitchOn() => isOn = true;
    public void SwitchOff() => isOn = false;
    public bool IsOn() => isOn;
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("LED Light starting the Blink function.");
        await Task.Delay(duration * repeatCount);
        Console.WriteLine("LED Light has finished the Blink function.");
    }

    public override string ToString() => $"The light is {(isOn ? "on" :
"off")}";
}
```

Un `ExtraFancyLight` peut prendre en charge les fonctions de clignotement et de minuterie directement :

C#

```
public class ExtraFancyLight : IBlinkingLight, ITimerLight, ILight
{
    private bool isOn;
    public void SwitchOn() => isOn = true;
    public void SwitchOff() => isOn = false;
    public bool IsOn() => isOn;
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("Extra Fancy Light starting the Blink function.");
        await Task.Delay(duration * repeatCount);
        Console.WriteLine("Extra Fancy Light has finished the Blink
function.");
    }
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Extra Fancy light starting timer function.");
        await Task.Delay(duration);
        Console.WriteLine("Extra Fancy light finished custom timer
```

```
function");
    }

    public override string ToString() => $"The light is {(isOn ? "on" :
"off")}";
}
```

Le `HalogenLight` que vous avez créé précédemment ne prend pas en charge le clignotement. Par conséquent, n'ajoutez pas le `IBlinkingLight` à la liste de ses interfaces prises en charge.

Détecter les types de lumière à l'aide de la correspondance de modèle

Ensuite, écrivons du code de test. Vous pouvez utiliser la fonctionnalité de critères spéciaux de C# pour déterminer les fonctionnalités d'une lumière en examinant les interfaces qu'elle prend en charge. La méthode suivante exerce les fonctionnalités prises en charge de chaque lumière :

C#

```
private static async Task TestLightCapabilities(ILight light)
{
    // Perform basic tests:
    light.SwitchOn();
    Console.WriteLine($"\tAfter switching on, the light is {(light.IsOn() ?
"on" : "off")}");
    light.SwitchOff();
    Console.WriteLine($"\tAfter switching off, the light is {(light.IsOn() ?
"on" : "off")}");

    if (light is ITimerLight timer)
    {
        Console.WriteLine("\tTesting timer function");
        await timer.TurnOnFor(1000);
        Console.WriteLine("\tTimer function completed");
    }
    else
    {
        Console.WriteLine("\tTimer function not supported.");
    }

    if (light is IBlinkingLight blinker)
    {
        Console.WriteLine("\tTesting blinking function");
        await blinker.Blink(500, 5);
        Console.WriteLine("\tBlink function completed");
    }
```

```
    else
    {
        Console.WriteLine("\tBlink function not supported.");
    }
}
```

Le code suivant dans votre méthode `Main` crée chaque type de lumière dans la séquence et teste cette lumière :

C#

```
static async Task Main(string[] args)
{
    Console.WriteLine("Testing the overhead light");
    var overhead = new OverheadLight();
    await TestLightCapabilities(overhead);
    Console.WriteLine();

    Console.WriteLine("Testing the halogen light");
    var halogen = new HalogenLight();
    await TestLightCapabilities(halogen);
    Console.WriteLine();

    Console.WriteLine("Testing the LED light");
    var led = new LEDLight();
    await TestLightCapabilities(led);
    Console.WriteLine();

    Console.WriteLine("Testing the fancy light");
    var fancy = new ExtraFancyLight();
    await TestLightCapabilities(fancy);
    Console.WriteLine();
}
```

Comment le compilateur détermine la meilleure implémentation

Ce scénario montre une interface de base sans implémentations. L'ajout d'une méthode à l'interface `ILight` introduit de nouvelles complexités. Les règles de langage qui régissent les méthodes d'interface par défaut réduisent l'effet sur les classes concrètes qui implémentent plusieurs interfaces dérivées. Nous allons améliorer l'interface d'origine avec une nouvelle méthode pour montrer comment cela change son utilisation. Chaque voyant lumineux peut signaler son état d'alimentation sous la forme d'une valeur énumérée :

C#

```
public enum PowerStatus
{
    NoPower,
    ACPower,
    FullBattery,
    MidBattery,
    LowBattery
}
```

L'implémentation par défaut suppose l'absence d'alimentation :

C#

```
public interface ILight
{
    void SwitchOn();
    void SwitchOff();
    bool IsOn();
    public PowerStatus Power() => PowerStatus.NoPower;
}
```

Ces modifications compilent correctement, même si le `ExtraFancyLight` déclare la prise en charge de l'interface `ILight` et des interfaces dérivées, `ITimerLight` et `IBlinkingLight`. Il n'y a qu'une seule implémentation, « la plus proche », déclarée dans l'interface `ILight`. Toute classe qui a déclaré une substitution devient l'implémentation « la plus proche ». Vous avez vu des exemples dans les classes précédentes qui remplacent les membres d'autres interfaces dérivées.

Évitez de remplacer la même méthode dans plusieurs interfaces dérivées. Cela crée un appel de méthode ambigu chaque fois qu'une classe implémente les deux interfaces dérivées. Le compilateur ne peut pas choisir une seule meilleure méthode ; il génère donc une erreur. Par exemple, si la `IBlinkingLight` et `ITimerLight` implémentent un remplacement de `PowerStatus`, `OverheadLight` doit fournir un remplacement plus spécifique. Sinon, le compilateur ne peut pas choisir entre les implémentations des deux interfaces dérivées. Vous pouvez généralement éviter cette situation en gardant les définitions d'interface petites et axées sur une fonctionnalité. Dans ce scénario, chaque fonctionnalité d'une lumière est sa propre interface, seulement les classes héritent de plusieurs interfaces.

Cet exemple montre un scénario dans lequel vous pouvez définir des fonctionnalités discrètes pouvant être combinées dans des classes. Vous déclarez n'importe quel ensemble de fonctionnalités prises en charge en déclarant les interfaces prises en charge par une classe. L'utilisation de méthodes d'interface virtuelle par défaut permet aux classes d'utiliser ou de définir une implémentation différente pour tout nombre des

méthodes d’interface. Cette fonctionnalité de langage offre de nouvelles façons de modéliser les systèmes réels que vous créez. Les méthodes d’interface par défaut fournissent un moyen plus clair d’exprimer des classes associées qui peuvent combiner différentes fonctionnalités à l’aide d’implémentations virtuelles de ces fonctionnalités.

Index et plages

Article • 09/05/2023

Les plages et les index fournissent une syntaxe concise pour accéder à des éléments uniques ou des plages en séquence.

Ce didacticiel vous montre comment effectuer les opérations suivantes :

- ✓ Utiliser la syntaxe pour les plages dans une séquence.
- ✓ Définissez implicitement un Range.
- ✓ Comprendre les décisions de conception pour le début et la fin de chaque séquence.
- ✓ Découvrir des scénarios pour les types Index et Range.

Prise en charge linguistique pour les index et les plages

Les index et les plages fournissent une syntaxe succincte pour accéder à des éléments ou des plages uniques en séquence.

Cette prise en charge linguistique s'appuie sur deux nouveaux types et deux nouveaux opérateurs :

- System.Index représente un index au sein d'une séquence.
- L'index de l'opérateur de fin `^`, qui spécifie qu'un index est relatif à la fin d'une séquence.
- System.Range représente une sous-plage d'une séquence.
- L'opérateur de plage `..`, qui spécifie le début et la fin d'une plage comme ses opérandes.

Commençons par les règles concernant les indices. Prenons pour exemple un tableau `sequence`. L'index `0` est identique à l'index `sequence[0]`. L'index `^0` est identique à l'index `sequence[sequence.Length]`. L'expression `sequence[^0]` lève une exception, tout comme `sequence[sequence.Length]` le fait. Pour n'importe quel nombre `n`, l'index `^n` est identique à l'index `sequence.Length - n`.

C#

```
string[] words = new string[]
{
                // index from start    index from end
    "The",      // 0                   ^9
```

```
    "quick",    // 1                    ^8
    "brown",    // 2                    ^7
    "fox",      // 3                    ^6
    "jumps",    // 4                    ^5
    "over",     // 5                    ^4
    "the",      // 6                    ^3
    "lazy",     // 7                    ^2
    "dog"       // 8                    ^1
};              // 9 (or words.Length) ^0
```

Vous pouvez récupérer le dernier mot avec l'index `^1`. Ajoutez le code suivant sous l'initialisation :

C#

```
Console.WriteLine($"The last word is {words[^1]}");
```

Une plage spécifie son *début* et sa *fin*. Le début de la plage est inclus, mais la fin de la plage est exclusive, ce qui signifie que le *début* est inclus dans la plage mais pas la *fin*. La plage `[0..^0]` représente la plage dans son intégralité, tout comme `[0..sequence.Length]` représente la plage entière.

Le code suivant crée une sous-plage qui comporte les mots « quick », « brown » et « fox » et va de `words[1]` à `words[3]`. L'élément `words[4]` n'est pas dans la plage.

C#

```
string[] quickBrownFox = words[1..4];
foreach (var word in quickBrownFox)
    Console.Write($"< {word} >");
Console.WriteLine();
```

Le code suivant retourne la plage avec « lazy » et « dog ». et comprend `words[^2]` et `words[^1]`. L'index de fin `words[^0]` n'est pas inclus. Ajoutez également le code suivant :

C#

```
string[] lazyDog = words[^2..^0];
foreach (var word in lazyDog)
    Console.Write($"< {word} >");
Console.WriteLine();
```

Les exemples suivants créent des plages ouvertes au début, à la fin ou les deux :

C#

```
string[] allWords = words[..]; // contains "The" through "dog".
string[] firstPhrase = words[..4]; // contains "The" through "fox"
string[] lastPhrase = words[6..]; // contains "the", "lazy" and "dog"
foreach (var word in allWords)
    Console.Write($"< {word} >");
Console.WriteLine();
foreach (var word in firstPhrase)
    Console.Write($"< {word} >");
Console.WriteLine();
foreach (var word in lastPhrase)
    Console.Write($"< {word} >");
Console.WriteLine();
```

Vous pouvez également déclarer des plages ou index comme variables. La variable peut ensuite être utilisée à l'intérieur des caractères `[` et `]` :

C#

```
Index the = ^3;
Console.WriteLine(words[the]);
Range phrase = 1..4;
string[] text = words[phrase];
foreach (var word in text)
    Console.Write($"< {word} >");
Console.WriteLine();
```

L'exemple suivant montre un grand nombre des raisons de ces choix. Modifiez `x`, `y` et `z` pour essayer différentes combinaisons. Quand vous effectuez des essais, utilisez des valeurs de telle sorte que `x` soit inférieur à `y` et `y` inférieur à `z` pour avoir des combinaisons valides. Ajoutez le code suivant à une nouvelle méthode. Essayez différentes combinaisons :

C#

```
int[] numbers = Enumerable.Range(0, 100).ToArray();
int x = 12;
int y = 25;
int z = 36;

Console.WriteLine($"{numbers[^x]} is the same as {numbers[numbers.Length -
x]}");
Console.WriteLine($"{numbers[x..y].Length} is the same as {y - x}");

Console.WriteLine("numbers[x..y] and numbers[y..z] are consecutive and
disjoint:");
Span<int> x_y = numbers[x..y];
Span<int> y_z = numbers[y..z];
Console.WriteLine($"\tnumbers[x..y] is {x_y[0]} through {x_y[^1]},
numbers[y..z] is {y_z[0]} through {y_z[^1]}");
```

```
Console.WriteLine("numbers[x..^x] removes x elements at each end:");
Span<int> x_x = numbers[x..^x];
Console.WriteLine($"\tnumbers[x..^x] starts with {x_x[0]} and ends with
{x_x[^1]}");

Console.WriteLine("numbers[..x] means numbers[0..x] and numbers[x..] means
numbers[x..^0]");
Span<int> start_x = numbers[..x];
Span<int> zero_x = numbers[0..x];
Console.WriteLine($"\t{start_x[0]}..{start_x[^1]} is the same as
{zero_x[0]}..{zero_x[^1]}");
Span<int> z_end = numbers[z..];
Span<int> z_zero = numbers[z..^0];
Console.WriteLine($"\t{z_end[0]}..{z_end[^1]} is the same as {z_zero[0]}..
{z_zero[^1]}");
```

Les tableaux ne sont pas les seuls à prendre en charge les index et les plages. Vous pouvez également utiliser des index et des plages avec string, Span<T> ou ReadOnlySpan<T>.

Conversions d'expression d'opérateur de plage implicite

Lors de l'utilisation de la syntaxe d'expression d'opérateur de plage, le compilateur convertit implicitement les valeurs de début et de fin en Index et, à partir de celles-ci, crée une instance Range. Le code suivant montre un exemple de conversion implicite à partir de la syntaxe d'expression d'opérateur de plage et son alternative explicite correspondante :

C#

```
Range implicitRange = 3..^5;

Range explicitRange = new(
    start: new Index(value: 3, fromEnd: false),
    end: new Index(value: 5, fromEnd: true));

if (implicitRange.Equals(explicitRange))
{
    Console.WriteLine(
        $"The implicit range '{implicitRange}' equals the explicit range
'{explicitRange}'");
}
// Sample output:
//     The implicit range '3..^5' equals the explicit range '3..^5'
```

ⓘ Important

Les conversions implicites de **Int32** à **Index** lèvent **ArgumentOutOfRangeException** lorsque la valeur est négative. De même, le constructeur `Index` lève `ArgumentOutOfRangeException` lorsque le paramètre `value` est négatif.

Prise en charge des types d'index et de plages

Les index et les plages fournissent une syntaxe claire et concise pour accéder à un élément unique ou à une plage d'éléments en séquence. Une expression d'index retourne généralement le type des éléments d'une séquence. Une expression de plage retourne généralement le même type de séquence que la séquence source.

Tout type qui fournit un indexeur avec un paramètre Index ou Range prend explicitement en charge les index ou plages, respectivement. Un indexeur qui prend un paramètre unique Range peut retourner un type de séquence différent, comme System.Span<T>.

> ⓘ **Important**
>
> Les performances du code utilisant l'opérateur de plage dépendent du type de l'opérande de séquence.
>
> La complexité temporelle de l'opérateur de plage dépend du type de séquence. Par exemple, si la séquence est un `string` ou un tableau, le résultat est une copie de la section spécifiée de l'entrée, de sorte que la complexité temporelle est *O(N)* (où N est la longueur de la plage). En revanche, s'il s'agit d'un **System.Span<T>** ou d'un **System.Memory<T>**, le résultat fait référence au même magasin de stockage, ce qui signifie qu'il n'y a pas de copie et que l'opération est *O(1)*.
>
> En plus de la complexité temporelle, cela entraîne des allocations et des copies supplémentaires, ce qui a un impact sur les performances. Dans le code sensible aux performances, envisagez d'utiliser `Span<T>` ou `Memory<T>` comme type de séquence, car l'opérateur de plage ne les alloue pas.

Un type est **comptable** s'il a une propriété nommée `Length` ou `Count` avec un getter accessible et un type de retour de `int`. Un type comptable qui ne prend pas explicitement en charge les index ou les plages peut fournir une prise en charge implicite pour ceux-ci. Pour plus d'informations, consultez les sections Prise en charge des index implicites et Prise en charge des plages implicites de la Note de proposition de fonctionnalité. Les plages utilisant la prise en charge des plages implicites retournent le même type de séquence que la séquence source.

Par exemple, les types .NET suivants prennent en charge les index et les plages String, Span<T> et ReadOnlySpan<T>. Le List<T> prend en charge les index, mais pas les plages.

Array a un comportement plus nuancé. Les tableaux de dimension unique prennent en charge les index et les plages. Les tableaux multidimensionnels ne prennent pas en charge les indexeurs ou les plages. L'indexeur d'un tableau multidimensionnel a plusieurs paramètres, pas un seul paramètre. Les tableaux en escalier, également appelés tableaux de tableaux, prennent en charge les plages et les indexeurs. L'exemple suivant montre comment itérer une sous-section rectangulaire d'un tableau en escalier. Il itère la section au centre, à l'exclusion des trois premières et dernières lignes, ainsi que des deux premières et dernières colonnes de chaque ligne sélectionnée :

C#

```
var jagged = new int[10][]
{
   new int[10] {  0, 1, 2, 3, 4, 5, 6, 7, 8, 9 },
   new int[10] { 10,11,12,13,14,15,16,17,18,19 },
   new int[10] { 20,21,22,23,24,25,26,27,28,29 },
   new int[10] { 30,31,32,33,34,35,36,37,38,39 },
   new int[10] { 40,41,42,43,44,45,46,47,48,49 },
   new int[10] { 50,51,52,53,54,55,56,57,58,59 },
   new int[10] { 60,61,62,63,64,65,66,67,68,69 },
   new int[10] { 70,71,72,73,74,75,76,77,78,79 },
   new int[10] { 80,81,82,83,84,85,86,87,88,89 },
   new int[10] { 90,91,92,93,94,95,96,97,98,99 },
};

var selectedRows = jagged[3..^3];

foreach (var row in selectedRows)
{
    var selectedColumns = row[2..^2];
    foreach (var cell in selectedColumns)
    {
        Console.Write($"{cell}, ");
    }
    Console.WriteLine();
}
```

Dans tous les cas, l'opérateur de plage pour Array alloue un tableau pour stocker les éléments retournés.

Scénarios pour les index et les plages

Vous utiliserez souvent des plages et des index pour analyser une partie d'une séquence plus grande. La nouvelle syntaxe est plus claire pour la lecture exacte de la partie de la séquence impliquée. La fonction locale `MovingAverage` prend un Range comme argument. La méthode énumère ensuite simplement cette plage lors du calcul des valeurs minimale, maximale et moyenne. Essayez le code suivant dans votre projet :

C#

```
int[] sequence = Sequence(1000);

for(int start = 0; start < sequence.Length; start += 100)
{
    Range r = start..(start+10);
    var (min, max, average) = MovingAverage(sequence, r);
    Console.WriteLine($"From {r.Start} to {r.End}:    \tMin: {min},\tMax:
{max},\tAverage: {average}");
}

for (int start = 0; start < sequence.Length; start += 100)
{
    Range r = ^(start + 10)..^start;
    var (min, max, average) = MovingAverage(sequence, r);
    Console.WriteLine($"From {r.Start} to {r.End}:  \tMin: {min},\tMax:
{max},\tAverage: {average}");
}

(int min, int max, double average) MovingAverage(int[] subSequence, Range
range) =>
    (
        subSequence[range].Min(),
        subSequence[range].Max(),
        subSequence[range].Average()
    );

int[] Sequence(int count) =>
    Enumerable.Range(0, count).Select(x => (int)(Math.Sqrt(x) *
100)).ToArray();
```

Remarque sur les indices et les tableaux de plage

Lorsque vous prenez une plage à partir d'un tableau, le résultat est un tableau qui est copié à partir du tableau initial, plutôt que référencé. La modification des valeurs dans le tableau résultant ne modifie pas les valeurs du tableau initial.

Par exemple :

C#

```
var arrayOfFiveItems = new[] { 1, 2, 3, 4, 5 };

var firstThreeItems = arrayOfFiveItems[..3]; // contains 1,2,3
firstThreeItems[0] =  11; // now contains 11,2,3

Console.WriteLine(string.Join(",", firstThreeItems));
Console.WriteLine(string.Join(",", arrayOfFiveItems));

// output:
// 11,2,3
// 1,2,3,4,5
```

Tutoriel : Exprimer plus clairement une intention de conception avec les types référence Nullable et non Nullable

Article • 10/05/2023

Les types référence nullable, qui viennent compléter les types référence de la même façon que les types valeur nullable complètent les types valeur. Pour déclarer une variable comme étant un **type référence Nullable**, on ajoute `?` au type. Par exemple, `string?` représente une `string` Nullable. Vous pouvez utiliser ces nouveaux types pour exprimer plus clairement votre intention de conception : certaines variables *doivent toujours avoir une valeur*, d'autres *peuvent ne pas en avoir*.

Ce didacticiel vous montre comment effectuer les opérations suivantes :

- ✓ incorporer des types référence Nullable et non Nullable dans vos conceptions ;
- ✓ activer les contrôles de type référence Nullable dans l'ensemble de votre code ;
- ✓ écrire du code permettant au compilateur d'appliquer ces décisions de conception ;
- ✓ utiliser la fonctionnalité de référence Nullable dans vos propres conceptions.

Prérequis

Vous devrez configurer votre ordinateur de façon à exécuter .NET, avec le compilateur C#. Le compilateur C# est accessible via Visual Studio 2022 ou le SDK .NET.

Ce tutoriel suppose de connaître C# et .NET, y compris Visual Studio ou l'interface CLI .NET.

Incorporer des types référence Nullable dans des conceptions

Dans ce tutoriel, vous allez générer une bibliothèque qui modélise la réalisation d'une enquête. Le code utilise des types référence Nullable et non Nullable pour représenter les concepts du monde réel. Les questions de l'enquête ne peuvent jamais être Null. Si la personne interrogée ne souhaite pas répondre à une question, la réponse peut être `null`.

Le code que vous allez écrire pour cet exemple exprime cette intention, que le compilateur se charge d'appliquer.

Créer l'application et activer les types référence Nullable

Créez une application console dans Visual Studio ou en ligne de commande avec `dotnet new console`. Nommez l'application `NullableIntroduction`. Une fois que vous avez créé l'application, vous devez préciser que l'ensemble du projet est compilé dans un **contexte d'annotation nullable** activé. Ouvrez le fichier *.csproj* et ajoutez un élément `Nullable` à l'élément `PropertyGroup`. Affectez-lui la valeur `enable`. Vous devez activer la fonctionnalité **types référence nullable** dans les projets antérieurs à C# 11. En effet, une fois la fonctionnalité activée, les déclarations de variables référence existantes deviennent des **types référence non Nullable**. Bien que cette décision aide à identifier les problèmes de contrôle de type Null dans le code, elle ne reflète pas forcément l'intention de conception d'origine :

```XML
<Nullable>enable</Nullable>
```

Avant .NET 6, les nouveaux projets n'incluent pas l'élément `Nullable`. À partir de .NET 6, les nouveaux projets incluent l'élément `<Nullable>enable</Nullable>` dans le fichier projet.

Concevoir les types de l'application

Cette application d'enquête implique la création d'un certain nombre de classes :

- une classe qui modélise la liste des questions ;
- une classe qui modélise la liste des personnes contactées pour l'enquête ;
- une classe qui modélise les réponses d'une personne ayant répondu à l'enquête.

Ces types utilisent des types référence Nullable et non Nullable pour indiquer quels membres sont requis et lesquels sont facultatifs. Les types référence Nullable communiquent clairement cette intention de conception :

- Les questions qui font partie de l'enquête ne peuvent jamais être Null : poser une question vide n'aurait pas de sens.
- Les personnes interrogées ne peuvent jamais être Null. Vous souhaitez faire un suivi des personnes que vous avez contactées, même de celles qui ont refusé de participer.
- La réponse à une question peut être Null. Les personnes interrogées peuvent refuser de répondre à certaines questions ou à la totalité d'entre elles.

Si vous avez déjà programmé en C#, vous avez peut-être une telle habitude des types référence qui autorisent les valeurs `null` que vous avez raté certaines occasions de déclarer des instances non nullable :

- La collection de questions doit être non Nullable.
- La collection de personnes interrogées doit être non Nullable.

En écrivant le code, vous allez voir qu'utiliser par défaut un type référence non Nullable pour les références permet d'éviter des erreurs courantes susceptibles d'aboutir à des exceptions NullReferenceException. Ce tutoriel vous aide à prendre des décisions sur les variables qui peuvent ou non être `null`. Avant, le langage ne fournissait pas de syntaxe permettant d'exprimer ces décisions ; maintenant, si.

Voici ce que l'application que vous allez créer va faire :

1. Créer une enquête et y ajouter des questions.
2. Créer un ensemble pseudo-aléatoire de répondants pour l'enquête.
3. Contacter les répondants jusqu'à ce que l'enquête atteigne la taille visée.
4. Écrire des statistiques importantes sur les réponses à l'enquête.

Créer l'enquête avec des types référence nullable et non nullable

La première partie du code crée l'enquête. Vous allez écrire des classes pour modéliser une question et une enquête. L'enquête comporte trois types de questions, en fonction du format de la réponse : réponses par oui ou non, réponses chiffrées et réponses textuelles. Créez une classe `public SurveyQuestion` :

C#

```
namespace NullableIntroduction
{
    public class SurveyQuestion
    {
    }
}
```

Le compilateur interprète chaque déclaration de variable de type référence comme **non nullable** pour le code dans un contexte d'annotation nullable activé. Un premier avertissement apparaît lorsque l'on ajoute des propriétés pour le texte et le type de la question avec le code suivant :

C#

```
namespace NullableIntroduction
{
    public enum QuestionType
    {
        YesNo,
        Number,
        Text
    }

    public class SurveyQuestion
    {
        public string QuestionText { get; }
        public QuestionType TypeOfQuestion { get; }
    }
}
```

Comme `QuestionText` n'est pas initialisé, le compilateur émet l'avertissement selon lequel une propriété non Nullable n'a pas été initialisée. Or, la conception exige que le texte de la question soit non Null. Ajoutez un constructeur pour l'initialiser, ainsi que la valeur `QuestionType`. Une fois terminée, la définition de classe se présente ainsi :

C#

```
namespace NullableIntroduction;

public enum QuestionType
{
    YesNo,
    Number,
    Text
}

public class SurveyQuestion
{
    public string QuestionText { get; }
    public QuestionType TypeOfQuestion { get; }

    public SurveyQuestion(QuestionType typeOfQuestion, string text) =>
        (TypeOfQuestion, QuestionText) = (typeOfQuestion, text);
}
```

Le fait d'ajouter le constructeur supprime l'avertissement. L'argument du constructeur étant également un type référence non Nullable, le compilateur ne génère pas d'avertissements.

Ensuite, créez une classe `public` nommée `SurveyRun`. Cette classe contient une liste d'objets `SurveyQuestion` et de méthodes permettant d'ajouter des questions à l'enquête, comme l'illustre le code suivant :

C#

```
using System.Collections.Generic;

namespace NullableIntroduction
{
    public class SurveyRun
    {
        private List<SurveyQuestion> surveyQuestions = new
List<SurveyQuestion>();

        public void AddQuestion(QuestionType type, string question) =>
            AddQuestion(new SurveyQuestion(type, question));
        public void AddQuestion(SurveyQuestion surveyQuestion) =>
surveyQuestions.Add(surveyQuestion);
    }
}
```

Comme tout à l'heure, il faut initialiser l'objet de liste sur une valeur non Null pour éviter que le compilateur n'émette un avertissement. Il n'y a pas de contrôle de type Null dans la deuxième surcharge de `AddQuestion`, car ce n'est pas nécessaire : vous avez déclaré cette variable comme étant non Nullable. Sa valeur ne peut pas être `null`.

Passez à *Program.cs* `Main` *dans votre éditeur et remplacez le contenu de* par les lignes de code suivantes :

C#

```
var surveyRun = new SurveyRun();
surveyRun.AddQuestion(QuestionType.YesNo, "Has your code ever thrown a
NullReferenceException?");
surveyRun.AddQuestion(new SurveyQuestion(QuestionType.Number, "How many
times (to the nearest 100) has that happened?"));
surveyRun.AddQuestion(QuestionType.Text, "What is your favorite color?");
```

Étant donné que l'ensemble du projet se trouve dans un contexte d'annotation nullable activé, vous obtenez des avertissements au moment où vous transmettez `null` à une méthode qui attend un type référence non nullable. Regardez le résultat en ajoutant la ligne suivante à `Main` :

C#

```
surveyRun.AddQuestion(QuestionType.Text, default);
```

Créer des personnes interrogées et obtenir des réponses à l'enquête

Ensuite, écrivez le code qui génère des réponses à l'enquête. Ce processus implique plusieurs petites tâches :

1. Créer une méthode qui génère des objets personne interrogée, représentant les personnes à qui il a été demandé de répondre à l'enquête.
2. Créer une logique pour simuler les questions posées à une personne interrogée et les réponses obtenues, ou noter qu'une personne interrogée n'a pas répondu.
3. Répéter le processus jusqu'à ce que les personnes interrogées soient en nombre suffisant.

Il vous faut une classe qui représente les réponse à l'enquête. Ajoutez-la maintenant. Activez la prise en charge Nullable. Ajoutez une propriété `Id` et un constructeur qui l'initialise, comme dans le code suivant :

C#

```
namespace NullableIntroduction
{
    public class SurveyResponse
    {
        public int Id { get; }

        public SurveyResponse(int id) => Id = id;
    }
}
```

Ensuite, ajoutez une méthode `static` pour créer de nouveaux participants en générant un ID aléatoire :

C#

```
private static readonly Random randomGenerator = new Random();
public static SurveyResponse GetRandomId() => new
SurveyResponse(randomGenerator.Next());
```

La responsabilité principale de cette classe consiste à générer les réponses d'un participant aux questions de l'enquête, ce qui se décompose en plusieurs étapes :

1. Demander à participer à l'enquête. Si la personne refuse, renvoyer une réponse manquante (Null).

2. Poser chaque question et enregistrer la réponse. Chacune des réponses peut également être manquante (Null).

Ajoutez le code suivant à la classe `SurveyResponse` :

C#

```
private Dictionary<int, string>? surveyResponses;
public bool AnswerSurvey(IEnumerable<SurveyQuestion> questions)
{
    if (ConsentToSurvey())
    {
        surveyResponses = new Dictionary<int, string>();
        int index = 0;
        foreach (var question in questions)
        {
            var answer = GenerateAnswer(question);
            if (answer != null)
            {
                surveyResponses.Add(index, answer);
            }
            index++;
        }
    }
    return surveyResponses != null;
}

private bool ConsentToSurvey() => randomGenerator.Next(0, 2) == 1;

private string? GenerateAnswer(SurveyQuestion question)
{
    switch (question.TypeOfQuestion)
    {
        case QuestionType.YesNo:
            int n = randomGenerator.Next(-1, 2);
            return (n == -1) ? default : (n == 0) ? "No" : "Yes";
        case QuestionType.Number:
            n = randomGenerator.Next(-30, 101);
            return (n < 0) ? default : n.ToString();
        case QuestionType.Text:
        default:
            switch (randomGenerator.Next(0, 5))
            {
                case 0:
                    return default;
                case 1:
                    return "Red";
                case 2:
                    return "Green";
                case 3:
                    return "Blue";
            }
            return "Red. No, Green. Wait.. Blue... AAARGGGGGHHH!";
```

```
    }
}
```

Les réponses à l'enquête sont stockées dans un `Dictionary<int, string>?`, qui peut donc être Null. Vous utilisez la nouvelle fonctionnalité du langage pour déclarer votre intention de conception, à la fois au compilateur et à toute personne qui lira votre code. Si vous déréférencez `surveyResponses` sans d'abord contrôler la valeur `null`, le compilateur émet un avertissement. La méthode `AnswerSurvey` ne génère pas d'avertissement, car le compilateur peut déterminer que la variable `surveyResponses` a été définie avant sur une valeur non Null.

L'utilisation de `null` pour les réponses manquantes met en évidence un point important pour travailler avec les types référence nullable : votre objectif n'est pas de supprimer toutes les valeurs `null` à partir de votre programme. Au lieu de cela, votre objectif est de vous assurer que le code que vous écrivez exprime l'intention de votre conception. Les valeurs manquantes sont un concept nécessaire à exprimer dans votre code. La valeur `null` est une méthode évidente pour exprimer les valeurs manquantes. Essayer de supprimer toutes les valeurs `null` mène uniquement à la définition d'une autre façon d'exprimer ces valeurs manquantes sans `null`.

Ensuite, il reste à écrire la méthode `PerformSurvey` dans la classe `SurveyRun`. Ajoutez le code suivant à la classe `SurveyRun` :

C#

```
private List<SurveyResponse>? respondents;
public void PerformSurvey(int numberOfRespondents)
{
    int respondentsConsenting = 0;
    respondents = new List<SurveyResponse>();
    while (respondentsConsenting < numberOfRespondents)
    {
        var respondent = SurveyResponse.GetRandomId();
        if (respondent.AnswerSurvey(surveyQuestions))
            respondentsConsenting++;
        respondents.Add(respondent);
    }
}
```

Là encore, le choix d'une `List<SurveyResponse>?` Nullable indique que la réponse peut être Null, et que l'enquête n'a pour le moment été menée auprès de personne. Des personnes interrogées sont ajoutées jusqu'à ce qu'elles soient suffisamment nombreuses à avoir accepté.

La dernière étape consiste à ajouter un appel pour effectuer l'enquête à la fin de la méthode `Main` :

```
C#

surveyRun.PerformSurvey(50);
```

Examiner les réponses à l'enquête

La dernière étape consiste à afficher les résultats de l'enquête, ce qui suppose d'ajouter du code à la plupart des classes déjà écrites. Ce code montre l'intérêt de distinguer les types référence Nullable et non Nullable. Commencez par ajouter les deux membres expression-bodied suivants à la classe `SurveyResponse` :

```
C#

public bool AnsweredSurvey => surveyResponses != null;
public string Answer(int index) => surveyResponses?.GetValueOrDefault(index)
?? "No answer";
```

`surveyResponses` étant un type référence non nullable, son déréférencement nécessite au préalable une recherche de valeurs null. Comme la méthode `Answer` retourne une chaîne non nullable, nous devons prévoir le cas où une réponse est manquante en utilisant l'opérateur de coalescence nulle.

Ensuite, ajoutez ces trois membres expression-bodied à la classe `SurveyRun` :

```
C#

public IEnumerable<SurveyResponse> AllParticipants => (respondents ??
Enumerable.Empty<SurveyResponse>());
public ICollection<SurveyQuestion> Questions => surveyQuestions;
public SurveyQuestion GetQuestion(int index) => surveyQuestions[index];
```

Le membre `AllParticipants` doit prendre en compte le fait que la variable `respondents` peut être Null, mais pas la valeur de retour. Si vous modifiez cette expression en supprimant `??` et la séquence vide qui suit, le compilateur émet un avertissement, indiquant que la méthode peut retourner `null` et sa signature de retour un type non Nullable.

Enfin, ajoutez la boucle suivante en bas de la méthode `Main` :

```
C#
```

```
foreach (var participant in surveyRun.AllParticipants)
{
    Console.WriteLine($"Participant: {participant.Id}:");
    if (participant.AnsweredSurvey)
    {
        for (int i = 0; i < surveyRun.Questions.Count; i++)
        {
            var answer = participant.Answer(i);
            Console.WriteLine($"\t{surveyRun.GetQuestion(i).QuestionText} :
{answer}");
        }
    }
    else
    {
        Console.WriteLine("\tNo responses");
    }
}
```

Aucun contrôle `null` n'est nécessaire dans ce code, car vous avez conçu les interfaces sous-jacentes de sorte qu'elles retournent toutes des types référence non Nullable.

Obtenir le code

Pour obtenir le code du tutoriel complet, consultez notre dépôt samples dans le dossier csharp/NullableIntroduction.

Faites des essais en modifiant les déclarations de type entre les types référence Nullable et non Nullable. Examinez les différents avertissements générés, qui visent à empêcher de déréférencer accidentellement un `null`.

Étapes suivantes

Découvrez comment employer le type référence nullable quand Entity Framework est utilisé :

Principes de base d'Entity Framework Core : Utilisation de types référence nullable

Tutoriel : Générer et consommer des flux asynchrones en utilisant C# et .NET

Article • 28/03/2023

Les **flux asynchrones** modélisent une source de données de streaming. Souvent, les flux de données récupèrent ou génèrent des éléments de manière asynchrone. Ils fournissent un modèle de programmation naturel pour les sources de données de streaming asynchrones.

Ce didacticiel vous montre comment effectuer les opérations suivantes :

- ✓ créer une source de données qui génère une séquence d'éléments de données de façon asynchrone ;
- ✓ consommer cette source de données de façon asynchrone ;
- ✓ prendre en charge l'annulation et les contextes capturés pour les flux asynchrones ;
- ✓ reconnaître quand l'interface et la source de données nouvelles sont préférables aux séquences de données synchrones précédentes.

Prérequis

Vous devrez configurer votre ordinateur de façon à exécuter .NET, avec le compilateur C#. Le compilateur C# est accessible via Visual Studio 2022 ou le SDK .NET.

Vous devrez créer un jeton d'accès GitHub afin de pouvoir accéder au point de terminaison GitHub GraphQL. Sélectionnez les autorisations suivantes pour votre jeton d'accès GitHub :

- repo:status
- public_repo

Enregistrez le jeton d'accès à un endroit sûr afin de pouvoir l'utiliser pour accéder au point de terminaison de l'API GitHub.

> ⚠ **Avertissement**
>
> Sécurisez votre jeton d'accès personnel. Tous les logiciels disposant de votre jeton d'accès personnel peuvent effectuer des appels d'API GitHub à l'aide de vos droits d'accès.

Ce tutoriel suppose de connaître C# et .NET, y compris Visual Studio ou l'interface CLI .NET.

Exécutez l'application de démarrage

Vous pouvez vous procurer le code de l'application de démarrage utilisée dans ce tutoriel dans notre référentiel dotnet/docs situé dans le dossier asynchronous-programming/snippets.

L'application de démarrage est une application console qui utilise l'interface GitHub GraphQL pour récupérer des problèmes récents écrits dans le référentiel dotnet/docs. Commencez par examiner le code suivant pour la méthode `Main` de l'application de démarrage :

C#

```
static async Task Main(string[] args)
{
    //Follow these steps to create a GitHub Access Token
    // https://help.github.com/articles/creating-a-personal-access-token-
for-the-command-line/#creating-a-token
    //Select the following permissions for your GitHub Access Token:
    // - repo:status
    // - public_repo
    // Replace the 3rd parameter to the following code with your GitHub
access token.
    var key = GetEnvVariable("GitHubKey",
    "You must store your GitHub key in the 'GitHubKey' environment
variable",
    "");

    var client = new GitHubClient(new
Octokit.ProductHeaderValue("IssueQueryDemo"))
    {
        Credentials = new Octokit.Credentials(key)
    };

    var progressReporter = new progressStatus((num) =>
    {
        Console.WriteLine($"Received {num} issues in total");
    });
    CancellationTokenSource cancellationSource = new
CancellationTokenSource();

    try
    {
        var results = await RunPagedQueryAsync(client, PagedIssueQuery,
"docs",
            cancellationSource.Token, progressReporter);
        foreach(var issue in results)
```

```
            Console.WriteLine(issue);
    }
    catch (OperationCanceledException)
    {
        Console.WriteLine("Work has been cancelled");
    }
}
```

Vous pouvez soit définir une variable d'environnement `GitHubKey` sur votre jeton d'accès personnel, soit remplacer le dernier argument dans l'appel par `GetEnvVariable` avec votre jeton d'accès personnel. Ne placez pas votre code d'accès dans le code source si vous prévoyez de partager la source avec d'autres utilisateurs. Ne chargez jamais de codes d'accès dans un référentiel source partagé.

Après la création du client de GitHub, le code dans `Main` crée un objet de rapport de progression et un jeton d'annulation. Une fois que ces objets sont créés, `Main` appelle `RunPagedQueryAsync` pour récupérer les 250 problèmes créés les plus récents. Une fois cette tâche terminée, les résultats sont affichés.

Lorsque vous exécutez l'application de démarrage, vous pouvez faire quelques observations importantes concernant son fonctionnement. Vous voyez la progression signalée pour chaque page retournée à partir de GitHub. Vous pouvez observer un temps de pause avant le retour de chaque nouvelle page de problèmes par GitHub. Enfin, les problèmes ne sont affichés que lorsque les 10 pages ont été récupérées à partir de GitHub.

Examinez l'implémentation

L'implémentation révèle pourquoi vous avez observé le comportement décrit dans la section précédente. Examinez the code for `RunPagedQueryAsync` :

C#

```
private static async Task<JArray> RunPagedQueryAsync(GitHubClient client,
string queryText, string repoName, CancellationToken cancel, IProgress<int>
progress)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    JArray finalResults = new JArray();
    bool hasMorePages = true;
    int pagesReturned = 0;
```

```
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new
Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results =
JObject.Parse(response.HttpResponse.Body.ToString()!);

        int totalCount = (int)issues(results)["totalCount"]!;
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"]!;
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)
["startCursor"]!.ToString();
        issuesReturned += issues(results)["nodes"]!.Count();
        finalResults.Merge(issues(results)["nodes"]!);
        progress?.Report(issuesReturned);
        cancel.ThrowIfCancellationRequested();
    }
    return finalResults;

    JObject issues(JObject result) => (JObject)result["data"]!
["repository"]!["issues"]!;
    JObject pageInfo(JObject result) => (JObject)issues(result)
["pageInfo"]!;
}
```

Concentrons-nous sur l'algorithme de pagination et sur la structure asynchrone du code précédent. (Vous pouvez consulter la documentation GitHub GraphQL pour obtenir des détails sur l'API GitHub GraphQL.) La méthode `RunPagedQueryAsync` énumère les problèmes du plus récent au plus ancien. Elle a besoin de 25 problèmes par page et examine la structure `pageInfo` de la réponse pour continuer avec la page précédente. Cela suit la prise en charge standard de la pagination de GraphQL pour les réponses multipages. La réponse inclut un objet `pageInfo` qui contient une valeur `hasPreviousPages` et une valeur `startCursor` utilisées pour demander la page précédente. Les problèmes se trouvent dans le tableau `nodes`. La méthode `RunPagedQueryAsync` ajoute ces nœuds à un tableau qui contient tous les résultats de toutes les pages.

Après la récupération et la restauration d'une page de résultats, `RunPagedQueryAsync` signale la progression et vérifie l'annulation. Si l'annulation a été demandée, `RunPagedQueryAsync` lève une OperationCanceledException.

Plusieurs éléments de ce code peuvent être améliorés. Plus important encore, `RunPagedQueryAsync` doit allouer du stockage pour tous les problèmes retournés. Cet

exemple s'arrête à 250 problèmes, car la récupération de tous les problèmes ouverts nécessiterait beaucoup plus de mémoire pour stocker tous les problèmes récupérées. Les protocoles pour la prise en charge des rapports de progression et de l'annulation rendent l'algorithme plus difficile à comprendre lors de sa première lecture. D'autres types et API sont impliqués. Vous devez tracer les communications via le CancellationTokenSource et son CancellationToken associé pour comprendre où l'annulation est demandée et où elle est accordée.

Les flux asynchrones sont mieux adaptés

Les flux asynchrones et la prise en charge associée du langage résolvent tous ces problèmes. Le code qui génère la séquence peut désormais utiliser `yield return` pour retourner des éléments dans une méthode qui a été déclarée avec le modificateur `async`. Vous pouvez consommer un flux asynchrone à l'aide une boucle `await foreach` tout comme vous consommez n'importe quelle séquence à l'aide d'une boucle `foreach`.

Ces nouvelles fonctionnalités de langage dépendent de trois nouvelles interfaces ajoutées à .NET Standard 2.1 et implémentées dans .NET Core 3.0 :

- System.Collections.Generic.IAsyncEnumerable<T>
- System.Collections.Generic.IAsyncEnumerator<T>
- System.IAsyncDisposable

Ces trois interfaces sont très certainement familières à la plupart des développeurs C#. Elles se comportent de manière similaire à leurs équivalents synchrones :

- System.Collections.Generic.IEnumerable<T>
- System.Collections.Generic.IEnumerator<T>
- System.IDisposable

Il est possible que le type System.Threading.Tasks.ValueTask ne soit pas familier. Le struct `ValueTask` fournit une API similaire à la classe System.Threading.Tasks.Task. `ValueTask` est utilisé dans ces interfaces pour des raisons de performances.

Convertir en flux asynchrones

Ensuite, convertissez la méthode `RunPagedQueryAsync` pour générer un flux asynchrone. Tout d'abord, modifiez la signature de `RunPagedQueryAsync` pour retourner un `IAsyncEnumerable<JToken>` et supprimer les objets de jeton d'annulation et de progression de la liste de paramètres comme indiqué dans le code suivant :

C#

```
private static async IAsyncEnumerable<JToken>
RunPagedQueryAsync(GitHubClient client,
    string queryText, string repoName)
```

Le code de démarrage traite chaque page lorsqu'elle est récupérée, comme indiqué dans le code suivant :

C#

```
finalResults.Merge(issues(results)["nodes"]!);
progress?.Report(issuesReturned);
cancel.ThrowIfCancellationRequested();
```

Remplacez ces trois lignes par le code suivant :

C#

```
foreach (JObject issue in issues(results)["nodes"]!)
    yield return issue;
```

Vous pouvez également supprimer la déclaration de `finalResults` plus tôt dans cette méthode et l'instruction `return` qui suit la boucle que vous avez modifiée.

Vous avez terminé les modifications permettant de générer un flux asynchrone. Une fois terminée, la méthode doit ressembler au code suivant :

C#

```
private static async IAsyncEnumerable<JToken>
RunPagedQueryAsync(GitHubClient client,
    string queryText, string repoName)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
```

```
        var response = await client.Connection.Post<string>(new
Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results =
JObject.Parse(response.HttpResponse.Body.ToString()!);

        int totalCount = (int)issues(results)["totalCount"]!;
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"]!;
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)
["startCursor"]!.ToString();
        issuesReturned += issues(results)["nodes"]!.Count();

        foreach (JObject issue in issues(results)["nodes"]!)
            yield return issue;
    }

    JObject issues(JObject result) => (JObject)result["data"]!
["repository"]!["issues"]!;
    JObject pageInfo(JObject result) => (JObject)issues(result)
["pageInfo"]!;
}
```

Ensuite, vous modifiez le code qui utilise la collection pour consommer le flux de données asynchrone. Recherchez dans `Main` le code suivant, qui traite l'ensemble des problèmes :

C#

```
var progressReporter = new progressStatus((num) =>
{
    Console.WriteLine($"Received {num} issues in total");
});
CancellationTokenSource cancellationSource = new CancellationTokenSource();

try
{
    var results = await RunPagedQueryAsync(client, PagedIssueQuery, "docs",
        cancellationSource.Token, progressReporter);
    foreach(var issue in results)
        Console.WriteLine(issue);
}
catch (OperationCanceledException)
{
    Console.WriteLine("Work has been cancelled");
}
```

Remplacez-le par la boucle `await foreach` suivante :

C#

```
int num = 0;
await foreach (var issue in RunPagedQueryAsync(client, PagedIssueQuery,
"docs"))
{
    Console.WriteLine(issue);
    Console.WriteLine($"Received {++num} issues in total");
}
```

La nouvelle interface IAsyncEnumerator<T> est dérivée de IAsyncDisposable. Cela signifie que la boucle précédente supprime de façon asynchrone le flux à la fin de la boucle. Vous pouvez imaginer que la boucle ressemble au code suivant :

C#

```
int num = 0;
var enumerator = RunPagedQueryAsync(client, PagedIssueQuery,
"docs").GetAsyncEnumerator();
try
{
    while (await enumerator.MoveNextAsync())
    {
        var issue = enumerator.Current;
        Console.WriteLine(issue);
        Console.WriteLine($"Received {++num} issues in total");
    }
} finally
{
    if (enumerator != null)
        await enumerator.DisposeAsync();
}
```

Par défaut, les éléments de flux sont traités dans le contexte capturé. Si vous souhaitez désactiver la capture du contexte, utilisez la méthode d'extension TaskAsyncEnumerableExtensions.ConfigureAwait. Pour plus d'informations sur les contextes de synchronisation et la capture du contexte actuel, consultez l'article traitant de la consommation du modèle asynchrone basé sur les tâches.

Les flux asynchrones prennent en charge l'annulation en utilisant le même protocole que les autres méthodes `async`. Pour prendre en charge l'annulation, vous devez modifier la signature de la méthode d'itérateur asynchrone comme suit :

C#

```
private static async IAsyncEnumerable<JToken>
RunPagedQueryAsync(GitHubClient client,
    string queryText, string repoName, [EnumeratorCancellation]
CancellationToken cancellationToken = default)
{
```

```
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new
Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results =
JObject.Parse(response.HttpResponse.Body.ToString()!);

        int totalCount = (int)issues(results)["totalCount"]!;
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"]!;
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)
["startCursor"]!.ToString();
        issuesReturned += issues(results)["nodes"]!.Count();

        foreach (JObject issue in issues(results)["nodes"]!)
            yield return issue;
    }

    JObject issues(JObject result) => (JObject)result["data"]!
["repository"]!["issues"]!;
    JObject pageInfo(JObject result) => (JObject)issues(result)
["pageInfo"]!;
}
```

L'attribut System.Runtime.CompilerServices.EnumeratorCancellationAttribute amène le compilateur à générer du code pour le IAsyncEnumerator<T> qui fait que le jeton transmis à `GetAsyncEnumerator` est visible par le corps de l'itérateur asynchrone en tant qu'argument. À l'intérieur de `runQueryAsync`, vous pouvez examiner l'état du jeton et annuler tout travail supplémentaire si nécessaire.

Vous devez utiliser une autre méthode d'extension, WithCancellation, pour transmettre le jeton d'annulation au flux asynchrone. Vous devez modifier la boucle énumérant les problèmes comme suit :

C#

```
private static async Task EnumerateWithCancellation(GitHubClient client)
{
```

```
    int num = 0;
    var cancellation = new CancellationTokenSource();
    await foreach (var issue in RunPagedQueryAsync(client, PagedIssueQuery,
"docs")
        .WithCancellation(cancellation.Token))
    {
        Console.WriteLine(issue);
        Console.WriteLine($"Received {++num} issues in total");
    }
}
```

Vous pouvez vous procurer le code du tutoriel terminé dans ce tutoriel dans notre référentiel dotnet/docs situé dans le dossier asynchronous-programming/snippets.

Exécutez l'application terminée

Exécutez de nouveau l'application. Comparez son comportement avec le comportement de l'application de démarrage. La première page de résultats est énumérée dès qu'elle est disponible. Une pause peut être observée lorsque chaque nouvelle page est demandée et récupérée, puis les résultats de la page suivante sont rapidement énumérés. Le bloc `try` / `catch` n'est pas nécessaire pour gérer l'annulation : l'appelant peut arrêter l'énumération de la collection. Le rapport de progression est clair, car le flux asynchrone génère des résultats à mesure que chaque page est téléchargée. Le statut de chaque problème retourné est inclus de façon fluide dans la boucle `await foreach`. Vous n'avez pas besoin d'un objet de rappel pour suivre la progression.

Vous pouvez voir des améliorations lors de l'utilisation de mémoire en examinant le code. Vous n'avez plus besoin d'allouer une collection pour stocker tous les résultats avant qu'ils ne soient énumérés. L'appelant peut déterminer comment utiliser les résultats et si une collection de stockage est nécessaire.

Exécutez les applications de démarrage et les applications terminées. Ceci vous permettra d'observer les différences entre les implémentations pour vous. À la fin de ce tutoriel, vous pouvez supprimer le jeton d'accès GitHub que vous avez créé au début. Si un attaquant arrive à accéder à ce jeton, il pourrait accéder aux API GitHub à l'aide de vos informations d'identification.

Dans ce didacticiel, vous avez utilisé des flux asynchrones pour lire des éléments individuels à partir d'une API réseau qui retourne des pages de données. Les flux asynchrones peuvent également lire à partir de « flux sans fin », comme un ticker boursier ou un dispositif de capteur. L'appel à `MoveNextAsync` retourne l'élément suivant dès qu'il est disponible.

Tutoriel : Créer un gestionnaire d'interpolation de chaînes personnalisé

Article • 06/04/2023

Dans ce tutoriel, vous apprendrez à :

- ✓ Implémenter le modèle de gestionnaire d'interpolation de chaîne
- ✓ Interagir avec le récepteur dans une opération d'interpolation de chaîne.
- ✓ Ajouter des arguments au gestionnaire d'interpolation de chaîne
- ✓ Comprendre les nouvelles fonctionnalités de bibliothèque pour l'interpolation de chaîne

Prérequis

Vous devrez configurer votre ordinateur de sorte qu'il exécute .NET 6, avec le compilateur C# 10. Le compilateur C# 10 est disponible à partir de Visual Studio 2022 ou du SDK .NET 6.

Ce tutoriel suppose de connaître C# et .NET, y compris Visual Studio ou l'interface CLI .NET.

Nouveau plan

C# 10 ajoute la prise en charge d'un *gestionnaire de chaînes interpolées* personnalisé. Un gestionnaire de chaînes interpolées est un type qui traite l'expression d'espace réservé dans une chaîne interpolée. Sans gestionnaire personnalisé, les espaces réservés sont traités de la même façon que String.Format. Chaque espace réservé est mis en forme en tant que texte, puis les composants sont concaténés pour former la chaîne résultante.

Vous pouvez écrire un gestionnaire pour n'importe quel scénario dans lequel vous utilisez les informations sur la chaîne résultante. Sera-t-elle utilisée ? Quelles sont les contraintes concernant le format ? Voici quelques exemples :

- Vous pouvez exiger qu'aucune des chaînes résultantes ne dépasse une certaine limite, par exemple 80 caractères. Vous pouvez traiter les chaînes interpolées pour remplir une mémoire tampon de longueur fixe et arrêter le traitement une fois cette longueur de mémoire tampon atteinte.
- Vous pouvez avoir un format tabulaire, et chaque espace réservé doit avoir une longueur fixe. Un gestionnaire personnalisé peut appliquer ces conditions, au lieu de forcer tout le code client à se conformer.

Dans ce tutoriel, vous allez créer un gestionnaire d'interpolation de chaîne pour l'un des principaux scénarios de performances : les bibliothèques de journalisation. Selon le niveau de journalisation configuré, le travail de construction d'un message de journal n'est pas nécessaire. Si la journalisation est désactivée, le travail de construction d'une chaîne à partir d'une expression de chaîne interpolée n'est pas nécessaire. Le message n'étant jamais imprimé, toute concaténation de chaîne peut être ignorée. En outre, les expressions utilisées dans les espaces réservés, y compris la génération de traces, n'ont pas besoin d'être effectuées.

Un gestionnaire de chaîne interpolée peut déterminer si la chaîne mise en forme sera utilisée et effectuer le travail nécessaire uniquement s'il le faut.

Implémentation initiale

Commençons par une classe de base `Logger` qui prend en charge différents niveaux :

C#

```
public enum LogLevel
{
    Off,
    Critical,
    Error,
    Warning,
    Information,
    Trace
}

public class Logger
{
    public LogLevel EnabledLevel { get; init; } = LogLevel.Error;

    public void LogMessage(LogLevel level, string msg)
    {
        if (EnabledLevel < level) return;
        Console.WriteLine(msg);
    }
}
```

Cette classe `Logger` prend en charge six niveaux différents. Lorsqu'un message ne passe pas le filtre de niveau de journalisation, il n'y a pas de sortie. L'API publique de l'enregistreur d'événements accepte une chaîne (entièrement mise en forme) comme message. Tout le travail de création de la chaîne a déjà été effectué.

Implémenter le modèle de gestionnaire

Cette étape consiste à générer un *gestionnaire de chaînes interpolées* qui recrée le comportement actuel. Un gestionnaire de chaînes interpolées est un type qui doit avoir les caractéristiques suivantes :

- System.Runtime.CompilerServices.InterpolatedStringHandlerAttribute appliqué au type.
- Constructeur avec deux paramètres `int`, `literalLength` et `formatCount`. (D'autres paramètres sont autorisés).
- Méthode publique `AppendLiteral` avec la signature : `public void AppendLiteral(string s)`.
- Méthode publique `AppendFormatted` générique avec la signature : `public void AppendFormatted<T>(T t)`.

En interne, le générateur crée la chaîne mise en forme et fournit un membre pour qu'un client récupère cette chaîne. Le code suivant montre un type `LogInterpolatedStringHandler` qui répond à ces exigences :

C#

```
[InterpolatedStringHandler]
public ref struct LogInterpolatedStringHandler
{
    // Storage for the built-up string
    StringBuilder builder;

    public LogInterpolatedStringHandler(int literalLength, int
formattedCount)
    {
        builder = new StringBuilder(literalLength);
        Console.WriteLine($"\tliteral length: {literalLength},
formattedCount: {formattedCount}");
    }

    public void AppendLiteral(string s)
    {
        Console.WriteLine($"\tAppendLiteral called: {{{s}}}");

        builder.Append(s);
        Console.WriteLine($"\tAppended the literal string");
    }

    public void AppendFormatted<T>(T t)
    {
        Console.WriteLine($"\tAppendFormatted called: {{{t}}} is of type
{typeof(T)}");

        builder.Append(t?.ToString());
        Console.WriteLine($"\tAppended the formatted object");
    }
```

```
    internal string GetFormattedText() => builder.ToString();
}
```

Vous pouvez maintenant ajouter une surcharge à `LogMessage` dans la classe `Logger` pour essayer votre nouveau gestionnaire de chaîne interpolée :

C#

```
public void LogMessage(LogLevel level, LogInterpolatedStringHandler builder)
{
    if (EnabledLevel < level) return;
    Console.WriteLine(builder.GetFormattedText());
}
```

Vous n'avez pas besoin de supprimer la méthode `LogMessage` d'origine. Le compilateur préférera une méthode avec un paramètre de gestionnaire interpolé à une méthode avec un paramètre `string` lorsque l'argument est une expression de chaîne interpolée.

Vous pouvez vérifier que le nouveau gestionnaire est appelé en utilisant le code suivant comme programme principal :

C#

```
var logger = new Logger() { EnabledLevel = LogLevel.Warning };
var time = DateTime.Now;

logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime: {time}. This
is an error. It will be printed.");
logger.LogMessage(LogLevel.Trace, $"Trace Level. CurrentTime: {time}. This
won't be printed.");
logger.LogMessage(LogLevel.Warning, "Warning Level. This warning is a
string, not an interpolated string expression.");
```

L'exécution de l'application génère une sortie similaire au texte suivant :

PowerShell

```
        literal length: 65, formattedCount: 1
        AppendLiteral called: {Error Level. CurrentTime: }
        Appended the literal string
        AppendFormatted called: {10/20/2021 12:19:10 PM} is of type
System.DateTime
        Appended the formatted object
        AppendLiteral called: {. This is an error. It will be printed.}
        Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. This is an error. It will
be printed.
```

```
        literal length: 50, formattedCount: 1
        AppendLiteral called: {Trace Level. CurrentTime: }
        Appended the literal string
        AppendFormatted called: {10/20/2021 12:19:10 PM} is of type
System.DateTime
        Appended the formatted object
        AppendLiteral called: {. This won't be printed.}
        Appended the literal string
Warning Level. This warning is a string, not an interpolated string
expression.
```

En traçant la sortie, vous pouvez voir comment le compilateur ajoute du code pour appeler le gestionnaire et générer la chaîne :

- Le compilateur ajoute un appel pour construire le gestionnaire, en passant la longueur totale du texte littéral dans la chaîne de format et le nombre d'espaces réservés.
- Le compilateur ajoute des appels à `AppendLiteral` et `AppendFormatted` pour chaque section de la chaîne littérale et pour chaque espace réservé.
- Le compilateur appelle la méthode `LogMessage` à l'aide de l'argument `CoreInterpolatedStringHandler`.

Enfin, notez que le dernier avertissement n'appelle pas le gestionnaire de chaînes interpolées. L'argument est une `string`, de sorte que l'appel invoque l'autre surcharge avec un paramètre de chaîne.

Ajouter d'autres fonctionnalités au gestionnaire

La version précédente du gestionnaire de chaînes interpolées implémente le modèle. Pour éviter de traiter chaque expression d'espace réservé, vous aurez besoin d'informations supplémentaires dans le gestionnaire. Dans cette section, vous allez améliorer votre gestionnaire afin qu'il travaille moins lorsque la chaîne construite n'a pas à être écrite dans le journal. Vous utilisez System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute pour spécifier un mappage entre les paramètres d'une API publique et les paramètres du constructeur d'un gestionnaire. Cela fournit au gestionnaire les informations nécessaires pour déterminer si la chaîne interpolée doit être évaluée.

Commençons par les modifications apportées au gestionnaire. Tout d'abord, ajoutez un champ pour effectuer le suivi si le gestionnaire est activé. Ajoutez deux paramètres au constructeur : un pour spécifier le niveau de journalisation pour ce message et l'autre en tant que référence à l'objet journal :

```
```

C#

```
private readonly bool enabled;

public LogInterpolatedStringHandler(int literalLength, int formattedCount,
Logger logger, LogLevel logLevel)
{
    enabled = logger.EnabledLevel >= logLevel;
    builder = new StringBuilder(literalLength);
    Console.WriteLine($"\tliteral length: {literalLength}, formattedCount:
{formattedCount}");
}
```

Ensuite, utilisez le champ afin que votre gestionnaire ajoute uniquement des littéraux ou des objets mis en forme lorsque la chaîne finale sera utilisée :

C#

```
public void AppendLiteral(string s)
{
    Console.WriteLine($"\tAppendLiteral called: {{{s}}}");
    if (!enabled) return;

    builder.Append(s);
    Console.WriteLine($"\tAppended the literal string");
}

public void AppendFormatted<T>(T t)
{
    Console.WriteLine($"\tAppendFormatted called: {{{t}}} is of type
{typeof(T)}");
    if (!enabled) return;

    builder.Append(t?.ToString());
    Console.WriteLine($"\tAppended the formatted object");
}
```

Ensuite, vous devez mettre à jour la déclaration `LogMessage` afin que le compilateur transmette les paramètres supplémentaires au constructeur du gestionnaire. Cela est géré à l'aide de l'attribut System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute dans l'argument du gestionnaire :

C#

```
public void LogMessage(LogLevel level,
[InterpolatedStringHandlerArgument("", "level")]
LogInterpolatedStringHandler builder)
{
```

```
    if (EnabledLevel < level) return;
    Console.WriteLine(builder.GetFormattedText());
}
```

Cet attribut spécifie la liste des arguments vers `LogMessage` qui sont mappés sur les paramètres qui suivent les paramètres `literalLength` et `formattedCount` requis. La chaîne vide (« ») spécifie le récepteur. Le compilateur remplace la valeur de l'objet `Logger` représenté par `this` pour l'argument suivant au constructeur du gestionnaire. Le compilateur remplace la valeur de `level` pour l'argument suivant. Vous pouvez fournir n'importe quel nombre d'arguments pour tous les gestionnaires que vous écrivez. Les arguments que vous ajoutez sont des arguments de chaîne.

Vous pouvez exécuter cette version à l'aide du même code de test. Cette fois, vous verrez les résultats suivants :

PowerShell

```
        literal length: 65, formattedCount: 1
        AppendLiteral called: {Error Level. CurrentTime: }
        Appended the literal string
        AppendFormatted called: {10/20/2021 12:19:10 PM} is of type
System.DateTime
        Appended the formatted object
        AppendLiteral called: {. This is an error. It will be printed.}
        Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. This is an error. It will
be printed.
        literal length: 50, formattedCount: 1
        AppendLiteral called: {Trace Level. CurrentTime: }
        AppendFormatted called: {10/20/2021 12:19:10 PM} is of type
System.DateTime
        AppendLiteral called: {. This won't be printed.}
Warning Level. This warning is a string, not an interpolated string
expression.
```

Vous pouvez voir que les méthodes `AppendLiteral` et `AppendFormat` sont appelées, mais qu'elles n'effectuent aucun travail. Le gestionnaire a déterminé que la chaîne finale n'est pas nécessaire : par conséquent, le gestionnaire ne la génère pas. Il y a encore quelques améliorations à apporter.

Tout d'abord, vous pouvez ajouter une surcharge de `AppendFormatted` qui limite l'argument à un type qui implémente System.IFormattable. Cette surcharge permet aux appelants d'ajouter des chaînes de format dans les espaces réservés. Lors de cette modification, nous allons également modifier le type de retour de l'autre méthode `AppendFormatted` et `AppendLiteral`, de `void` à `bool` (si l'une de ces méthodes possède différents types de retour, vous obtiendrez une erreur de compilation). Cette

modification permet le *court-circuitage*. Les méthodes renvoient `false` pour indiquer que le traitement de l'expression de chaîne interpolée doit être arrêté. `true` indique qu'il doit continuer. Dans cet exemple, vous l'utilisez pour arrêter le traitement lorsque la chaîne résultante n'est pas nécessaire. Le court-circuitage permet des actions plus précises. Vous pouvez arrêter le traitement de l'expression une fois qu'elle atteint une certaine longueur, pour prendre en charge les mémoires tampons de longueur fixe. Ou, une condition peut indiquer que les éléments restants ne sont pas nécessaires.

C#

```
public void AppendFormatted<T>(T t, string format) where T : IFormattable
{
    Console.WriteLine($"\tAppendFormatted (IFormattable version) called: {t}
with format {{{format}}} is of type {typeof(T)},");

    builder.Append(t?.ToString(format, null));
    Console.WriteLine($"\tAppended the formatted object");
}
```

Avec cet ajout, vous pouvez spécifier des chaînes de format dans votre expression de chaîne interpolée :

C#

```
var time = DateTime.Now;

logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime: {time}. The
time doesn't use formatting.");
logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime: {time:t}. This
is an error. It will be printed.");
logger.LogMessage(LogLevel.Trace, $"Trace Level. CurrentTime: {time:t}. This
won't be printed.");
```

Le `:t` dans le premier message spécifie le « format de temps court » pour l'heure actuelle. L'exemple précédent a montré l'une des surcharges de la méthode `AppendFormatted` que vous pouvez créer pour votre gestionnaire. Vous n'avez pas besoin de spécifier d'argument générique pour l'objet mis en forme. Vous disposez peut-être de méthodes plus efficaces pour convertir les types que vous créez en chaîne. Vous pouvez écrire des surcharges de `AppendFormatted` qui acceptent ces types au lieu d'un argument générique. Le compilateur choisit la meilleure surcharge. Le runtime utilise cette technique pour convertir System.Span<T> en sortie de chaîne. Vous pouvez ajouter un paramètre entier pour spécifier l'*alignement* de la sortie, avec ou sans IFormattable. Le System.Runtime.CompilerServices.DefaultInterpolatedStringHandler fourni avec .NET 6 contient neuf surcharges de AppendFormatted pour différentes

utilisations. Vous pouvez l'utiliser comme référence lors de la création d'un gestionnaire à vos fins.

Exécutez l'exemple maintenant ; vous verrez que pour le message `Trace`, seul le premier `AppendLiteral` est appelé :

PowerShell

```
        literal length: 60, formattedCount: 1
        AppendLiteral called: Error Level. CurrentTime:
        Appended the literal string
        AppendFormatted called: 10/20/2021 12:18:29 PM is of type
System.DateTime
        Appended the formatted object
        AppendLiteral called: . The time doesn't use formatting.
        Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:18:29 PM. The time doesn't use
formatting.
        literal length: 65, formattedCount: 1
        AppendLiteral called: Error Level. CurrentTime:
        Appended the literal string
        AppendFormatted (IFormattable version) called: 10/20/2021 12:18:29
PM with format {t} is of type System.DateTime,
        Appended the formatted object
        AppendLiteral called: . This is an error. It will be printed.
        Appended the literal string
Error Level. CurrentTime: 12:18 PM. This is an error. It will be printed.
        literal length: 50, formattedCount: 1
        AppendLiteral called: Trace Level. CurrentTime:
Warning Level. This warning is a string, not an interpolated string
expression.
```

Vous pouvez effectuer une dernière mise à jour du constructeur du gestionnaire pour améliorer l'efficacité. Le gestionnaire peut ajouter un paramètre `out bool` final. La définition de ce paramètre `false` sur indique que le gestionnaire ne doit pas être appelé du tout pour traiter l'expression de chaîne interpolée :

C#

```
public LogInterpolatedStringHandler(int literalLength, int formattedCount,
Logger logger, LogLevel level, out bool isEnabled)
{
    isEnabled = logger.EnabledLevel >= level;
    Console.WriteLine($"\tliteral length: {literalLength}, formattedCount:
{formattedCount}");
    builder = isEnabled ? new StringBuilder(literalLength) : default!;
}
```

Cette modification signifie que vous pouvez supprimer le champ `enabled`. Ensuite, vous pouvez modifier le type de retour de `AppendLiteral` et `AppendFormatted` vers `void`. À présent, lorsque vous exécutez l'exemple, vous obtenez la sortie suivante :

PowerShell

```
        literal length: 60, formattedCount: 1
        AppendLiteral called: Error Level. CurrentTime:
        Appended the literal string
        AppendFormatted called: 10/20/2021 12:19:10 PM is of type
System.DateTime
        Appended the formatted object
        AppendLiteral called: . The time doesn't use formatting.
        Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. The time doesn't use
formatting.
        literal length: 65, formattedCount: 1
        AppendLiteral called: Error Level. CurrentTime:
        Appended the literal string
        AppendFormatted (IFormattable version) called: 10/20/2021 12:19:10
PM with format {t} is of type System.DateTime,
        Appended the formatted object
        AppendLiteral called: . This is an error. It will be printed.
        Appended the literal string
Error Level. CurrentTime: 12:19 PM. This is an error. It will be printed.
        literal length: 50, formattedCount: 1
Warning Level. This warning is a string, not an interpolated string
expression.
```

La seule sortie quand `LogLevel.Trace` a été spécifié est la sortie du constructeur. Le gestionnaire a indiqué qu'il n'est pas activé ; aucune des méthodes `Append` n'a donc été appelée.

Cet exemple illustre un point important pour les gestionnaires de chaînes interpolées, en particulier lorsque des bibliothèques de journalisation sont utilisées. Les effets secondaires dans les espaces réservés peuvent ne pas tous se produire. Ajoutez le code suivant à votre programme principal et voyez ce comportement en action :

C#

```
int index = 0;
int numberOfIncrements = 0;
for (var level = LogLevel.Critical; level <= LogLevel.Trace; level++)
{
    Console.WriteLine(level);
    logger.LogMessage(level, $"{level}: Increment index a few times
{index++}, {index++}, {index++}, {index++}, {index++}");
    numberOfIncrements += 5;
}
```

```
Console.WriteLine($"Value of index {index}, value of numberOfIncrements:
{numberOfIncrements}");
```

Vous pouvez voir que la variable `index` est incrémentée cinq fois à chaque itération de la boucle. Étant donné que les espaces réservés sont évalués uniquement pour les niveaux `Critical`, `Error` et `Warning`, et non pour `Information` et `Trace`, la valeur finale de `index` ne correspond pas aux attentes :

PowerShell

```
Critical
Critical: Increment index a few times 0, 1, 2, 3, 4
Error
Error: Increment index a few times 5, 6, 7, 8, 9
Warning
Warning: Increment index a few times 10, 11, 12, 13, 14
Information
Trace
Value of index 15, value of numberOfIncrements: 25
```

Les gestionnaires de chaînes interpolées offrent un meilleur contrôle sur la façon dont une expression de chaîne interpolée est convertie en chaîne. L'équipe du runtime .NET a déjà utilisé cette fonctionnalité pour améliorer les performances dans plusieurs domaines. Vous pouvez utiliser la même fonctionnalité dans vos propres bibliothèques. Pour explorer plus loin, examinez le System.Runtime.CompilerServices.DefaultInterpolatedStringHandler. Il fournit une implémentation plus complète que celle que vous avez créée ici. Vous verrez de nombreuses autres surcharges possibles pour les méthodes `Append`.

Utiliser l'interpolation de chaîne pour construire des chaînes mises en forme

Article • 10/05/2023

Ce tutoriel explique comment utiliser l'interpolation de chaîne en C# pour insérer des valeurs dans une chaîne de résultat unique. Vous allez écrire un code en C# et afficher les résultats de la compilation et de l'exécution du code. Le tutoriel contient une série de leçons qui expliquent comment insérer des valeurs dans une chaîne et mettre en forme ces valeurs de différentes façons.

Ce tutoriel suppose que vous disposez d'un ordinateur que vous pouvez utiliser pour le développement. Le tutoriel .NET Hello World en 10 minutes contient des instructions pour configurer un environnement de développement local sur Windows, Linux ou macOS. Vous pouvez également suivre la version interactive de ce tutoriel dans votre navigateur.

Créer une chaîne interpolée

Créez un répertoire nommé *interpolated*. Faites-en le répertoire actif et exécutez la commande suivante à partir d'une fenêtre de console :

CLI .NET

```
dotnet new console
```

Cette commande crée une nouvelle application console .NET Core dans le répertoire actuel.

Ouvrez *Program.cs* dans votre éditeur favori, puis remplacez la ligne `Console.WriteLine("Hello World!");` par le code qui suit, en remplaçant `<name>` par votre nom :

C#

```
var name = "<name>";
Console.WriteLine($"Hello, {name}. It's a pleasure to meet you!");
```

Essayez ce code en tapant `dotnet run` dans la fenêtre de console. Quand vous exécutez le programme, celui-ci affiche une chaîne unique qui inclut votre nom dans le message d'accueil. La chaîne qui se trouve dans l'appel de méthode WriteLine est une *expression*

de chaîne interpolée. C'est un genre de modèle qui vous permet de construire une chaîne unique (appelée *chaîne de résultat*) à partir d'une chaîne qui comprend du code incorporé. Les chaînes interpolées sont particulièrement utiles pour insérer des valeurs dans une chaîne ou pour concaténer (joindre) des chaînes.

Cet exemple simple contient les deux éléments que chaque chaîne interpolée doit avoir :

- Un littéral de chaîne qui commence par le caractère `$` avant ses guillemets ouvrants. Il ne peut pas y avoir d'espace entre le symbole `$` et les guillemets. (Si vous voulez voir ce qui se passe si vous en incluez un, insérez un espace après le caractère `$`, enregistrez le fichier et réexécutez le programme en tapant `dotnet run` dans la fenêtre de console. Le compilateur C# affiche un message d'erreur, « erreur CS1056 : Caractère « $ » inattendu ».)

- Une ou plusieurs *expressions d'interpolation*. Une expression d'interpolation est indiquée par des accolades ouvrantes et fermantes (`{` et `}`). Vous pouvez placer n'importe quelle expression C# qui retourne une valeur (notamment `null`) à l'intérieur des accolades.

Essayons quelques autres exemples d'interpolation de chaîne avec d'autres types de données.

Inclure différents types de données

Dans la section précédente, vous avez utilisé l'interpolation de chaîne pour insérer une chaîne à l'intérieur d'une autre. Le résultat d'une expression d'interpolation peut toutefois être de n'importe quel type de données. Nous allons insérer des valeurs de différents types de données dans une chaîne interpolée.

Dans l'exemple suivant, nous commençons par définir une classe comme type de données `Vegetable` avec la propriété `Name` et la méthode `ToString`, qui remplace le comportement de la méthode Object.ToString(). Le modificateur d'accès public permet à n'importe quel code client d'obtenir la représentation sous forme de chaîne d'une instance de `Vegetable`. Dans l'exemple, la méthode `Vegetable.ToString` retourne la valeur de la propriété `Name` qui est initialisée au niveau du constructeur `Vegetable` :

```
C#

public Vegetable(string name) => Name = name;
```

Ensuite, nous créons une instance de la classe `Vegetable` nommée `item` en utilisant l'opérateur new et en ajoutant un nom pour le constructeur `Vegetable` :

C#

```
var item = new Vegetable("eggplant");
```

Enfin, nous incluons la variable `item` dans une chaîne interpolée qui contient également une valeur DateTime, une valeur Decimal et une valeur d'énumération `Unit`. Remplacez tout le code C# dans votre éditeur par le code suivant, puis utilisez la commande `dotnet run` pour l'exécuter :

C#

```
using System;

public class Vegetable
{
   public Vegetable(string name) => Name = name;

   public string Name { get; }

   public override string ToString() => Name;
}

public class Program
{
   public enum Unit { item, kilogram, gram, dozen };

   public static void Main()
   {
      var item = new Vegetable("eggplant");
      var date = DateTime.Now;
      var price = 1.99m;
      var unit = Unit.item;
      Console.WriteLine($"On {date}, the price of {item} was {price} per
{unit}.");
   }
}
```

Notez que l'expression d'interpolation `item` dans la chaîne interpolée résout le texte « eggplant » (aubergine) dans la chaîne de résultat. En effet, lorsque le résultat de l'expression n'est pas de type chaîne, il est résolu en une chaîne de la façon suivante :

- Si l'expression d'interpolation donne une valeur `null`, une chaîne vide («» ou String.Empty) est utilisée.
- Si l'expression d'interpolation ne donne pas une valeur `null`, en général, la méthode `ToString` du type de résultat est appelée. Vous pouvez tester cela en mettant à jour l'implémentation de la méthode `Vegetable.ToString`. Il n'est peut-

être même pas nécessaire d'implémenter la méthode `ToString` étant donné que chaque type a déjà une implémentation de cette méthode. Pour tester cela, commentez la définition de la méthode `Vegetable.ToString` dans l'exemple (en la faisant précéder d'un symbole de commentaire `//`). Dans la sortie, la chaîne « eggplant » (aubergines) est remplacée par le nom de type complet (« Vegetable » dans cet exemple), ce qui est le comportement par défaut de la méthode Object.ToString(). Le comportement par défaut de la méthode `ToString` pour une valeur d'énumération est de retourner la représentation sous forme de chaîne de la valeur.

Dans la sortie de cet exemple, la date est trop précise (le prix des aubergines ne change pas chaque seconde) et la valeur du prix n'indique pas la devise locale. Dans la section suivante, vous découvrirez comment résoudre ces problèmes en contrôlant le format des représentations sous forme de chaînes des résultats des expressions.

Contrôler la mise en forme des expressions d'interpolation

Dans la section précédente, deux chaînes à la mise en forme incorrecte ont été insérées dans la chaîne de résultat. L'une était une valeur de date et d'heure pour laquelle seule la date était appropriée. La deuxième était un prix qui n'indiquait pas la devise locale. Ces deux problèmes sont faciles à résoudre. En utilisant l'interpolation de chaîne, vous pouvez spécifier des *chaînes de format* qui contrôlent la mise en forme de types particuliers. Modifiez l'appel à `Console.WriteLine` de l'exemple précédent de façon à inclure les chaînes de format pour les expressions de date et de prix, comme indiqué dans la ligne de code suivante :

```
C#

Console.WriteLine($"On {date:d}, the price of {item} was {price:C2} per
{unit}.");
```

Vous spécifiez une chaîne de format en plaçant après l'expression d'interpolation un signe deux-points (« : ») et la chaîne de format. « d » est une chaîne de format de date et d'heure standard qui représente le format de date courte. « C2 » est une chaîne de format numérique standard qui représente un nombre sous forme de valeur monétaire avec deux chiffres après la virgule.

Plusieurs types dans les bibliothèques .NET prennent en charge un ensemble prédéfini de chaînes de format. Il s'agit notamment de tous les types numériques et des types de date et d'heure. Pour obtenir une liste complète des types qui prennent en charge les

chaînes de format, consultez Chaînes de format et types de bibliothèque de classes .NET dans l'article Mise en forme des types dans .NET.

Essayez de modifier les chaînes de format dans votre éditeur de texte et, à chaque modification, relancez le programme pour voir comment celles-ci affectent la mise en forme de la date et de l'heure et la valeur numérique. Remplacez le « d » dans `{date:d}` par « t » (pour afficher le format d'heure courte), « y » (pour afficher l'année et mois) et « yyyy » (pour afficher l'année sous forme de nombre à quatre chiffres). Remplacez le « C2 » dans `{price:C2}` par « e » (pour la notation exponentielle) et « F3 » (pour une valeur numérique avec trois chiffres après la virgule).

En plus de contrôler la mise en forme, vous pouvez contrôler la largeur de champ et l'alignement des chaînes mises en forme qui sont incluses dans la chaîne de résultat. Dans la section suivante, vous allez découvrir comment effectuer cette opération.

Contrôler la largeur de champ et l'alignement des expressions d'interpolation

En règle générale, lorsque le résultat d'une expression d'interpolation est mise en forme en chaîne, cette chaîne est inclue dans une chaîne de résultats sans espace de début ou de fin. Contrôler la largeur d'un champ et l'alignement du texte vous permet de rendre une sortie plus lisible, en particulier quand vous utilisez un jeu de données. Pour voir cela, remplacez tout le code dans votre éditeur de texte par le code suivant, puis tapez `dotnet run` pour exécuter le programme :

C#

```
using System;
using System.Collections.Generic;

public class Example
{
   public static void Main()
   {
      var titles = new Dictionary<string, string>()
      {
          ["Doyle, Arthur Conan"] = "Hound of the Baskervilles, The",
          ["London, Jack"] = "Call of the Wild, The",
          ["Shakespeare, William"] = "Tempest, The"
      };

      Console.WriteLine("Author and Title List");
      Console.WriteLine();
      Console.WriteLine($"|{"Author",-25}|{"Title",30}|");
      foreach (var title in titles)
         Console.WriteLine($"|{title.Key,-25}|{title.Value,30}|");
```

```
   }
}
```

Les noms des auteurs sont alignés à gauche, et leurs titres sont alignés à droite. Vous spécifiez l'alignement en ajoutant une virgule (« , ») après une expression d'interpolation et en spécifiant la largeur de champ *minimum*. Si la valeur spécifiée est un nombre positif, le champ est aligné à droite. Si c'est un nombre négatif, le champ est aligné à gauche.

Essayez de supprimer les signes négatifs du code `{"Author",-25}` et `{title.Key,-25}`, puis réexécutez l'exemple, comme avec le code suivant :

C#

```
Console.WriteLine($"|{"Author",25}|{"Title",30}|");
foreach (var title in titles)
   Console.WriteLine($"|{title.Key,25}|{title.Value,30}|");
```

Cette fois, les informations sur l'auteur sont alignées à droite.

Vous pouvez combiner un spécificateur d'alignement et une chaîne de format pour une même expression d'interpolation. Pour cela, spécifiez d'abord l'alignement, puis un signe deux-points et la chaîne de format. Remplacez tout le code à l'intérieur de la méthode `Main` par le code suivant, qui affiche trois chaînes mises en forme avec des largeurs de champ définies. Ensuite, exécutez le programme en entrant la commande `dotnet run`.

C#

```
Console.WriteLine($"[{DateTime.Now,-20:d}] Hour [{DateTime.Now,-10:HH}]
[{1063.342,15:N2}] feet");
```

La sortie ressemble à ceci :

Console

```
[04/14/2018          ] Hour [16        ] [       1,063.34] feet
```

Vous avez terminé le tutoriel sur l'interpolation de chaîne.

Pour plus d'informations, consultez la rubrique Interpolation de chaîne et le tutoriel Interpolation de chaîne en C#.

Interpolation de chaîne en C#

Article • 02/09/2023

Ce tutoriel vous montre comment utiliser une interpolation de chaîne pour mettre en forme et inclure des résultats d'expressions dans une chaîne de résultat. Les exemples supposent que vous êtes familiarisé avec les concepts de base de C# et la mise en forme des types .NET. Si vous ne connaissez pas l'interpolation de chaînes ou la mise en forme de types .NET, consultez d'abord le tutoriel interactif sur l'interpolation de chaînes. Pour plus d'informations sur la mise en forme des types dans .NET, consultez Mise en forme des types dans .NET.

Présentation

Pour identifier un littéral de chaîne comme chaîne interpolée, préfixez-la du symbole `$`. Vous pouvez incorporer n'importe quelle expression C# valide qui retourne une valeur dans une chaîne interpolée. Dans l'exemple suivant, dès qu'une expression est évaluée, son résultat est converti en chaîne et est inclus dans une chaîne de résultat :

C#

```
double a = 3;
double b = 4;
Console.WriteLine($"Area of the right triangle with legs of {a} and {b} is
{0.5 * a * b}");
Console.WriteLine($"Length of the hypotenuse of the right triangle with legs
of {a} and {b} is {CalculateHypotenuse(a, b)}");
double CalculateHypotenuse(double leg1, double leg2) => Math.Sqrt(leg1 *
leg1 + leg2 * leg2);
// Output:
// Area of the right triangle with legs of 3 and 4 is 6
// Length of the hypotenuse of the right triangle with legs of 3 and 4 is 5
```

Comme le montre l'exemple, vous incluez une expression dans une chaîne interpolée en la plaçant entre des accolades :

C#

```
{<interpolationExpression>}
```

Les chaînes interpolées prennent en charge toutes les fonctions de la fonctionnalité Mise en forme de chaîne composite. Elles constituent ainsi une alternative plus lisible à l'utilisation de la méthode String.Format.

Comment spécifier une chaîne de format pour une expression d'interpolation

Pour spécifier une chaîne de format prise en charge par le type du résultat d'expression, placez un signe deux-points (« : ») et la chaîne de format après l'expression d'interpolation :

C#

```
{<interpolationExpression>:<formatString>}
```

L'exemple suivant montre comment spécifier des chaînes de format standard et personnalisées pour des expressions qui produisent des résultats de type date/heure ou numérique :

C#

```
var date = new DateTime(1731, 11, 25);
Console.WriteLine($"On {date:dddd, MMMM dd, yyyy} L. Euler introduced the
letter e to denote {Math.E:F5}.");
// Output:
// On Sunday, November 25, 1731 L. Euler introduced the letter e to denote
2.71828.
```

Pour plus d'informations, consultez la section Composant de chaîne de format de l'article Mise en forme composite.

Comment contrôler la largeur de champ et l'alignement de l'expression d'interpolation mise en forme

Pour spécifier la largeur minimale du champ et l'alignement du résultat de l'expression mise en forme, placez une virgule (« , ») et l'expression constante après l'expression d'interpolation :

C#

```
{<interpolationExpression>,<alignment>}
```

Si la valeur de *l'alignement* est positive, le résultat de l'expression mise en forme est aligné à droite ; si la valeur est négative, il est aligné à gauche.

Si vous devez spécifier à la fois un alignement et une chaîne de format, commencez par le composant d'alignement :

```
C#

{<interpolationExpression>,<alignment>:<formatString>}
```

L'exemple suivant montre comment spécifier l'alignement ; il utilise des caractères de barre verticale (« | ») pour délimiter les champs texte :

```
C#

const int NameAlignment = -9;
const int ValueAlignment = 7;
double a = 3;
double b = 4;
Console.WriteLine($"Three classical Pythagorean means of {a} and {b}:");
Console.WriteLine($"|{"Arithmetic",NameAlignment}|{0.5 * (a +
b),ValueAlignment:F3}|");
Console.WriteLine($"|{"Geometric",NameAlignment}|{Math.Sqrt(a *
b),ValueAlignment:F3}|");
Console.WriteLine($"|{"Harmonic",NameAlignment}|{2 / (1 / a + 1 /
b),ValueAlignment:F3}|");
// Output:
// Three classical Pythagorean means of 3 and 4:
// |Arithmetic|   3.500|
// |Geometric|   3.464|
// |Harmonic |   3.429|
```

Comme le montre l'exemple de sortie, si la longueur du résultat d'expression mise en forme dépasse la largeur de champ spécifiée, la valeur de *l'alignement* est ignorée.

Pour plus d'informations, consultez la section Composant d'alignement de l'article Mise en forme composite.

Comment utiliser des séquences d'échappement dans une chaîne interpolée

Les chaînes interpolées prennent en charge toutes les séquences d'échappement qui peuvent être utilisés dans les littéraux de chaîne ordinaires. Pour plus d'informations, consultez Séquences d'échappement de chaîne.

Pour interpréter les séquences d'échappement littéralement, utilisez un littéral de chaîne textuelle. Une chaîne verbatim interpolée commence par les caractères `$` et `@`. Vous

pouvez utiliser `$` et `@` dans n'importe quel ordre : `$@"..."` et `@$"..."` sont toutes deux des chaînes verbatim interpolées valides.

Pour inclure une accolade, « { » ou «} », dans une chaîne de résultat, utilisez deux accolades, « {{ » ou «}} ». Pour plus d'informations, consultez la section Accolades d'échappement de l'article Mise en forme composite.

L'exemple suivant montre comment inclure des accolades dans une chaîne de résultat et construire une chaîne interpolée textuelle :

C#

```
var xs = new int[] { 1, 2, 7, 9 };
var ys = new int[] { 7, 9, 12 };
Console.WriteLine($"Find the intersection of the {{{string.Join(", ",xs)}}}
and {{{string.Join(", ",ys)}}} sets.");
// Output:
// Find the intersection of the {1, 2, 7, 9} and {7, 9, 12} sets.

var userName = "Jane";
var stringWithEscapes = $"C:\\Users\\{userName}\\Documents";
var verbatimInterpolated = $@"C:\Users\{userName}\Documents";
Console.WriteLine(stringWithEscapes);
Console.WriteLine(verbatimInterpolated);
// Output:
// C:\Users\Jane\Documents
// C:\Users\Jane\Documents
```

À compter de C# 11, vous pouvez utiliser des littéraux de chaîne brute interpolés.

Comment utiliser un opérateur conditionnel ternaire `?:` dans une expression d'interpolation

Comme le caractère deux-points (« : ») a une signification spéciale dans un élément avec une expression d'interpolation, pour utiliser un opérateur conditionnel dans une expression, placez-la entre parenthèses, comme le montre l'exemple suivant :

C#

```
var rand = new Random();
for (int i = 0; i < 7; i++)
{
    Console.WriteLine($"Coin flip: {(rand.NextDouble() < 0.5 ? "heads" :
"tails")}");
}
```

Comment créer une chaîne de résultat spécifique à une culture avec une interpolation de chaîne

Par défaut, une chaîne interpolée utilise la culture active définie par la propriété CultureInfo.CurrentCulture pour toutes les opérations de mise en forme.

À compter de .NET 6, vous pouvez utiliser la méthode String.Create(IFormatProvider, DefaultInterpolatedStringHandler) pour résoudre une chaîne interpolée en chaîne de résultat spécifique à la culture, comme l'illustre l'exemple suivant :

C#

```
var cultures = new System.Globalization.CultureInfo[]
{
    System.Globalization.CultureInfo.GetCultureInfo("en-US"),
    System.Globalization.CultureInfo.GetCultureInfo("en-GB"),
    System.Globalization.CultureInfo.GetCultureInfo("nl-NL"),
    System.Globalization.CultureInfo.InvariantCulture
};
var date = DateTime.Now;
var number = 31_415_926.536;
foreach (var culture in cultures)
{
    var cultureSpecificMessage = string.Create(culture, $"{date,23}
{number,20:N3}");
    Console.WriteLine($"{culture.Name,-10}{cultureSpecificMessage}");
}
// Output is similar to:
// en-US       8/27/2023 12:35:31 PM      31,415,926.536
// en-GB         27/08/2023 12:35:31      31,415,926.536
// nl-NL         27-08-2023 12:35:31      31.415.926,536
//               08/27/2023 12:35:31      31,415,926.536
```

Dans les versions antérieures de .NET, utilisez la conversion implicite d'une chaîne interpolée en une instance de System.FormattableString et appelez sa méthode ToString(IFormatProvider) pour créer une chaîne de résultat spécifique à une culture. L'exemple suivant montre comment effectuer cette opération :

C#

```
var cultures = new System.Globalization.CultureInfo[]
{
    System.Globalization.CultureInfo.GetCultureInfo("en-US"),
    System.Globalization.CultureInfo.GetCultureInfo("en-GB"),
    System.Globalization.CultureInfo.GetCultureInfo("nl-NL"),
    System.Globalization.CultureInfo.InvariantCulture
```

```
};
var date = DateTime.Now;
var number = 31_415_926.536;
FormattableString message = $"{date,23}{number,20:N3}";
foreach (var culture in cultures)
{
    var cultureSpecificMessage = message.ToString(culture);
    Console.WriteLine($"{culture.Name,-10}{cultureSpecificMessage}");
}
// Output is similar to:
// en-US       8/27/2023 12:35:31 PM      31,415,926.536
// en-GB         27/08/2023 12:35:31      31,415,926.536
// nl-NL         27-08-2023 12:35:31      31.415.926,536
//               08/27/2023 12:35:31      31,415,926.536
```

Comme le montre l'exemple, vous pouvez utiliser une même instance de FormattableString pour générer plusieurs chaînes de résultat pour différentes cultures.

Comment créer une chaîne de résultat avec la culture invariante

À compter de .NET 6, utilisez la méthode String.Create(IFormatProvider, DefaultInterpolatedStringHandler) pour résoudre une chaîne interpolée en chaîne de résultat pour le InvariantCulture, comme l'illustre l'exemple suivant :

C#

```
string message = string.Create(CultureInfo.InvariantCulture, $"Date and time
in invariant culture: {DateTime.Now}");
Console.WriteLine(message);
// Output is similar to:
// Date and time in invariant culture: 05/17/2018 15:46:24
```

Dans les versions antérieures de .NET, ainsi qu'avec la méthode FormattableString.ToString(IFormatProvider), vous pouvez utiliser la méthode FormattableString.Invariant statique, comme l'illustre l'exemple suivant :

C#

```
string message = FormattableString.Invariant($"Date and time in invariant
culture: {DateTime.Now}");
Console.WriteLine(message);
// Output is similar to:
// Date and time in invariant culture: 05/17/2018 15:46:24
```

Conclusion

Ce tutoriel décrit des scénarios courants d'utilisation de l'interpolation de chaîne. Pour plus d'informations sur l'interpolation de chaîne, consultez Interpolation de chaîne. Pour plus d'informations sur la mise en forme des types dans .NET, consultez les articles Mise en forme des types dans .NET et Mise en forme composite.

Voir aussi

- String.Format
- System.FormattableString
- System.IFormattable
- Chaînes

Application console

Article • 14/03/2023

Ce didacticiel vous présente un certain nombre de fonctionnalités de .NET et du langage C#. Vous apprendrez ce qui suit :

- Principes de base du CLI .NET
- La structure d'une application de console C#
- E/S console
- Les principes fondamentaux des API d'E/S de fichier dans .NET
- Les principes fondamentaux de la programmation asynchrone basée sur des tâches dans .NET

Vous allez générer une application qui lit un fichier texte et retourne le contenu du fichier texte dans la console. La sortie sur la console se fait à un rythme permettant de la lire à haute voix. Vous pouvez accélérer ou ralentir la vitesse en appuyant sur les touches '<' (inférieur à) ou '>' (supérieur à). Vous pouvez exécuter cette application sur Windows, Linux, macOS ou dans un conteneur Docker.

Il existe un grand nombre de fonctionnalités dans ce didacticiel. Nous allons les construire une par une.

Prérequis

- .NET 6 SDK .
- Éditeur de code.

Créer l'application

La première étape consiste à créer une nouvelle application. Ouvrez une invite de commandes et créez un nouveau répertoire pour votre application. Réglez-le comme répertoire actuel. Saisissez la commande `dotnet new console` à l'invite. Elle crée les fichiers de démarrage d'une application « Hello World » de base.

Avant d'apporter des modifications, exécutons l'application simple Hello World. Après avoir créé l'application, saisissez `dotnet run` à l'invite de commandes. Cette commande exécute le processus de restauration de package NuGet, crée l'exécutable de l'application et l'exécute.

Le code d'application Hello World simple se trouve intégralement dans le fichier *Program.cs*. Ouvrez ce fichier avec votre éditeur de texte préféré. Remplacez le code

dans *Program.cs* par le code suivant :

```C#
namespace TeleprompterConsole;

internal class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
    }
}
```

En haut du fichier, remarquez l'instruction `namespace`. Comme d'autres langages orientés objet que vous avez peut-être utilisés, C# utilise des espaces de noms pour organiser les types. Ce programme Hello World ne fait pas exception. Vous pouvez voir que le programme est placé dans l'espace de noms avec le nom `TeleprompterConsole`.

Lecture et affichage du fichier

La première fonctionnalité à ajouter est la capacité à lire un fichier texte et à afficher tout le texte dans la console. Tout d'abord, nous allons ajouter un fichier texte. Copiez le fichier sampleQuotes.txt à partir du dépôt GitHub pour cet exemple dans votre répertoire de projet. Il servira de script pour votre application. Pour plus d'informations sur le téléchargement de l'exemple d'application pour ce didacticiel, consultez les instructions dans Exemples et tutoriels.

Ensuite, ajoutez la méthode suivante dans votre classe `Program` (juste en dessous de la méthode `Main`) :

```C#
static IEnumerable<string> ReadFrom(string file)
{
    string? line;
    using (var reader = File.OpenText(file))
    {
        while ((line = reader.ReadLine()) != null)
        {
            yield return line;
        }
    }
}
```

Cette méthode est un type spécial de méthode C# appelé *méthode d'itérateur*. Les méthodes d'itérateur retournent des séquences qui sont évaluées de manière tardive. Cela signifie que chaque élément de la séquence est généré lorsque cela est demandé par le code utilisant la séquence. Les méthodes d'itérateur sont des méthodes qui contiennent une ou plusieurs instructions yield return. L'objet retourné par la méthode `ReadFrom` contient le code pour générer chaque élément dans la séquence. Dans cet exemple, cela implique la lecture de la ligne de texte suivante à partir du fichier source et le renvoi de cette chaîne. Chaque fois que le code appelant demande l'élément suivant de la séquence, le code lit la ligne suivante du texte à partir du fichier et la renvoie. Quand le fichier est entièrement lu, la séquence indique qu'il n'y a pas d'autres éléments.

Il existe deux éléments de syntaxe de C# qui peuvent être nouveaux pour vous. L'instruction using dans cette méthode gère le nettoyage des ressources. La variable initialisée dans l'instruction `using` (`reader`, dans cet exemple) doit implémenter l'interface IDisposable. Cette interface définit une méthode unique, `Dispose`, qui doit être appelée lorsque les ressources doivent être libérées. Le compilateur génère l'appel lorsque l'exécution atteint l'accolade fermante de l'instruction `using`. Le code généré par le compilateur garantit que la ressource est libérée même si une exception est levée à partir du code dans le bloc défini par l'instruction using.

La variable `reader` est définie à l'aide du mot-clé `var`. var définit une *variable locale implicitement typée*. Cela signifie que le type de la variable est déterminé par le type au moment de la compilation de l'objet assigné à la variable. Ici, c'est la valeur retournée par la méthode OpenText(String), qui est un objet StreamReader.

À présent, nous allons remplir le code pour lire le fichier dans la méthode `Main` :

C#

```
var lines = ReadFrom("sampleQuotes.txt");
foreach (var line in lines)
{
    Console.WriteLine(line);
}
```

Exécutez le programme (à l'aide de `dotnet run`). Chaque ligne est envoyée sur la console.

Ajout de délais et de mise en forme à la sortie

Ce que vous avez s'affiche beaucoup trop rapidement pour le lire à haute voix. Vous devez maintenant ajouter des délais à la sortie. Lorsque vous commencez, vous créez une partie du code principal qui permet le traitement asynchrone. Toutefois, ces premières étapes suivront quelques anti-modèles. Les anti-modèles sont signalés dans les commentaires lorsque vous ajoutez le code, et le code sera actualisé ultérieurement.

Il existe deux étapes dans cette section. Tout d'abord, vous allez mettre à jour la méthode d'itérateur pour retourner des mots uniques au lieu de lignes entières. Vous le faites avec ces modifications. Remplacez l'instruction `yield return line;` par le code suivant :

C#

```
var words = line.Split(' ');
foreach (var word in words)
{
    yield return word + " ";
}
yield return Environment.NewLine;
```

Ensuite, vous devez modifier la façon dont vous consommez les lignes du fichier, et ajoutez un délai après chaque mot. Remplacez l'instruction `Console.WriteLine(line)` dans la méthode `Main` avec le bloc suivant :

C#

```
Console.Write(line);
if (!string.IsNullOrWhiteSpace(line))
{
    var pause = Task.Delay(200);
    // Synchronously waiting on a task is an
    // anti-pattern. This will get fixed in later
    // steps.
    pause.Wait();
}
```

Exécutez l'exemple et vérifiez le résultat. À présent, chaque mot unique est affiché séparément, suivi d'un délai de 200 ms. Toutefois, la sortie affichée présente certains problèmes, car le fichier texte source possède plusieurs lignes qui ont plus de 80 caractères sans saut de ligne. Ce qui peut être difficile à lire lors du défilement. Ce problème est facile à résoudre. Vous allez simplement effectuer le suivi de la longueur de chaque ligne et générer une nouvelle ligne chaque fois que la longueur de ligne atteint un certain seuil. Déclarez une variable locale après la déclaration de `words` dans la méthode `ReadFrom` qui contient la longueur de ligne :

C#

```
var lineLength = 0;
```

Ensuite, ajoutez le code suivant après l'instruction `yield return word + " ";` (avant l'accolade fermante) :

C#

```
lineLength += word.Length + 1;
if (lineLength > 70)
{
    yield return Environment.NewLine;
    lineLength = 0;
}
```

Exécutez l'exemple et vous serez en mesure de lire à haute voix au rythme préconfiguré.

Tâches asynchrones

Dans cette dernière étape, vous allez ajouter le code pour écrire la sortie de façon asynchrone dans une tâche, lorsque vous exécutez également une autre tâche pour lire l'entrée de l'utilisateur s'il souhaite accélérer ou ralentir l'affichage du texte, ou l'arrêter. Cela représente plusieurs étapes et à la fin, vous aurez toutes les mises à jour dont vous avez besoin. La première étape consiste à créer une méthode de retour Task asynchrone qui représente le code que vous avez créé jusqu'à présent pour lire et afficher le fichier.

Ajoutez cette méthode à votre classe `Program` (elle est extraite du corps de votre méthode `Main`) :

C#

```
private static async Task ShowTeleprompter()
{
    var words = ReadFrom("sampleQuotes.txt");
    foreach (var word in words)
    {
        Console.Write(word);
        if (!string.IsNullOrWhiteSpace(word))
        {
            await Task.Delay(200);
        }
    }
}
```

Vous remarquerez deux modifications. Tout d'abord, dans le corps de la méthode, au lieu d'appeler Wait() pour attendre de manière synchrone qu'une tâche se termine, cette version utilise le mot-clé `await`. Pour ce faire, vous devez ajouter le modificateur `async` pour la signature de méthode. Cette méthode renvoie une `Task`. Notez qu'il n'y a aucune instruction de retour renvoyant un objet `Task`. Au lieu de cela, cet objet `Task` est créé par le code que le compilateur génère lorsque vous utilisez l'opérateur `await`. Vous pouvez imaginer que cette méthode renvoie une valeur lorsqu'elle atteint un `await`. La `Task` renvoyée indique que la tâche n'a pas été effectuée. La méthode reprend lorsque la tâche attendue se termine. Lorsqu'elle a terminé son exécution, la `Task` renvoyée indique qu'elle est terminée. Le code appelant peut surveiller cette `Task` renvoyée pour déterminer si elle est terminée.

Ajoutez un mot clé `await` avant l'appel à `ShowTeleprompter` :

C#

```
await ShowTeleprompter();
```

Pour cela, vous devez modifier la signature de la méthode `Main` en :

C#

```
static async Task Main(string[] args)
```

Pour en savoir plus sur la async Main méthode, consultez la section relative aux notions de base.

Ensuite, vous devez écrire la seconde méthode asynchrone pour lire à partir de la console et chercher les touches '<' (inférieur à), '>' (supérieur à) et 'X' ou 'x'. Voici la méthode que vous ajoutez pour cette tâche :

C#

```
private static async Task GetInput()
{
    var delay = 200;
    Action work = () =>
    {
        do {
            var key = Console.ReadKey(true);
            if (key.KeyChar == '>')
            {
                delay -= 10;
            }
            else if (key.KeyChar == '<')
```

```
            {
                delay += 10;
            }
            else if (key.KeyChar == 'X' || key.KeyChar == 'x')
            {
                break;
            }
        } while (true);
    };
    await Task.Run(work);
}
```

Cela crée une expression lambda pour représenter un délégué Action qui lit une touche de la console et modifie une variable locale représentant le délai lorsque l'utilisateur appuie sur les touches '<' (inférieur à) et '>' (supérieur à). La méthode déléguée se termine lorsque l'utilisateur appuie sur les touches 'X' ou 'x', ce qui autorise l'utilisateur à arrêter l'affichage du texte à tout moment. Cette méthode utilise ReadKey() pour bloquer et attendre que l'utilisateur appuie sur une touche.

Pour terminer cette fonctionnalité, vous devez créer une nouvelle méthode de retour `async Task` qui démarre ces deux tâches (`GetInput` et `ShowTeleprompter`) et gère également les données partagées entre ces deux tâches.

Il est temps de créer une classe qui peut gérer les données partagées entre ces deux tâches. Cette classe contient deux propriétés publiques : le délai et un indicateur `Done` pour indiquer que le fichier a été entièrement lu :

C#

```
namespace TeleprompterConsole;

internal class TelePrompterConfig
{
    public int DelayInMilliseconds { get; private set; } = 200;
    public void UpdateDelay(int increment) // negative to speed up
    {
        var newDelay = Min(DelayInMilliseconds + increment, 1000);
        newDelay = Max(newDelay, 20);
        DelayInMilliseconds = newDelay;
    }
    public bool Done { get; private set; }
    public void SetDone()
    {
        Done = true;
    }
}
```

Placez cette classe dans un nouveau fichier et incluez-la à l'espace de noms `TeleprompterConsole`, comme indiqué. Vous devrez également ajouter une instruction `using static` en haut du fichier pour pouvoir référencer les méthodes `Min` et `Max` sans les noms de la classe ou de l'espace de noms qui les contient. Une instruction using static importe les méthodes d'une classe. Cela contraste avec l'instruction `using` sans `static`, qui importe toutes les classes à partir d'un espace de noms.

C#

```
using static System.Math;
```

Ensuite, vous devez mettre à jour les méthodes `ShowTeleprompter` et `GetInput` pour utiliser le nouvel objet `config`. Écrivez une dernière `Task` renvoyant une méthode `async` pour démarrer les deux tâches et quitter lorsque la première tâche est terminée :

C#

```
private static async Task RunTeleprompter()
{
    var config = new TelePrompterConfig();
    var displayTask = ShowTeleprompter(config);

    var speedTask = GetInput(config);
    await Task.WhenAny(displayTask, speedTask);
}
```

Ici, la nouvelle méthode est l'appel WhenAny(Task[]). Elle crée une `Task` qui se termine dès que les tâches de sa liste d'arguments se terminent.

Ensuite, vous devez mettre à jour les méthodes `ShowTeleprompter` et `GetInput` pour utiliser l'objet `config` pour le délai :

C#

```
private static async Task ShowTeleprompter(TelePrompterConfig config)
{
    var words = ReadFrom("sampleQuotes.txt");
    foreach (var word in words)
    {
        Console.Write(word);
        if (!string.IsNullOrWhiteSpace(word))
        {
            await Task.Delay(config.DelayInMilliseconds);
        }
    }
    config.SetDone();
}
```

```
private static async Task GetInput(TelePrompterConfig config)
{
    Action work = () =>
    {
        do {
            var key = Console.ReadKey(true);
            if (key.KeyChar == '>')
                config.UpdateDelay(-10);
            else if (key.KeyChar == '<')
                config.UpdateDelay(10);
            else if (key.KeyChar == 'X' || key.KeyChar == 'x')
                config.SetDone();
        } while (!config.Done);
    };
    await Task.Run(work);
}
```

Cette nouvelle version de `ShowTeleprompter` appelle une nouvelle méthode dans la classe `TelePrompterConfig`. À présent, vous devez mettre à jour `Main` pour appeler `RunTeleprompter` au lieu de `ShowTeleprompter` :

C#

```
await RunTeleprompter();
```

Conclusion

Ce didacticiel vous a montré certaines des fonctionnalités du langage C# et les bibliothèques .NET Core liées au travail dans les applications console. Vous pouvez utiliser ces connaissances pour explorer davantage le langage ainsi que les classes présentées ici. Vous avez vu les principes de base des E/S de fichier et de console, l'utilisation bloquante et non bloquante de la programmation asynchrone basée sur des tâches, une présentation du langage C# et de la façon dont les programmes C# sont organisés, et le CLI .NET.

Pour plus d'informations sur les E/S de fichier, consultez Fichier et flux de données E/S. Pour plus d'informations sur le modèle de programmation asynchrone utilisé dans ce didacticiel, consultez Programmation asynchrone basée sur les tâches et Asynchronous programming.

Tutoriel : Effectuer des requêtes HTTP dans une application console .NET en utilisant C#

Article • 10/05/2023

Ce didacticiel génère une application qui émet des requêtes HTTP vers un service REST sur GitHub. L'application lit les informations au format JSON et les convertit en objets C#. La conversion d'objets JSON en objets C# est appelée *désérialisation*.

Le didacticiel montre comment :

- ✓ Envoyer des requêtes HTTP.
- ✓ Désérialiser les réponses JSON.
- ✓ Configurer la désérialisation avec des attributs.

Si vous préférez utiliser l'échantillon final pour ce didacticiel, vous pouvez le télécharger. Pour obtenir des instructions de téléchargement, consultez Exemples et didacticiels.

Prérequis

- SDK .NET 6.0 ou ultérieur
- Un éditeur de code tel que [Visual Studio Code (éditeur multiplateforme open source). Vous pouvez exécuter l'échantillon d'application sous Windows, Linux ou macOS, ou dans un conteneur Docker.

Créer l'application cliente

1. Ouvrez une invite de commandes et créez un nouveau répertoire pour votre application. Réglez-le comme répertoire actuel.

2. Entrez la commande suivante dans la fenêtre de console :

   CLI .NET

   ```
   dotnet new console --name WebAPIClient
   ```

   Cette commande crée les fichiers de démarrage d'une application « Hello World » de base. Le nom du projet est « WebAPIClient ».

3. Accédez au répertoire « WebAPIClient » et exécutez l'application.

CLI .NET

```
cd WebAPIClient
```

CLI .NET

```
dotnet run
```

dotnet run s'exécute dotnet restore automatiquement pour restaurer toutes les dépendances dont l'application a besoin. Exécute dotnet build également si nécessaire. Vous devez voir la sortie `"Hello, World!"` de l'application. Dans votre terminal, appuyez sur `Ctrl`+`C` pour arrêter l'application.

Effectuer des requêtes HTTP

Cette application appelle l'API GitHub pour obtenir des informations sur les projets sous .NET Foundation . Le point de terminaison a la valeur https://api.github.com/orgs/dotnet/repos. Pour récupérer des informations, elle effectue une requête HTTP GET. Les navigateurs font également des requêtes HTTP GET, vous pouvez donc coller cette URL dans votre barre d'adresse pour voir les informations que vous recevez et traitez.

Utilisez la HttpClient classe pour effectuer des requêtes HTTP. HttpClient prend en charge uniquement les méthodes Async pour ses API de longue durée. Ainsi, les étapes suivantes créent une méthode asynchrone et l'appellent à partir de la méthode Main.

1. Ouvrez le fichier `Program.cs` dans votre répertoire de projet et remplacez son contenu par ceci :

   C#

   ```
   await ProcessRepositoriesAsync();

   static async Task ProcessRepositoriesAsync(HttpClient client)
   {
   }
   ```

   Ce code :

   - Remplace l'instruction `Console.WriteLine` par un appel à `ProcessRepositoriesAsync` qui utilise le mot clé `await`.
   - Définit une méthode `ProcessRepositoriesAsync` vide.

2. Dans la classe `Program`, utilisez un HttpClient pour gérer les demandes et les réponses, en remplaçant le contenu par le C# suivant.

```C#
using System.Net.Http.Headers;

using HttpClient client = new();
client.DefaultRequestHeaders.Accept.Clear();
client.DefaultRequestHeaders.Accept.Add(
    new
MediaTypeWithQualityHeaderValue("application/vnd.github.v3+json"));
client.DefaultRequestHeaders.Add("User-Agent", ".NET Foundation
Repository Reporter");

await ProcessRepositoriesAsync(client);

static async Task ProcessRepositoriesAsync(HttpClient client)
{
}
```

Ce code :

- Configure les en-têtes HTTP pour toutes les requêtes :
  - Un en-tête Accept pour accepter les réponses JSON
  - En-tête de User-Agent. Ces deux en-têtes sont vérifiés par le code du serveur GitHub et sont nécessaires pour récupérer des informations à partir de GitHub.

3. Dans la méthode `ProcessRepositoriesAsync`, appelez le point de terminaison GitHub qui retourne une liste de tous les référentiels sous l'organisation .NET Foundation :

```C#
static async Task ProcessRepositoriesAsync(HttpClient client)
{
    var json = await client.GetStringAsync(
        "https://api.github.com/orgs/dotnet/repos");

    Console.Write(json);
}
```

Ce code :

- Attend la tâche retournée par la méthode appelante HttpClient.GetStringAsync(String). Cette méthode envoie une requête HTTP

GET à l'URI spécifié. Le corps de la réponse est retourné en tant que String, qui est disponible lorsque la tâche se termine.
- La chaîne de réponse `json` est imprimée dans la console.

4. Générez l'application et exécutez-la.

```
CLI .NET
```

```
dotnet run
```

Il n'y a pas d'avertissement de build, car `ProcessRepositoriesAsync` contient maintenant un opérateur `await`. La sortie est un long affichage du texte JSON.

Désérialiser le résultat JSON

Les étapes suivantes convertissent la réponse JSON en objets C#. Vous utilisez la classe System.Text.Json.JsonSerializer pour désérialiser JSON en objets.

1. Créez un fichier nommé *Repository.cs* et ajoutez le code suivant :

```
C#
```

```
public record class Repository(string name);
```

Le code précédent définit une classe pour représenter l'objet JSON retourné par l'API GitHub. Vous utiliserez cette classe pour afficher une liste de noms de référentiels.

Le JSON d'un objet de référentiel contient des dizaines de propriétés, mais seule la propriété `name` sera désérialisée. Le sérialiseur ignore automatiquement les propriétés JSON pour lesquelles il n'existe aucune correspondance dans la classe cible. Cette fonctionnalité facilite la création de types qui fonctionnent avec uniquement un sous-ensemble des champs dans un paquet JSON volumineux.

La convention C# consiste à mettre en majuscules la première lettre des noms de propriétés, mais la propriété `name` commence ici par une lettre minuscule, car cela correspond exactement à ce qui se trouve dans le JSON. Plus tard, vous verrez comment utiliser des noms de propriétés C# qui ne correspondent pas aux noms de propriétés JSON.

2. Utilisez le sérialiseur pour convertir le JSON en objets C#. Remplacez l'appel à GetStringAsync(String) dans la méthode `ProcessRepositoriesAsync` avec les lignes

suivantes :

```
C#
```

```
await using Stream stream =
    await
client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
var repositories =
    await JsonSerializer.DeserializeAsync<List<Repository>>(stream);
```

Le code mis à jour remplace GetStringAsync(String) par GetStreamAsync(String). Cette méthode de sérialiseur utilise un flux plutôt qu'une chaîne comme source.

Le premier argument JsonSerializer.DeserializeAsync<TValue>(Stream, JsonSerializerOptions, CancellationToken) est une expression `await`. Les expressions `await` peuvent apparaître quasiment n'importe où dans votre code, bien que jusqu'à présent, vous les avez uniquement vues dans le cadre d'une instruction d'assignation. Les deux autres paramètres, `JsonSerializerOptions` et `CancellationToken`, sont facultatifs et sont omis dans l'extrait de code.

La `DeserializeAsync` méthode est *générique*, ce qui signifie que vous fournissez des arguments de type pour le type d'objets à créer à partir du texte JSON. Dans cet exemple, vous désérialisez un `List<Repository>`, qui est un autre objet générique, un System.Collections.Generic.List<T>. La classe `List<T>` stocke une collection d'objets. L'argument type déclare le type d'objets stockés dans la `List<T>`. L'argument type est votre enregistrement `Repository`, car le texte JSON représente une collection d'objets de référentiel.

3. Ajoutez du code pour afficher le nom de chaque référentiel. Remplacez les lignes :

```
C#
```

```
Console.Write(json);
```

par le code suivant :

```
C#
```

```
foreach (var repo in repositories ?? Enumerable.Empty<Repository>())
    Console.Write(repo.name);
```

4. Les directives suivantes `using` doivent être présentes en haut du fichier :

```
C#
```

```
using System.Net.Http.Headers;
using System.Text.Json;
```

5. Exécutez l'application.

CLI .NET

```
dotnet run
```

La sortie est une liste des noms des référentiels qui font partie de la .NET Foundation.

Configurer la désérialisation

1. Dans *Repository.cs*, remplacez le contenu du fichier par le code C# suivant.

C#

```
using System.Text.Json.Serialization;

public record class Repository(
    [property: JsonPropertyName("name")] string Name);
```

Ce code :

- Remplace le nom de la propriété `name` par `Name`.
- Ajoute le JsonPropertyNameAttribute pour spécifier comment cette propriété apparaît dans le JSON.

2. Dans *Program.cs*, mettez à jour le code pour utiliser la nouvelle majuscule de la propriété `Name` :

C#

```
foreach (var repo in repositories)
   Console.Write(repo.Name);
```

3. Exécutez l'application.

La sortie est la même.

Refactoriser le code

La méthode `ProcessRepositoriesAsync` peut faire le travail de façon asynchrone et renvoyer une collection des espaces de stockage. Modifiez cette méthode pour retourner `Task<List<Repository>>`, puis déplacez le code qui écrit dans la console près de son appelant.

1. Modifiez la signature de `ProcessRepositoriesAsync` pour retourner une tâche dont le résultat est une liste d'objets `Repository` :

   ```
   C#
   ```

   ```
   static async Task<List<Repository>> ProcessRepositoriesAsync()
   ```

2. Retournez simplement les dépôts après le traitement de la réponse JSON :

   ```
   C#
   ```

   ```
   await using Stream stream =
       await
   client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
   var repositories =
       await JsonSerializer.DeserializeAsync<List<Repository>>(stream);
   return repositories ?? new();
   ```

   Le compilateur génère l'objet `Task<T>` pour la valeur renvoyée, car vous avez marqué cette méthode comme `async`.

3. Modifiez le fichier *Program.cs*, en remplaçant l'appel à `ProcessRepositoriesAsync` par ce qui suit pour capturer les résultats et écrire le nom de chaque référentiel dans la console.

   ```
   C#
   ```

   ```
   var repositories = await ProcessRepositoriesAsync(client);

   foreach (var repo in repositories)
       Console.Write(repo.Name);
   ```

4. Exécutez l'application.

   La sortie est la même.

Désérialiser d'autres propriétés

Les étapes suivantes ajoutent du code pour traiter davantage de propriétés dans le paquet JSON reçu. Il est peu probable que vous souhaitiez traiter toutes les propriétés, mais en ajouter quelques autres permet de démontrer d'autres fonctionnalités de C#.

1. Remplacez le contenu de la classe `Repository` par la définition suivante `record` :

   C#

   ```
   using System.Text.Json.Serialization;

   public record class Repository(
       [property: JsonPropertyName("name")] string Name,
       [property: JsonPropertyName("description")] string Description,
       [property: JsonPropertyName("html_url")] Uri GitHubHomeUrl,
       [property: JsonPropertyName("homepage")] Uri Homepage,
       [property: JsonPropertyName("watchers")] int Watchers);
   ```

   Les Uri types et `int` disposent de fonctionnalités intégrées pour les convertir vers et à partir de la représentation sous forme de chaîne. Aucun code supplémentaire n'est nécessaire pour désérialiser ces types cibles à partir du format de chaîne JSON. Si le paquet JSON contient des données qui ne sont pas converties en type cible, l'action de sérialisation lève une exception.

2. Mettez à jour la `foreach` boucle dans le fichier *Program.cs* pour afficher les valeurs de propriété :

   C#

   ```
   foreach (var repo in repositories)
   {
       Console.WriteLine($"Name: {repo.Name}");
       Console.WriteLine($"Homepage: {repo.Homepage}");
       Console.WriteLine($"GitHub: {repo.GitHubHomeUrl}");
       Console.WriteLine($"Description: {repo.Description}");
       Console.WriteLine($"Watchers: {repo.Watchers:#,0}");
       Console.WriteLine();
   }
   ```

3. Exécutez l'application.

   La liste inclut désormais les propriétés supplémentaires.

Ajouter une propriété de date

La date de la dernière opération push est mise en forme de cette façon dans la réponse JSON :

JSON

```
2016-02-08T21:27:00Z
```

Ce format est pour le temps universel coordonné (UTC), de sorte que le résultat de la désérialisation est une DateTime valeur dont Kind la propriété est Utc.

Pour obtenir une date et une heure représentées dans votre fuseau horaire, vous devez écrire une méthode de conversion personnalisée.

1. Dans *Repository.cs*, ajoutez une propriété pour la représentation UTC de la date et de l'heure et une propriété readonly `LastPush` qui retourne la date convertie en heure locale, le fichier doit ressembler à ce qui suit :

   C#

   ```
   using System.Text.Json.Serialization;

   public record class Repository(
       [property: JsonPropertyName("name")] string Name,
       [property: JsonPropertyName("description")] string Description,
       [property: JsonPropertyName("html_url")] Uri GitHubHomeUrl,
       [property: JsonPropertyName("homepage")] Uri Homepage,
       [property: JsonPropertyName("watchers")] int Watchers,
       [property: JsonPropertyName("pushed_at")] DateTime LastPushUtc)
   {
       public DateTime LastPush => LastPushUtc.ToLocalTime();
   }
   ```

   La propriété `LastPush` est définie à l'aide d'un *membre expression-bodied* pour l'accesseur `get` . Il n'y a pas d'accesseur `set`. Omettre l'accesseur `set` est une façon de définir une propriété en *lecture seule* en C#. (Oui, vous pouvez créer des propriétés *en écriture seule* en C#, mais leur intérêt est limité.)

2. Ajoutez une autre instruction de sortie dans *Program.cs* à nouveau :

   C#

   ```
   Console.WriteLine($"Last push: {repo.LastPush}");
   ```

3. L'application complète doit ressembler au fichier *Program.cs* suivant :

   C#

   ```
   using System.Net.Http.Headers;
   using System.Text.Json;
   ```

```
using HttpClient client = new();
client.DefaultRequestHeaders.Accept.Clear();
client.DefaultRequestHeaders.Accept.Add(
    new
MediaTypeWithQualityHeaderValue("application/vnd.github.v3+json"));
client.DefaultRequestHeaders.Add("User-Agent", ".NET Foundation
Repository Reporter");

var repositories = await ProcessRepositoriesAsync(client);

foreach (var repo in repositories)
{
    Console.WriteLine($"Name: {repo.Name}");
    Console.WriteLine($"Homepage: {repo.Homepage}");
    Console.WriteLine($"GitHub: {repo.GitHubHomeUrl}");
    Console.WriteLine($"Description: {repo.Description}");
    Console.WriteLine($"Watchers: {repo.Watchers:#,0}");
    Console.WriteLine($"{repo.LastPush}");
    Console.WriteLine();
}

static async Task<List<Repository>> ProcessRepositoriesAsync(HttpClient
client)
{
    await using Stream stream =
        await
client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
    var repositories =
        await JsonSerializer.DeserializeAsync<List<Repository>>
(stream);
    return repositories ?? new();
}
```

4. Exécutez l'application.

   La sortie inclut la date et l'heure du dernier envoi (push) à chaque référentiel.

Étapes suivantes

Dans ce didacticiel, vous avez créé une application qui effectue des requêtes web et analyse les résultats. Votre version de l'application doit maintenant correspondre à l'échantillon terminé.

Pour en savoir plus sur la configuration de la sérialisation JSON, consultez Guide pratique pour sérialiser et désérialiser (marshaler et démarshaler) JSON dans .NET.

Utiliser LINQ (Language Integrated Query)

Article • 10/05/2023

Introduction

Ce tutoriel vous présente les fonctionnalités de .NET Core et du langage C#. Vous découvrirez comment :

- Générer des séquences avec LINQ.
- Écrire des méthodes pouvant être facilement utilisées dans des requêtes LINQ.
- Faire la distinction entre l'évaluation stricte et l'évaluation paresseuse.

Vous apprendrez ces techniques en créant une application qui illustre l'une des compétences de base de tout magicien : le mélange faro. En quelques mots, le mélange faro est une technique qui consiste à diviser un paquet de cartes en deux moitiés exactes, puis à intercaler une carte sur deux de chacune des deux moitiés de façon à reconstruire le jeu d'origine.

Les magiciens utilisent cette technique parce que chaque carte se trouve à un emplacement connu après chaque mélange, suivant un motif répétitif.

Dans notre cas, c'est une façon plaisante d'envisager la manipulation de séquences de données. L'application que vous allez créer construit un jeu de cartes, puis procède à une séquence de mélanges, en affichant la séquence à chaque fois. Vous comparerez également le nouvel ordre à l'ordre d'origine.

Ce didacticiel comporte plusieurs étapes. Après chaque étape, vous pourrez exécuter l'application et voir la progression. Vous pouvez également voir l'exemple terminé dans le dépôt GitHub dotnet/samples. Pour obtenir des instructions de téléchargement, consultez Exemples et didacticiels.

Prérequis

Vous devez configurer votre ordinateur pour exécuter .NET core. Vous trouverez les instructions d'installation dans la page de téléchargement de .NET Core. Vous pouvez exécuter cette application sur Windows, Ubuntu Linux, OS X ou dans un conteneur Docker. Vous devez installer l'éditeur de code de votre choix. Les descriptions ci-dessous utilisent Visual Studio Code, éditeur multiplateforme open source. Cependant, vous pouvez utiliser les outils avec lesquels vous êtes le plus à l'aise.

Création de l'application

La première étape consiste à créer une nouvelle application. Ouvrez une invite de commandes et créez un nouveau répertoire pour votre application. Réglez-le comme répertoire actuel. Saisissez la commande `dotnet new console` à l'invite. Elle crée les fichiers de démarrage d'une application « Hello World » de base.

Si vous n'avez jamais utilisé C#, ce didacticiel explique la structure d'un programme C#. Vous pouvez le lire, puis revenir ici pour en savoir plus sur LINQ.

Créer le jeu de données

Avant de commencer, vérifiez que les lignes suivantes figurent en haut du fichier `Program.cs` généré par `dotnet new console` :

C#

```
// Program.cs
using System;
using System.Collections.Generic;
using System.Linq;
```

Si ces trois lignes (instructions `using`) ne figurent pas en haut du fichier, le programme ne se compilera pas.

Maintenant que vous avez toutes les références dont vous avez besoin, réfléchissons à ce qui compose un jeu de cartes. En général, un jeu de cartes à jouer comporte quatre couleurs, et chaque couleur a treize valeurs. On pourrait envisager de créer directement une classe `Card` et de remplir manuellement une collection d'objets `Card`. Avec LINQ, il est possible d'être plus concis que cela. Au lieu de créer une classe `Card`, vous pouvez créer deux séquences pour représenter respectivement les couleurs et les rangs. Créez deux *méthodes d'itération* simples qui génèreront les rangs et les couleurs sous forme de IEnumerable<T> de chaînes :

C#

```
// Program.cs
// The Main() method

static IEnumerable<string> Suits()
{
    yield return "clubs";
    yield return "diamonds";
    yield return "hearts";
```

```
    yield return "spades";
}

static IEnumerable<string> Ranks()
{
    yield return "two";
    yield return "three";
    yield return "four";
    yield return "five";
    yield return "six";
    yield return "seven";
    yield return "eight";
    yield return "nine";
    yield return "ten";
    yield return "jack";
    yield return "queen";
    yield return "king";
    yield return "ace";
}
```

Placez-les sous la méthode `Main` dans votre fichier `Program.cs`. Les deux méthodes utilisent la syntaxe `yield return` pour produire une séquence lors de leur exécution. Le compilateur génère un objet qui implémente IEnumerable<T> et génère la séquence de chaînes au fur et à mesure.

Maintenant, utilisez ces méthodes d'itération pour créer le jeu de cartes. Placez la requête LINQ dans la méthode `Main` :

C#

```
// Program.cs
static void Main(string[] args)
{
    var startingDeck = from s in Suits()
                       from r in Ranks()
                       select new { Suit = s, Rank = r };

    // Display each card that we've generated and placed in startingDeck in
the console
    foreach (var card in startingDeck)
    {
        Console.WriteLine(card);
    }
}
```

Les clauses `from` multiples produisent un SelectMany, qui crée une séquence unique en combinant chaque élément de la première séquence avec chaque élément de la deuxième séquence. L'ordre est important ici. Le premier élément de la première séquence source (couleurs) est associé à chacun des éléments de la deuxième séquence

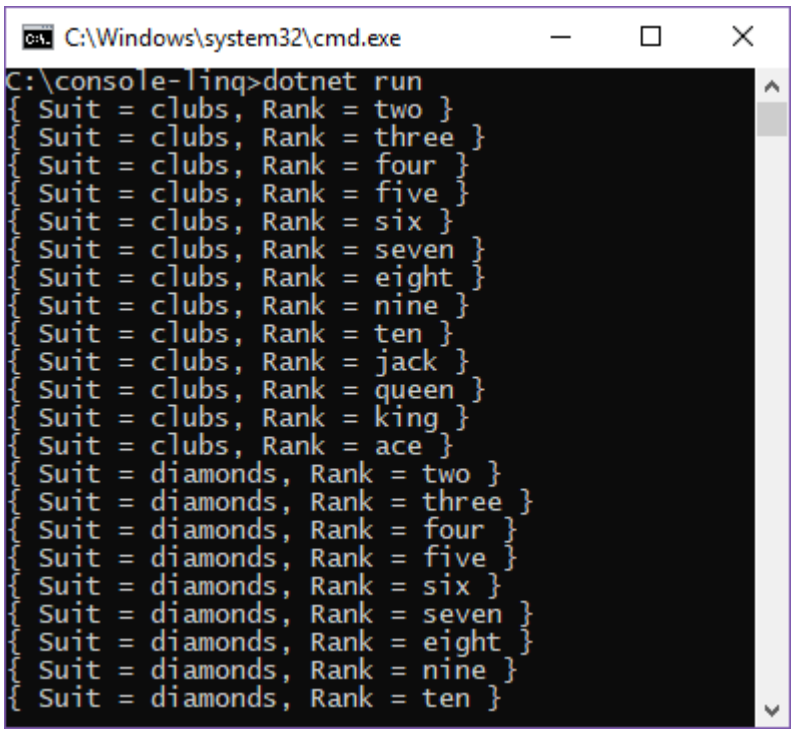
(rangs). Cette opération génère les treize cartes de la première couleur. Ce processus est reproduit pour chaque élément de la première séquence (couleurs). Le résultat final est un jeu de cartes classé par couleurs, puis par valeurs.

Il est important de garder à l'esprit la chose suivante : que vous choisissiez d'écrire votre code LINQ dans la syntaxe de requête utilisée plus haut ou d'utiliser plutôt la syntaxe de méthode, vous pourrez toujours passer d'une forme syntaxique à l'autre. La requête ci-dessus, écrite dans la syntaxe de requête, s'écrit ainsi dans la syntaxe de méthode :

```
C#

var startingDeck = Suits().SelectMany(suit => Ranks().Select(rank => new {
Suit = suit, Rank = rank }));
```

Le compilateur traduit les instructions LINQ écrites avec la syntaxe de requête dans la syntaxe d'appel de méthode équivalente. Par conséquent, les deux versions de la requête produisent le même résultat quel que soit le choix de syntaxe. Choisissez la plus adaptée à votre situation : par exemple, si certains membres de votre équipe ont des difficultés à utiliser la syntaxe de méthode, privilégiez dans la mesure du possible la syntaxe de requête.

Ensuite, exécutez l'exemple que vous avez commencé à élaborer. Il affiche les 52 cartes du jeu. Il peut être très utile d'exécuter cet exemple avec un débogueur pour observer la façon dont les méthodes `Suits()` et `Ranks()` s'exécutent. Vous pouvez clairement voir que chaque chaîne de chaque séquence est générée uniquement au moment requis.



Manipuler l'ordre

Maintenant, réfléchissons à la façon dont nous allons battre les cartes du jeu. Pour bien faire, la première étape consiste à couper le jeu en deux. Les méthodes Take et Skip, qui font partie des API LINQ, offrent cette fonctionnalité. Placez-les sous la boucle `foreach` :

C#

```
// Program.cs
public static void Main(string[] args)
{
    var startingDeck = from s in Suits()
                       from r in Ranks()
                       select new { Suit = s, Rank = r };

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    // 52 cards in a deck, so 52 / 2 = 26
    var top = startingDeck.Take(26);
    var bottom = startingDeck.Skip(26);
}
```

Cependant, il n'y a pas de méthode de battage dans la bibliothèque standard ; vous devez donc écrire la vôtre. Comme elle illustrera plusieurs techniques des programmes LINQ, nous allons expliquer les différentes parties du processus étape par étape.

Pour ajouter des fonctionnalités aux interactions possibles avec les IEnumerable<T> obtenus à partir des requêtes LINQ, vous allez écrire des méthodes d'un genre particulier, nommées méthodes d'extension. En bref, une méthode d'extension est une *méthode statique* spéciale qui ajoute de nouvelles fonctionnalités à un type existant sans qu'il soit nécessaire de modifier le type d'origine.

Pour accueillir vos méthodes d'extension, ajoutez à votre programme un nouveau fichier de classe *statique*, nommé `Extensions.cs`, puis créez la première méthode d'extension :

C#

```
// Extensions.cs
using System;
using System.Collections.Generic;
using System.Linq;

namespace LinqFaroShuffle
{
    public static class Extensions
```

```
    {
        public static IEnumerable<T> InterleaveSequenceWith<T>(this
IEnumerable<T> first, IEnumerable<T> second)
        {
            // Your implementation will go here soon enough
        }
    }
}
```

Examinez attentivement la signature de méthode, et en particulier les paramètres :

C#

```
public static IEnumerable<T> InterleaveSequenceWith<T> (this IEnumerable<T>
first, IEnumerable<T> second)
```

On peut voir l'ajout du modificateur `this` au premier argument de la méthode. Cela signifie que vous appelez la méthode comme s'il s'agissait d'une méthode membre du type du premier argument. Cette déclaration de méthode suit également un idiome standard selon lequel les types d'entrée et de sortie sont `IEnumerable<T>`. Cette pratique permet d'enchaîner les méthodes LINQ afin d'exécuter des requêtes plus complexes.

Bien entendu, le jeu ayant été coupé en deux, il faut réunir les deux moitiés. Dans le code, cela signifie énumérer d'un seul coup les deux séquences acquises avec Take et Skip, en intercalant (*`interleaving`*) les éléments pour créer une seule séquence : le jeu de cartes battu. Pour écrire une méthode LINQ qui fonctionne avec deux séquences, vous devez comprendre comment IEnumerable<T> fonctionne.

L'interface IEnumerable<T> possède une seule méthode : GetEnumerator. L'objet retourné par GetEnumerator a une méthode permettant d'atteindre l'élément suivant et une propriété qui récupère l'élément actif dans la séquence. Vous allez utiliser ces deux membres pour énumérer la collection et retourner les éléments. Cette méthode Interleave sera une méthode d'itération ; par conséquent, au lieu de créer une collection et de la retourner, vous allez utiliser la syntaxe `yield return` présentée ci-dessus.

Voici l'implémentation de cette méthode :

C#

```
public static IEnumerable<T> InterleaveSequenceWith<T>
    (this IEnumerable<T> first, IEnumerable<T> second)
{
    var firstIter = first.GetEnumerator();
    var secondIter = second.GetEnumerator();

    while (firstIter.MoveNext() && secondIter.MoveNext())
```

```
    {
        yield return firstIter.Current;
        yield return secondIter.Current;
    }
}
```

Maintenant que vous avez écrit cette méthode, revenez à la méthode `Main` et mélangez le jeu une fois :

C#

```
// Program.cs
public static void Main(string[] args)
{
    var startingDeck = from s in Suits()
                       from r in Ranks()
                       select new { Suit = s, Rank = r };

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    var top = startingDeck.Take(26);
    var bottom = startingDeck.Skip(26);
    var shuffle = top.InterleaveSequenceWith(bottom);

    foreach (var c in shuffle)
    {
        Console.WriteLine(c);
    }
}
```

Comparaisons

Combien de battages faut-il compter pour remettre le jeu dans l'ordre d'origine ? Pour le savoir, il faut écrire une méthode qui détermine si deux séquences sont égales. Ensuite, il vous faudra placer le code qui mélange le jeu dans une boucle et vérifier si le jeu est de nouveau dans l'ordre.

Vous ne devriez pas avoir de problèmes à écrire une méthode qui détermine si deux séquences sont égales. La structure est similaire à celle de la méthode que vous avez écrite pour mélanger le jeu. Seulement, cette fois, au lieu d'utiliser l'instruction `yield return` sur chaque élément, vous allez comparer les éléments correspondants de chaque séquence. Une fois que la séquence aura été énumérée en entier, si tous les éléments correspondent, les séquences sont les mêmes :

C#

```
public static bool SequenceEquals<T>
    (this IEnumerable<T> first, IEnumerable<T> second)
{
    var firstIter = first.GetEnumerator();
    var secondIter = second.GetEnumerator();

    while ((firstIter?.MoveNext() == true) && secondIter.MoveNext())
    {
        if ((firstIter.Current is not null) &&
!firstIter.Current.Equals(secondIter.Current))
        {
            return false;
        }
    }

    return true;
}
```

Cet exemple montre un autre terme LINQ : les méthodes terminales. Elles prennent une séquence en entrée (ou, dans ce cas, deux séquences) et retournent une valeur scalaire unique. Les méthodes terminales sont toujours la dernière méthode de la chaîne de méthodes d'une requête LINQ ; d'où le nom « terminal ».

Vous pourrez les voir en action lorsque vous les utiliserez pour déterminer si le jeu est dans l'ordre d'origine. Placez le code de mélange à l'intérieur d'une boucle, et arrêtez-la lorsque la séquence est à nouveau dans l'ordre d'origine en appliquant la méthode `SequenceEquals()`. Vous pouvez voir qu'il s'agit toujours de la dernière méthode d'une requête, parce qu'elle retourne une valeur unique plutôt qu'une séquence :

C#

```
// Program.cs
static void Main(string[] args)
{
    // Query for building the deck

    // Shuffling using InterleaveSequenceWith<T>();

    var times = 0;
    // We can re-use the shuffle variable from earlier, or you can make a
new one
    shuffle = startingDeck;
    do
    {
        shuffle = shuffle.Take(26).InterleaveSequenceWith(shuffle.Skip(26));

        foreach (var card in shuffle)
        {
```

```
            Console.WriteLine(card);
        }
        Console.WriteLine();
        times++;

    } while (!startingDeck.SequenceEquals(shuffle));

    Console.WriteLine(times);
}
```

Exécutez le code ainsi obtenu et étudiez la façon dont le jeu se réorganise à chaque battage. Après huit battages (itérations de la boucle do-while), le jeu revient à la configuration qu'il avait à sa création avec la requête LINQ initiale.

Optimisations

L'exemple que vous avez produit jusque-là exécute un *mélange extérieur*, où les cartes du haut et du bas restent les mêmes à chaque exécution. Faisons une modification en effectuant plutôt un *mélange intérieur*, où les 52 cartes changent toutes de position. Dans le cas d'un mélange intérieur, on intercale les deux moitiés du jeu de sorte que la première carte de la moitié du dessous devienne la première carte du jeu. Cela signifie que la dernière carte de la moitié du dessus devient la carte du bas. Cette modification simple ne concerne qu'une ligne de code. Mettez à jour la requête de battage actuelle en inversant les positions de Take et de Skip, de façon à modifier l'ordre des moitiés du dessus et du dessous du jeu :

C#

```
shuffle = shuffle.Skip(26).InterleaveSequenceWith(shuffle.Take(26));
```

Réexécutez le programme : vous verrez qu'il faut 52 itérations pour que le jeu se réorganise. Vous commencerez également à remarquer de sérieuses dégradations des performances lors de l'exécution du programme.

Il y a plusieurs raisons à cela. L'une des causes principales de cette baisse de performances est une utilisation inefficace de *l'évaluation paresseuse*.

En bref, l'évaluation d'une instruction n'est effectuée que lorsque sa valeur devient nécessaire. Les requêtes LINQ sont évaluées de cette manière. Les séquences sont générées uniquement au moment où les éléments sont demandés. En règle générale, c'est un avantage majeur de LINQ. Toutefois, dans une utilisation du type de ce programme, cela entraîne une croissance exponentielle de la durée d'exécution.

N'oubliez pas que nous avons généré le jeu d'origine à l'aide d'une requête LINQ. Chaque mélange est généré en effectuant trois requêtes LINQ sur le jeu précédent. Toutes ces opérations sont effectuées de façon retardée. Cela signifie également qu'elles sont effectuées à chaque fois que la séquence est demandée. À la 52e itération, vous aurez régénéré le jeu d'origine de nombreuses fois. Nous allons écrire dans un journal pour illustrer ce comportement. Ensuite, vous résoudrez le problème.

Dans votre fichier `Extensions.cs`, tapez ou copiez la méthode d'extension ci-dessous. Elle crée un fichier nommé `debug.log` au sein de votre répertoire de projet et enregistre la requête en cours d'exécution dans le fichier journal. Elle peut être ajoutée à une requête pour marquer le fait qu'elle s'est exécutée.

C#

```
public static IEnumerable<T> LogQuery<T>
    (this IEnumerable<T> sequence, string tag)
{
    // File.AppendText creates a new file if the file doesn't exist.
    using (var writer = File.AppendText("debug.log"))
    {
        writer.WriteLine($"Executing Query {tag}");
    }

    return sequence;
}
```

Vous verrez une ligne ondulée rouge sous `File`, ce qui signifie qu'elle n'existe pas. Elle ne sera pas compilée, puisque le compilateur ne sait pas ce qu'est `File`. Pour résoudre ce problème, veillez à ajouter la ligne de code suivante sous la première ligne dans `Extensions.cs` :

C#

```
using System.IO;
```

Cela devrait résoudre le problème et faire disparaître l'erreur rouge.

Instrumentez ensuite la définition de chaque requête avec un message de journalisation :

C#

```
// Program.cs
public static void Main(string[] args)
{
    var startingDeck = (from s in Suits().LogQuery("Suit Generation")
```

```
                        from r in Ranks().LogQuery("Rank Generation")
                        select new { Suit = s, Rank = r
}).LogQuery("Starting Deck");

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    Console.WriteLine();
    var times = 0;
    var shuffle = startingDeck;

    do
    {
        // Out shuffle
        /*
        shuffle = shuffle.Take(26)
            .LogQuery("Top Half")
            .InterleaveSequenceWith(shuffle.Skip(26)
            .LogQuery("Bottom Half"))
            .LogQuery("Shuffle");
        */

        // In shuffle
        shuffle = shuffle.Skip(26).LogQuery("Bottom Half")
                .InterleaveSequenceWith(shuffle.Take(26).LogQuery("Top
Half"))
                .LogQuery("Shuffle");

        foreach (var c in shuffle)
        {
            Console.WriteLine(c);
        }

        times++;
        Console.WriteLine(times);
    } while (!startingDeck.SequenceEquals(shuffle));

    Console.WriteLine(times);
}
```

Remarquez que vous n'écrivez pas dans le journal à chaque fois que vous accédez à une requête, mais seulement lors de la création de la requête d'origine. Le programme met toujours longtemps à s'exécuter, mais vous pouvez maintenant voir pourquoi. Si vous perdez patience au cours de l'exécution du mélange intérieur avec journalisation, revenez au mélange extérieur. Vous verrez quand même les effets de l'évaluation paresseuse. Sur une itération, elle exécute 2 592 requêtes, génération de toutes les valeurs et couleurs comprise.

Vous pouvez améliorer les performances du code en réduisant le nombre d'exécutions effectuées. L'un des correctifs les plus simples consiste à mettre en *cache* les résultats de la requête LINQ d'origine qui construit le jeu de cartes. Actuellement, les requêtes sont réexécutées chaque fois que la boucle do-while effectue une itération, construisant et battant de nouveau le jeu de cartes à chaque fois. Pour mettre en cache le jeu de cartes, vous pouvez utiliser les méthodes LINQ ToArray et ToList ; ajoutées aux requêtes, elles effectuent les actions voulues, mais en stockant respectivement les résultats dans un tableau ou dans une liste. Ajoutez la méthode LINQ ToArray aux deux requêtes et réexécutez le programme :

C#

```
public static void Main(string[] args)
{
    IEnumerable<Suit>? suits = Suits();
    IEnumerable<Rank>? ranks = Ranks();

    if ((suits is null) || (ranks is null))
        return;

    var startingDeck = (from s in suits.LogQuery("Suit Generation")
                        from r in ranks.LogQuery("Value Generation")
                        select new { Suit = s, Rank = r })
                        .LogQuery("Starting Deck")
                        .ToArray();

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    Console.WriteLine();

    var times = 0;
    var shuffle = startingDeck;

    do
    {
        /*
        shuffle = shuffle.Take(26)
            .LogQuery("Top Half")
            .InterleaveSequenceWith(shuffle.Skip(26).LogQuery("Bottom
Half"))
            .LogQuery("Shuffle")
            .ToArray();
        */

        shuffle = shuffle.Skip(26)
            .LogQuery("Bottom Half")
            .InterleaveSequenceWith(shuffle.Take(26).LogQuery("Top Half"))
            .LogQuery("Shuffle")
```

```
            .ToArray();

        foreach (var c in shuffle)
        {
            Console.WriteLine(c);
        }

        times++;
        Console.WriteLine(times);
    } while (!startingDeck.SequenceEquals(shuffle));

    Console.WriteLine(times);
}
```

Le mélange extérieur est descendu à 30 requêtes. Si vous repassez au mélange intérieur, vous constaterez des améliorations similaires : il ne comporte plus que 162 requêtes.

Sachez que cet exemple est **conçu** pour mettre en évidence les cas d'usage où l'évaluation paresseuse cause des problèmes de performances. S'il est important de voir son impact sur les performances du code, il faut également comprendre que toutes les requêtes ne doivent pas s'exécuter de manière stricte. La baisse de performances que l'on constate sans ToArray est due au fait que chaque nouvelle disposition du jeu de cartes est construite à partir de la configuration précédente. Avec l'évaluation paresseuse, chaque nouvelle configuration du jeu est générée à partir du jeu d'origine, y compris l'exécution du code qui a construit `startingDeck`, ce qui entraîne une grande quantité de travail supplémentaire.

Dans la pratique, certains algorithmes fonctionnent bien avec l'évaluation stricte, d'autres avec l'évaluation paresseuse. Au quotidien, l'évaluation paresseuse est en général un meilleur choix lorsque la source de données est un processus distinct, comme un moteur de base de données. L'évaluation paresseuse permet dans ce cas d'exécuter des requêtes plus complexes avec un seul aller-retour entre le processus de base de données et le reste du code. LINQ peut s'adapter tant à l'évaluation paresseuse qu'à l'évaluation stricte. Par conséquent, examinez vos processus et choisissez le type d'évaluation qui vous offre les meilleures performances.

Conclusion

Dans ce projet, nous avons vu comment :

- utiliser des requêtes LINQ pour agréger des données en une séquence explicite ;
- écrire des méthodes d'extension pour ajouter des fonctionnalités personnalisées aux requêtes LINQ ;

- localiser les zones du code où les requêtes LINQ risquent de poser des problèmes de performances, comme une dégradation de la vitesse ;
- utiliser l'évaluation paresseuse et l'évaluation stricte dans des requêtes LINQ, avec les implications que cela comporte sur les performances de requête.

En dehors de LINQ, vous avez appris une technique de tour de cartes utilisée par les magiciens. Les magiciens utilisent le mélange faro pour pouvoir contrôler le déplacement de chaque carte dans le jeu. Maintenant que vous le savez, gardez le secret !

Pour plus d'informations sur LINQ, voir :

- LINQ (Language-Integrated Query)
- Introduction à LINQ
- Opérations de requête LINQ de base (C#)
- Transformations de données avec LINQ (C#)
- Syntaxe de requête et syntaxe de méthode dans LINQ (C#)
- Fonctionnalités C# qui prennent en charge LINQ

Définir et lire des attributs personnalisés

Article • 20/03/2023

Les attributs fournissent un moyen permettant d'associer des informations au code de manière déclarative. Ils peuvent également fournir un élément réutilisable qui peut être appliqué à diverses cibles. Prenons l'exemple d'ObsoleteAttribute. Vous pouvez l'appliquer aux classes, structures, méthodes, constructeurs et bien plus encore. Il *déclare* que l'élément est obsolète. Il revient ensuite au compilateur C# de rechercher cet attribut et d'effectuer des actions en réponse.

Dans ce tutoriel, vous découvrez comment ajouter des attributs à votre code, comment créer et utiliser vos propres attributs et comment utiliser des attributs qui sont intégrés à .NET.

Prérequis

Vous devez configurer votre ordinateur pour exécuter .NET. Vous trouverez les instructions d'installation dans la page Téléchargements .NET. Vous pouvez exécuter cette application sur Windows, Ubuntu Linux, macOS ou dans un conteneur Docker. Vous devez installer l'éditeur de code de votre choix. Les descriptions suivantes utilisent Visual Studio Code, un éditeur multiplateforme open source. Cependant, vous pouvez utiliser les outils avec lesquels vous êtes le plus à l'aise.

Créer l'application

Maintenant que vous avez installé tous les outils, créez une application console .NET. Pour utiliser le générateur de ligne de commande, exécutez la commande suivante dans votre interpréteur de commandes préféré :

CLI .NET

```
dotnet new console
```

Cette commande crée des fichiers projet .NET simples. Exécutez `dotnet restore` pour restaurer les dépendances nécessaires à la compilation de ce projet.

Vous n'avez pas besoin d'exécuter dotnet restore, car il est exécuté implicitement par toutes les commandes qui nécessitent une restauration pour se produire, comme `dotnet new`, `dotnet build`, `dotnet run`, `dotnet test`, `dotnet publish` et `dotnet pack`. Pour désactiver la restauration implicite, utilisez l'option `--no-restore` .

La commande `dotnet restore` est toujours utile dans certains scénarios où la restauration explicite est logique, comme les builds d'intégration continue dans Azure DevOps Services ou dans les systèmes de génération qui doivent contrôler explicitement le moment où la restauration se produit.

Pour plus d'informations sur la gestion des flux NuGet, consultez la documentation de dotnet restore.

Pour exécuter le programme, utilisez `dotnet run`. Vous devriez voir le résultat dans la « Hello, World » dans la console.

Ajouter des attributs au code

En C#, les attributs sont des classes qui héritent de la classe de base `Attribute`. Toute classe qui hérite de `Attribute` peut être utilisée comme une sorte de « balise » sur les autres éléments de code. Par exemple, il existe un attribut appelé `ObsoleteAttribute`. Cet attribut signale que le code est obsolète et ne doit plus être utilisé. Vous pouvez placer cet attribut sur une classe, par exemple, en utilisant des crochets.

C#

```
[Obsolete]
public class MyClass
{
}
```

Bien que classe soit nommée `ObsoleteAttribute`, il vous suffit d'utiliser `[Obsolete]` dans le code. La majorité du code C# suit cette convention. Vous pouvez utiliser le nom complet `[ObsoleteAttribute]` si vous le souhaitez.

Lorsque vous marquez une classe comme étant obsolète, il est judicieux de fournir des informations indiquant *pourquoi* elle est obsolète, et/ou *quelle classe* utiliser à la place. Vous incluez un paramètre de chaîne à l'attribut Obsolete pour fournir cette explication.

C#

```
[Obsolete("ThisClass is obsolete. Use ThisClass2 instead.")]
public class ThisClass
{
}
```

La chaîne est passée comme argument à un constructeur `ObsoleteAttribute`, comme si vous écriviez `var attr = new ObsoleteAttribute("some string")`.

Les paramètres à un constructeur d'attribut sont limités aux types simples/littéraux : `bool, int, double, string, Type, enums, etc` et les tableaux de ces types. Vous ne pouvez pas utiliser une expression ou une variable. Vous êtes libre d'utiliser des paramètres positionnels ou nommés.

Créer votre propre attribut

Vous créez un attribut en définissant une nouvelle classe qui hérite de la classe de base `Attribute`.

C#

```
public class MySpecialAttribute : Attribute
{
}
```

Avec le code précédent, vous pouvez utiliser `[MySpecial]` (ou `[MySpecialAttribute]`) comme attribut ailleurs dans la base de code.

C#

```
[MySpecial]
public class SomeOtherClass
{
}
```

Les attributs dans la bibliothèque de classes de base .NET, comme `ObsoleteAttribute`, déclenchent certains comportements au sein du compilateur. Toutefois, un attribut que vous créez agit uniquement comme métadonnées et ne produit pas de code dans la classe d'attributs en cours d'exécution. Il vous appartient d'agir sur ces métadonnées ailleurs dans votre code.

Il y a ici un « piège » à éviter. Comme mentionné précédemment, seuls certains types peuvent être passés comme arguments lors de l'utilisation d'attributs. Toutefois, quand vous créez un type d'attribut, le compilateur C# ne vous empêche pas de créer ces paramètres. Dans l'exemple suivant, vous avez créé un attribut avec un constructeur qui se compile correctement.

C#

```
public class GotchaAttribute : Attribute
{
    public GotchaAttribute(Foo myClass, string str)
    {
```

```
    }
}
```

Toutefois, vous ne pouvez pas utiliser ce constructeur avec une syntaxe d'attribut.

C#

```
[Gotcha(new Foo(), "test")] // does not compile
public class AttributeFail
{
}
```

Le code précédent provoque une erreur du compilateur du type : `Attribute constructor parameter 'myClass' has type 'Foo', which is not a valid attribute parameter type`

Comment limiter l'utilisation d'attributs

Les attributs peuvent être utilisés sur les « cibles » suivantes. Les exemples précédents les montrent sur des classes, mais ils peuvent également être utilisés sur :

- Assembly
- Classe
- Constructeur
- Délégué
- Énumération
- Événement
- Champ
- GenericParameter
- Interface
- Méthode
- Module
- Paramètre
- Propriété
- ReturnValue
- Structure

Quand vous créez une classe d'attributs, par défaut, C# vous permet d'utiliser cet attribut sur l'une des cibles d'attribut possibles. Si vous souhaitez limiter votre attribut à certaines cibles, vous pouvez le faire à l'aide de `AttributeUsageAttribute` sur votre classe d'attributs. C'est exact, un attribut sur un attribut !

C#

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
public class MyAttributeForClassAndStructOnly : Attribute
{
}
```

Si vous tentez de placer l'attribut ci-dessus sur quelque chose qui n'est ni une classe ni un struct, vous obtenez une erreur du compilateur du type : `Attribute 'MyAttributeForClassAndStructOnly' is not valid on this declaration type. It is only valid on 'class, struct' declarations`

C#

```
public class Foo
{
    // if the below attribute was uncommented, it would cause a compiler
error
    // [MyAttributeForClassAndStructOnly]
    public Foo()
    { }
}
```

Guide d'utilisation des attributs associés à un élément de code

Les attributs agissent comme métadonnées. Sans une certaine force extérieure, vous ne les voyez pas faire grand-chose.

Pour rechercher des attributs et agir dessus, la réflexion est nécessaire. La réflexion vous permet d'écrire du code en C# qui examine un autre code. Par exemple, vous pouvez utiliser la réflexion pour obtenir des informations sur une classe (ajoutez `using System.Reflection;` au début de votre code) :

C#

```
TypeInfo typeInfo = typeof(MyClass).GetTypeInfo();
Console.WriteLine("The assembly qualified name of MyClass is " +
typeInfo.AssemblyQualifiedName);
```

Vous obtenez une sortie qui ressemble à ceci : `The assembly qualified name of MyClass is ConsoleApplication.MyClass, attributes, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null`

Une fois que vous avez un objet `TypeInfo` (ou un objet `MemberInfo`, `FieldInfo` ou autre), vous pouvez utiliser la méthode `GetCustomAttributes`. Cette méthode retourne une collection d'objets `Attribute`. Vous pouvez également utiliser `GetCustomAttribute` et spécifier un type d'attribut.

Voici un exemple d'utilisation de `GetCustomAttributes` sur une instance de `MemberInfo` pour `MyClass` (avec un attribut `[Obsolete]` associé, comme vu précédemment).

C#

```
var attrs = typeInfo.GetCustomAttributes();
foreach(var attr in attrs)
    Console.WriteLine("Attribute on MyClass: " + attr.GetType().Name);
```

Qui imprime sur la console : `Attribute on MyClass: ObsoleteAttribute`. Essayez d'ajouter d'autres attributs à `MyClass`.

Il est important de noter que ces objets `Attribute` sont instanciés de manière différée. Autrement dit, ils ne sont pas instanciés tant que vous n'utilisez pas `GetCustomAttribute` ou `GetCustomAttributes`. Ils sont également instanciés chaque fois. Le fait d'appeler `GetCustomAttributes` deux fois de suite retourne deux instances différentes de `ObsoleteAttribute`.

Attributs communs dans le runtime

Les attributs sont utilisés par de nombreux outils et frameworks. NUnit utilise des attributs tels que `[Test]` et `[TestFixture]`, qui sont utilisés par le Test Runner de NUnit. ASP.NET MVC utilise des attributs tels que `[Authorize]` et fournit un framework de filtres d'action pour traiter les problèmes transversaux sur les actions MVC. PostSharp utilise la syntaxe d'attribut pour permettre la programmation orientée aspect en C#.

Voici quelques attributs importants générés dans les bibliothèques de classe de base de .NET Core :

- `[Obsolete]`. Celui-ci a été utilisé dans les exemples ci-dessus, et il se trouve dans l'espace de noms `System`. Il est utile de fournir des informations déclaratives sur une base de code changeante. Un message peut être fourni sous la forme d'une chaîne et un autre paramètre booléen peut servir à passer d'un avertissement du compilateur à une erreur du compilateur.
- `[Conditional]`. Cet attribut se trouve dans l'espace de noms `System.Diagnostics`. Cet attribut peut être appliqué aux méthodes (ou classes d'attributs). Vous devez

transmettre une chaîne au constructeur. Si cette chaîne ne correspond pas à une directive `#define`, le compilateur C# supprime les appels à cette méthode (mais pas la méthode proprement dite). En général, vous utilisez cette technique pour le débogage (diagnostic).

- `[CallerMemberName]`. Cet attribut peut être utilisé sur les paramètres et se trouve dans l'espace de noms `System.Runtime.CompilerServices`. `CallerMemberName` est un attribut qui permet d'injecter le nom de la méthode qui appelle une autre méthode. C'est un moyen d'éliminer les « chaînes magiques » lors de l'implémentation d'INotifyPropertyChanged dans divers frameworks d'interface utilisateur. Par exemple :

C#

```
public class MyUIClass : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler? PropertyChanged;

    public void RaisePropertyChanged([CallerMemberName] string propertyName
= default!)
    {
        PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(propertyName));
    }

    private string? _name;
    public string? Name
    {
        get { return _name;}
        set
        {
            if (value != _name)
            {
                _name = value;
                RaisePropertyChanged();   // notice that "Name" is not
needed here explicitly
            }
        }
    }
}
```

Dans le code ci-dessus, il est inutile d'avoir une chaîne littérale `"Name"`. L'utilisation de `CallerMemberName` permet d'éviter les bogues liés aux fautes de frappe et facilite également la refactorisation/le renommage. Si les attributs offrent à C# la puissance déclarative, ils constituent une forme de code de type métadonnées et n'agissent pas par eux-mêmes.

Types références Nullables

Article • 09/05/2023

Dans un contexte oubliant la valeur Null, tous les types de référence pouvaient accepter la valeur Null. Les *types de référence pouvant accepter la valeur Null* font référence à un groupe de fonctionnalités activées dans un contexte pouvant accepter la valeur Null qui réduisent la probabilité que votre code provoque la levée System.NullReferenceExceptiondu runtime . Les *types de référence pouvant accepter la valeur Null* incluent trois fonctionnalités qui vous permettent d'éviter ces exceptions, notamment la possibilité de marquer explicitement un type de référence comme *pouvant accepter la valeur Null* :

- Analyse de flux statique améliorée qui détermine si une variable peut être `null` avant de la déréférencer.
- Attributs qui annotent les API afin que l'analyse de flux détermine *l'état Null*.
- Annotations des variables que les développeurs utilisent pour déclarer explicitement *l'état Null* prévu pour une variable.

L'analyse de l'état Null et les annotations variables sont désactivées par défaut pour les projets existants, ce qui signifie que tous les types de référence continuent de pouvoir accepter la valeur Null. À compter de .NET 6, elles sont activées par défaut pour les *nouveaux* projets. Pour plus d'informations sur l'activation de ces fonctionnalités en déclarant un *contexte d'annotation pouvant accepter la valeur Null*, consultez Contextes pouvant accepter la valeur Null.

Le reste de cet article décrit comment ces trois zones de fonctionnalités fonctionnent pour produire des avertissements lorsque votre code peut **déréférencer** une `null` valeur. Le déréférencement d'une variable signifie accéder à l'un de ses membres à l'aide de l'opérateur `.` (point), comme illustré dans l'exemple suivant :

C#

```
string message = "Hello, World!";
int length = message.Length; // dereferencing "message"
```

Lorsque vous déréférencez une variable dont la valeur est `null`, le runtime lève une System.NullReferenceException.

Vous pouvez également explorer ces concepts dans notre module Learn sur la sécurité des valeurs Null en C#.

Analyse de l'état Null

L'analyse de l'état Null suit *l'état Null* des références. Cette analyse statique émet des avertissements lorsque votre code peut déréférencer `null`. Vous pouvez traiter ces avertissements pour réduire les incidences lorsque le runtime lève une System.NullReferenceException. Le compilateur utilise l'analyse statique pour déterminer l'*état Null* d'une variable. Une variable est soit *not-null*, soit *maybe-null*. Le compilateur détermine qu'une variable est *not-null* de deux manières :

1. Une valeur connue pour être *not-null* a été attribuée à la variable.
2. La variable a été vérifiée par rapport à `null` et n'a pas été modifiée depuis cette vérification.

Toute variable que le compilateur n'a pas déterminée comme *not-null* est considérée comme *maybe-null*. L'analyse fournit des avertissements dans les situations où vous pouvez déréférencer accidentellement une valeur `null`. Le compilateur produit des avertissements basés sur *l'état Null*.

- Lorsqu'une variable est*not-null*, cette variable peut être déréférencée en toute sécurité.
- Lorsqu'une variable est *maybe-null*, cette variable doit être vérifiée pour s'assurer qu'elle n'est pas `null` avant de la déréférencer.

Prenons l'exemple suivant :

C#

```
string message = null;

// warning: dereference null.
Console.WriteLine($"The length of the message is {message.Length}");

var originalMessage = message;
message = "Hello, World!";

// No warning. Analysis determined "message" is not null.
Console.WriteLine($"The length of the message is {message.Length}");

// warning!
Console.WriteLine(originalMessage.Length);
```

Dans l'exemple précédent, le compilateur détermine que `message` est *maybe-null* lorsque le premier message est imprimé. Il n'y a pas d'avertissement pour le deuxième message. La dernière ligne de code produit un avertissement, car `originalMessage` peut

être null. L'exemple suivant montre une utilisation plus pratique pour parcourir une arborescence de nœuds jusqu'à la racine, en traitant chaque nœud pendant le parcours :

```
C#
```

```
void FindRoot(Node node, Action<Node> processNode)
{
    for (var current = node; current != null; current = current.Parent)
    {
        processNode(current);
    }
}
```

Le code précédent ne génère aucun avertissement pour le déréférencement de la variable `current`. L'analyse statique détermine que `current` n'est jamais déréférencé lorsqu'il est *maybe-null*. La variable `current` est vérifiée par rapport à `null` avant que la variable `current.Parent` ne soit accédée et avant de passer `current` à l'action `ProcessNode`. Les exemples précédents montrent comment le compilateur détermine *l'état Null* des variables locales lorsqu'elles sont initialisées, affectées ou comparées à `null`.

L'analyse de l'état Null ne trace pas dans les méthodes appelées. Par conséquent, les champs initialisés dans une méthode d'assistance traditionnelle appelée par les constructeurs génèrent un avertissement avec le modèle suivant :

> Le *nom* de la propriété non-nullable doit contenir une valeur non nulle lors de la sortie du constructeur.

Vous pouvez traiter ces avertissements à l'aide de l'une des deux manières suivantes : *chaînage du constructeur* ou *attributs pouvant accepter la valeur Null* sur la méthode d'assistance. Le code suivant montre un exemple de chaque. La classe `Person` utilise un constructeur commun appelé par tous les autres constructeurs. La classe `Student` a une méthode d'assistance annotée avec l'attribut System.Diagnostics.CodeAnalysis.MemberNotNullAttribute :

```
C#
```

```
using System.Diagnostics.CodeAnalysis;

public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
```

```
    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

    public Person() : this("John", "Doe") { }
}

public class Student : Person
{
    public string Major { get; set; }

    public Student(string firstName, string lastName, string major)
        : base(firstName, lastName)
    {
        SetMajor(major);
    }

    public Student(string firstName, string lastName) :
        base(firstName, lastName)
    {
        SetMajor();
    }

    public Student()
    {
        SetMajor();
    }

    [MemberNotNull(nameof(Major))]
    private void SetMajor(string? major = default)
    {
        Major = major ?? "Undeclared";
    }
}
```

> ⓘ **Notes**
>
> Un certain nombre d'améliorations apportées à l'assignation définie et à l'analyse de l'état Null ont été ajoutées dans C# 10. Lorsque vous effectuez une mise à niveau vers C# 10, vous pouvez trouver moins d'avertissements pouvant accepter la valeur Null qui sont des faux positifs. Vous pouvez en savoir plus sur les améliorations apportées à la **spécification des fonctionnalités pour les améliorations d'assignation définitives**.

L'analyse de l'état des variables pouvant accepter la valeur Null et les avertissements générés par le compilateur vous aident à éviter les erreurs de programme en déréférencement `null`. L'article sur la résolution des avertissements sur la valeur Null

fournit des techniques de correction des avertissements que vous verrez probablement dans votre code.

Attributs sur les signatures d'API

L'analyse de l'état Null a besoin d'indications de la part des développeurs pour comprendre la sémantique des API. Certaines API fournissent des vérifications de la valeur Null et doivent modifier *l'état Null* d'une variable de *maybe-null* à *not-null*. D'autres API retournent des expressions qui sont*not-null* ou *maybe-null* en fonction de *l'état Null* des arguments d'entrée. Prenons par exemple le code suivant, qui affiche un message :

C#

```
public void PrintMessage(string message)
{
    if (!string.IsNullOrWhiteSpace(message))
    {
        Console.WriteLine($"{DateTime.Now}: {message}");
    }
}
```

En fonction de l'inspection, tout développeur considérerait ce code comme sûr et ne devrait pas générer d'avertissements. Le compilateur ne sait pas que `IsNullOrWhiteSpace` fournit une vérification de la valeur Null. Lorsque `IsNullOrWhitespace` retourne `false`, *l'état Null* de la chaîne est *non Null*. Lorsque `IsNullOrWhitespace` retourne `true`, *l'état Null* n'est pas modifié. Dans l'exemple précédent, la signature inclut NotNullWhen pour indiquer l'état Null de `message` :

C#

```
public static bool IsNullOrWhiteSpace([NotNullWhen(false)] string message);
```

Les attributs fournissent des informations détaillées sur l'état Null des arguments, des valeurs renvoyées et des membres de l'instance d'objet utilisée pour appeler un membre. Vous trouverez plus d'informations sur chaque attribut dans l'article de référence de langage sur les attributs de référence pouvant accepter la valeur Null. Les API de runtime .NET ont toutes été annotées dans .NET 5. Vous améliorez l'analyse statique en annotant vos API pour fournir des informations sémantiques sur *l'état Null* des arguments et des valeurs renvoyées.

Annotations de variables pouvant accepter la valeur Null

L'analyse de l'*état Null* fournit une analyse robuste pour la plupart des variables. Le compilateur a besoin de plus d'informations de votre part pour les variables membres. Le compilateur ne peut pas faire d'hypothèses sur l'ordre auquel les membres publics sont accédés. Tout membre public est accessible dans n'importe quel ordre. L'un des constructeurs accessibles peut être utilisé pour initialiser l'objet. Si un champ membre peut être défini sur `null`, le compilateur doit supposer que son *état Null* est *maybe-null* au début de chaque méthode.

Vous utilisez des annotations qui peuvent déclarer si une variable est un **type de référence pouvant accepter la valeur Null** ou un **type de référence non nullable**. Ces annotations constituent des instructions importantes concernant *l'état Null* pour les variables :

- **Une référence n'est pas censée être null**. L'état par défaut d'une variable de référence non nullable est *not-null*. Le compilateur applique des règles qui garantissent que vous pouvez sans problème déréférencer ces variables sans avoir d'abord à vérifier qu'elles ne sont pas null :
  - La variable doit être initialisée avec une valeur non null.
  - La variable ne peut jamais se voir attribuer la valeur `null`. Le compilateur émet un avertissement lorsque le code affecte une expression *maybe-null* à une variable qui ne doit pas être null.
- **Une référence peut être null**. L'état par défaut d'une variable de référence pouvant accepter la valeur Null est *not-null*. Le compilateur applique des règles pour vérifier que vous avez correctement vérifié une référence `null` :
  - La variable peut être déréférencée seulement si le compilateur peut garantir que la valeur n'est pas `null`.
  - Ces variables peuvent être initialisées avec la valeur `null` par défaut et peuvent se voir attribuer la valeur `null` dans un autre code.
  - Le compilateur n'émet pas d'avertissements lorsque le code affecte une expression *maybe-null* à une variable qui peut être null.

Toute variable de référence qui n'est pas censée être `null` a un *état Null* de *not-null*. Toute variable de référence qui peut être `null` initialement a *l'état Nullmaybe-null*.

Un **type de référence nullable** est inidqué avec la même syntaxe que celle des types valeur nullable : un `?` est ajouté au type de la variable. Par exemple, la déclaration de variable suivante représente une variable de chaîne nullable, `name` :

```
```

C#

```
string? name;
```

Toute variable où `?` n'est pas ajouté au nom de type est un **type référence non nullable**. Cela comprend toutes les variables de type référence du code existant si vous avez activé cette fonctionnalité. Toutefois, toutes les variables locales implicitement typées (déclarées à l'aide de `var`) sont des **types de référence pouvant accepter la valeur Null**. Comme l'ont montré les sections précédentes, l'analyse statique détermine *l'état Null* des variables locales pour déterminer si elles sont *maybe-null*.

Parfois, vous devez remplacer un avertissement quand vous savez qu'une variable n'est pas null, mais que le compilateur détermine que son *état Null* est *maybe-null*. Vous utilisez l'opérateur null-forgiving `!` à la suite d'un nom de variable pour forcer *l'état Null* à être *not-null*. Par exemple, si vous savez que la variable `name` n'est pas `null` alors que le compilateur génère un avertissement, vous pouvez écrire le code suivant pour remplacer l'analyse du compilateur :

C#

```
name!.Length;
```

Les types référence et les types valeur pouvant accepter la valeur Null fournissent un concept sémantique similaire : une variable peut représenter une valeur ou un objet, ou cette variable peut être `null`. Toutefois, les types référence et les types valeur pouvant accepter la valeur Null sont implémentés différemment : les types valeur pouvant accepter la valeur Null sont implémentés à l'aide de System.Nullable<T>, et les types référence pouvant accepter la valeur Null sont implémentés par les attributs lus par le compilateur. Par exemple, `string?` et `string` sont tous deux représentés par le même type System.String. Toutefois, `int?` et `int` sont représentés par `System.Nullable<System.Int32>` et System.Int32, respectivement.

Les types de référence pouvant accepter la valeur Null sont une fonctionnalité de compilation. Cela signifie qu'il est possible pour les appelants d'ignorer les avertissements et d'utiliser `null` intentionnellement comme argument pour une méthode qui attend une référence non nullable. Les auteurs de bibliothèque doivent inclure des vérifications du runtime relatives aux valeurs d'argument null. ArgumentNullException.ThrowIfNull est l'option préférée pour vérifier un paramètre par rapport à la valeur Null au moment de l'exécution.

ⓘ Important

L'activation des annotations nullables peut modifier la manière dont Entity Framework Core détermine si un membre de données est requis. Pour plus d'informations, consultez l'article **Principes de base d'Entity Framework Core : Utilisation des types référence nullables**.

Génériques

Les génériques nécessitent des règles détaillées pour gérer `T?` pour tout paramètre de type `T`. Les règles sont nécessairement détaillées en raison de l'historique et de l'implémentation différente d'un type valeur et d'un type référence pouvant accepter la valeur Null. Les types valeur pouvant accepter la valeur Null sont implémentés à l'aide du struct System.Nullable<T>. Les types référence pouvant accepter la valeur Null sont implémentés en tant qu'annotations de type qui fournissent des règles sémantiques au compilateur.

- Si l'argument de type pour `T` est un type référence, `T?` fait référence au type de référence pouvant accepter la valeur Null correspondant. Par exemple, si `T` est un `string`, alors `T?` est un `string?`.
- Si l'argument de type pour `T` est un type valeur, `T?` fait référence au même type valeur, `T`. Par exemple, si `T` est un `int`, `T?` est également un `int`.
- Si l'argument de type pour `T` est un type référence pouvant accepter la valeur Null, `T?` fait référence au type de référence pouvant accepter la valeur Null correspondant. Par exemple, si `T` est un `string?`, alors `T?` est aussi un `string?`.
- Si l'argument de type pour `T` est un type valeur pouvant accepter la valeur Null, `T?` fait référence au même type valeur pouvant accepter la valeur Null. Par exemple, si `T` est un `int?`, alors `T?` est aussi un `int?`.

Pour les valeurs renvoyées, `T?` équivaut à `[MaybeNull]T` ; pour les valeurs d'argument, `T?` équivaut à `[AllowNull]T`. Pour plus d'informations, consultez l'article Attributs pour l'analyse de l'état Null dans la référence de langage.

Vous pouvez spécifier un comportement différent à l'aide de contraintes :

- La contrainte `class` signifie que `T` doit être un type référence non nullable (par exemple `string`). Le compilateur génère un avertissement si vous utilisez un type référence pouvant accepter la valeur Null, par exemple `string?` pour `T`.
- La contrainte `class?`signifie que `T` doit être un type référence, non nullable (`string`) ou un type référence pouvant accepter la valeur nulle (par exemple `string?`). Lorsque le paramètre de type est un type référence pouvant accepter la

valeur Nul, tel que `string?`, une expression de `T?` fait référence à ce même type référence pouvant accepter la valeur Null, comme `string?`.

- La contrainte `notnull` signifie que `T` doit être un type référence non nullable ou un type valeur non nullable. Si vous utilisez un type référence ou un type valeur pouvant accepter la valeur Null pour le paramètre de type, le compilateur génère un avertissement. En outre, quand `T` est un type valeur, la valeur renvoyée est ce type valeur, et non le type de valeur pouvant accepter la valeur Null correspondant.

Ces contraintes permettent de fournir au compilateur plus d'informations sur la façon dont `T` sera utilisé. Cela est utile lorsque les développeurs choisissent le type pour `T`et cela fournit une meilleure analyse de *l'état Null* lorsqu'une instance du type générique est utilisée.

Contextes nullables

Les nouvelles fonctionnalités qui protègent contre la levée d'une System.NullReferenceException peut être perturbante lorsqu'elles sont activées dans un codebase existant :

- Toutes les variables de référence explicitement typées sont interprétées comme des types référence non nullables.
- La signification de la contrainte `class` dans les génériques a changé pour désigner un type référence non nullable.
- De nouveaux avertissements sont générés en raison de ces nouvelles règles.

Vous devez choisir explicitement d'utiliser ces fonctionnalités dans vos projets existants. Cela fournit un chemin de migration et préserve la compatibilité descendante. Les contextes nullables permettent de contrôler précisément comment le compilateur interprète les variables de type référence. Le **contexte d'annotation nullable** détermine le comportement du compilateur. Il existe quatre valeurs pour le **contexte d'annotation nullable** :

- *disable* : le code *oublie la valeur Null*.
  - Les avertissements de valeur Null sont désactivés.
  - Toutes les variables de type référence sont des types référence pouvant accepter la valeur Null.
  - Vous ne pouvez pas déclarer une variable en tant que type référence pouvant accepter la valeur Null à l'aide du suffixe `?` sur le type.
  - Vous pouvez utiliser l'opérateur null forgiving, `!`, mais il n'a aucun effet.

- *enable* : le compilateur active toutes les analyses de référence Null et toutes les fonctionnalités de langage.
  - Tous les nouveaux avertissements nullables sont activés.
  - Vous pouvez utiliser le suffixe `?` pour déclarer un type référence pouvant accepter la valeur Null.
  - Toutes les autres variables de type référence sont des types référence non-nullable.
  - L'opérateur null forgiving supprime les avertissements pour une affectation possible à `null`.
- *warnings* : le compilateur effectue toutes les analyses de valeurs Null et émet des avertissements quand le code peut déréférencer `null`.
  - Tous les nouveaux avertissements nullables sont activés.
  - L'utilisation du suffixe `?` pour déclarer un type référence pouvant accepter la valeur Null génère un avertissement.
  - Toutes les variables de type référence sont autorisées à être null. Toutefois, les membres ont *l'état Null* de *not-null* à l'ouverture de toutes les méthodes, sauf s'ils sont déclarés avec le suffixe `?`.
  - Vous pouvez utiliser l'opérateur null forgiving, `!`.
- *annotations* : le compilateur n'effectue pas d'analyse de la valeur Null ou n'émet pas d'avertissements lorsque le code peut déréférencer `null`.
  - Les nouveaux avertissements de valeur Null sont désactivés.
  - Vous pouvez utiliser le suffixe `?` pour déclarer un type référence pouvant accepter la valeur Null.
  - Toutes les autres variables de type référence sont des types référence non-nullable.
  - Vous pouvez utiliser l'opérateur null forgiving, `!`, mais il n'a aucun effet.

Le contexte d'annotation nullable et le contexte d'avertissement nullable peuvent être définis pour un projet avec l'élément <Nullable> de votre fichier *.csproj*. Cet élément configure la manière dont le compilateur interprète la nullabilité des types et quels avertissements sont générés. Le tableau suivant présente les valeurs autorisées et résume les contextes qu'elles spécifient.

Context	**Avertissements de déréférencement**	**Avertissements d'assignation**	**Types référence**	**Suffixe `?`**	**L'opérateur `!`**
`disable`	Désactivé	Désactivé	Tous acceptent la valeur Null	Ne peut pas être utilisé	N'a aucun effet

Context	Avertissements de déréférencement	Avertissements d'assignation	Types référence	Suffixe `?`	L'opérateur `!`
`enable`	activé	activé	Non nullable, sauf déclaré avec `?`	Déclare le type pouvant accepter la valeur Null	Supprime les avertissements en cas d'affectation `null` possible
`warnings`	activé	Non applicable	Tous acceptent la valeur Null, mais les membres sont considérés *not-null* au moment de l'ouverture des méthodes	Génère un avertissement	Supprime les avertissements en cas d'affectation `null` possible
`annotations`	Désactivé	Désactivé	Non nullable, sauf déclaré avec `?`	Déclare le type pouvant accepter la valeur Null	N'a aucun effet

Les variables de type référence dans le code compilé dans un contexte *désactivé* sont *nullable-oblivious*. Vous pouvez assigner un littéral `null` ou une variable *maybe-null* à une variable qui peut avoir la valeur *nullable-obliviou*. Toutefois, l'état par défaut d'une variable *nullable-oblivious* est*not-null*.

Vous pouvez choisir le paramètre le mieux adapté à votre projet :

- Choisissez *disable* pour les projets hérités que vous ne souhaitez pas mettre à jour en fonction des diagnostics ou de nouvelles fonctionnalités.
- Choisissez *warnings* pour déterminer où votre code peut lever System.NullReferenceException. Vous pouvez résoudre ces avertissements avant de modifier le code pour activer les types référence non nullables.
- Choisissez *annotations* pour exprimer votre intention de conception avant d'activer les avertissements.

- Choisissez *enable* pour les nouveaux projets et les projets actifs pour lesquels vous souhaitez vous protéger contre les exceptions de référence Null.

Exemple :

```xml
<Nullable>enable</Nullable>
```

Vous pouvez aussi utiliser des directives pour définir ces mêmes contextes partout dans votre code source. Celles-ci sont particulièrement utiles lorsque vous migrez un codebase volumineux.

- `#nullable enable` : Définit le contexte d'annotation nullable et le contexte d'avertissement nullable sur **enable**.
- `#nullable disable` : Définit le contexte d'annotation nullable et le contexte d'avertissement nullable sur **disable**.
- `#nullable restore` : Restaure le contexte d'annotation nullable et le contexte d'avertissement nullable avec les paramètres du projet.
- `#nullable disable warnings` : Définit le contexte d'avertissement nullable sur **disable**.
- `#nullable enable warnings` :Définit le contexte d'avertissement nullable sur **enable**.
- `#nullable restore warnings` : Restaure le contexte d'avertissement nullable avec les paramètres du projet.
- `#nullable disable annotations` : Définit le contexte d'annotation nullable sur **disable**.
- `#nullable enable annotations` : Définit le contexte d'annotation nullable sur **enable**.
- `#nullable restore annotations` : Restaure le contexte d'avertissement d'annotation avec les paramètres du projet.

Pour n'importe quelle ligne de code, vous pouvez définir l'une des combinaisons suivantes :

Contexte d'avertissement	Contexte des annotations	Utilisation
paramètres par défaut du projet	paramètres par défaut du projet	Default
enable	disable	Corriger les avertissements liés à l'analyse du code

Contexte d'avertissement	Contexte des annotations	Utilisation
enable	paramètres par défaut du projet	Corriger les avertissements liés à l'analyse du code
paramètres par défaut du projet	enable	Ajouter des annotations de type
enable	enable	Code déjà migré
disable	enable	Annoter le code avant de corriger les avertissements
disable	disable	Ajout du code hérité au projet migré
paramètres par défaut du projet	disable	Rarement
disable	paramètres par défaut du projet	Rarement

Ces neuf combinaisons vous offrent un contrôle précis sur les diagnostics émis pour votre code par le compilateur. Vous pouvez activer davantage de fonctionnalités dans n'importe quelle zone que vous mettez à jour, sans voir d'avertissements supplémentaires que vous n'êtes pas encore prêt à traiter.

> **ⓘ Important**
>
> Le contexte global pouvant accepter la valeur Null ne s'applique pas aux fichiers de code générés. Dans l'une ou l'autre stratégie, le contexte pouvant accepter la valeur Null est *désactivé* pour tout fichier source marqué comme généré. Cela signifie que des API dans les fichiers générés ne sont pas annotées. Il existe quatre façons de marquer un fichier comme généré :
>
> 1. Dans .editorconfig, spécifiez `generated_code = true` dans une section qui s'applique à ce fichier.
> 2. Placez `<auto-generated/>` ou `<auto-generated>` dans un commentaire en haut du fichier. Il peut se trouver sur n'importe quelle ligne de ce commentaire, mais le bloc de commentaires doit être le premier élément du fichier.
> 3. Le nom du fichier doit commencer par *TemporaryGeneratedFile_*.
> 4. Le nom du fichier doit se terminer par *.designer.cs*, *.generated.cs*, *.g.cs* ou *.g.i.cs*.
>
> Les générateurs peuvent s'inscrire à l'aide de la **#nullable** directive du préprocesseur.

Les contextes d'annotation et d'avertissement nullables sont **désactivés**. Cela signifie que votre code existant compile sans rien changer et sans générer aucun nouvel avertissement. À partir de .NET 6, les nouveaux projets incluent l'élément `<Nullable>enable</Nullable>` dans les modèles de projet.

Ces options fournissent deux stratégies distinctes pour mettre à jour un codebase existant afin d'utiliser des types référence pouvant accepter la valeur Null.

Pièges connus

Les tableaux et les structs qui contiennent des types référence sont des pièges connus dans les références pouvant accepter la valeur Null et l'analyse statique qui détermine la sécurité null. Dans les deux cas, une référence non nullable peut être initialisée sur `null`, sans générer d'avertissements.

Structures

Un struct qui contient des types référence non nullables autorise l'assignation de `default` pour celui-ci sans avertissement. Prenons l'exemple suivant :

C#

```
using System;

#nullable enable

public struct Student
{
    public string FirstName;
    public string? MiddleName;
    public string LastName;
}

public static class Program
{
    public static void PrintStudent(Student student)
    {
        Console.WriteLine($"First name: {student.FirstName.ToUpper()}");
        Console.WriteLine($"Middle name: {student.MiddleName?.ToUpper()}");
        Console.WriteLine($"Last name: {student.LastName.ToUpper()}");
    }

    public static void Main() => PrintStudent(default);
}
```

Dans l'exemple précédent, il n'y a aucun avertissement dans `PrintStudent(default)` alors que les types de référence non nullables `FirstName` et `LastName` sont null.

Un autre cas plus commun concerne le traitement des structs génériques. Prenons l'exemple suivant :

C#

```
#nullable enable

public struct Foo<T>
{
    public T Bar { get; set; }
}

public static class Program
{
    public static void Main()
    {
        string s = default(Foo<string>).Bar;
    }
}
```

Dans l'exemple précédent, la propriété `Bar` va être `null` au moment de l'exécution, et elle est affectée à une chaîne non nullable sans aucun avertissement.

Tableaux

Les tableaux sont également un piège connu dans les types référence nullables. Prenons l'exemple suivant qui ne génère aucun avertissement :

C#

```
using System;

#nullable enable

public static class Program
{
    public static void Main()
    {
        string[] values = new string[10];
        string s = values[0];
        Console.WriteLine(s.ToUpper());
    }
}
```

Dans l'exemple précédent, la déclaration du tableau montre qu'il contient des chaînes non nullables, tandis que ses éléments sont tous initialisés sur `null`. Ensuite, une variable `s` est affectée à la valeur `null` (le premier élément du tableau). Enfin, la variable `s` est déréférencée, ce qui cause une exception d'exécution.

Voir aussi

- Proposition de types références Nullables
- Brouillon de spécification des types référence Nullable
- Annotations des paramètres de type sans contraintes
- Tutoriel d'introduction aux références nullables
- **Nullable** (option du compilateur C#)

Mettre à jour un code base avec des types référence pouvant accepter la valeur Null pour améliorer les avertissements de diagnostics Null

Article • 07/04/2023

Les types de référence nullables vous permettent de déclarer si les variables d'un type référence doivent ou ne doivent pas être affectées à une valeur `null`. L'analyse statique et les avertissements du compilateur lorsque votre code peut faire l'objet d'une déréférencement `null` constituent l'avantage le plus important de cette fonctionnalité. Une fois activé, le compilateur génère des avertissements qui vous aident à éviter de lever System.NullReferenceException une fois votre code exécuté.

Si votre codebase est relativement petit, vous pouvez activer la fonctionnalité dans votre projet, traiter les avertissements et profiter des avantages des diagnostics améliorés. Les bases de code plus volumineuses peuvent nécessiter une approche plus structurée pour traiter les avertissements au fil du temps, ce qui active la fonctionnalité pour certains d'entre eux lorsque vous adressez des avertissements dans différents types ou fichiers. Cet article décrit différentes stratégies pour mettre à jour une base de code et les compromis associés à ces stratégies. Avant de commencer votre migration, lisez la vue d'ensemble conceptuelle des types de références nullables. Celle-ci couvre l'analyse statique du compilateur, les valeurs *d'état nullmaybe-null* et *not-null* et les annotations nullables. Une fois que vous connaîtrez ces concepts et termes, vous serez prêt à migrer votre code.

Planifier votre migration

Quelle que soit la façon dont vous mettez à jour votre codebase, l'objectif est que les avertissements nullables et les annotations nullables soient activés dans votre projet. Une fois que vous atteignez cet objectif, vous aurez le paramètre `<nullable>Enable</nullable>` dans votre projet. Vous n'aurez pas besoin des directives de préprocesseur pour ajuster les paramètres ailleurs.

Le premier choix consiste à définir la valeur par défaut du projet. Les choix sont les suivants :

1. ***Désactivation nullable comme valeur par défaut*** : *la désactivation* est la valeur par défaut si vous n'ajoutez pas d'élément `Nullable` à votre fichier projet. Utilisez cette

valeur par défaut lorsque vous n'ajoutez pas activement de nouveaux fichiers à la base de code. L'activité principale consiste à mettre à jour la bibliothèque pour utiliser des types de référence nullables. L'utilisation de cette valeur par défaut signifie que vous ajoutez une directive de préprocesseur nullable à chaque fichier lorsque vous mettez à jour son code.

2. ***Activation nullable comme valeur par défaut*** : définissez cette valeur par défaut lorsque vous développez activement de nouvelles fonctionnalités. Vous souhaitez que tout nouveau code tire parti des types de référence nullables et de l'analyse statique nullable. L'utilisation de cette valeur par défaut signifie que vous devez ajouter un `#nullable disable` au début de chaque fichier. Vous supprimerez ces directives de préprocesseur lorsque vous traiterez les avertissements dans chaque fichier.
3. ***Avertissements nullables comme valeur par défaut*** : choisissez cette valeur par défaut pour une migration en deux phases. Dans la première phase, adressez les avertissements. Dans la deuxième phase, activez les annotations pour déclarer *l'état null* attendu d'une variable. L'utilisation de cette valeur par défaut signifie que vous devez ajouter un `#nullable disable` au début de chaque fichier.
4. ***Annotations nullables*** comme valeur par défaut. Annotez le code avant de traiter les avertissements.

L'activation nullable comme valeur par défaut crée un travail plus frontal pour ajouter les directives de préprocesseur à chaque fichier. L'avantage est que chaque nouveau fichier de code ajouté au projet sera nullable. Tout nouveau travail prend en compte la valeur Null ; seul le code existant doit être mis à jour. La désactivation de la valeur nullable comme valeur par défaut fonctionne mieux si la bibliothèque est stable et que le principal objectif du développement est d'adopter des types de référence nullables. Vous activez les types de référence nullables en annotant les API. Une fois que vous avez terminé, vous activez les types de référence nullables pour l'ensemble du projet. Lorsque vous créez un nouveau fichier, vous devez ajouter les directives de préprocesseur et le rendre nullable. Si des développeurs de votre équipe l'oublient, ce nouveau code se trouve désormais dans le backlog de travail pour rendre tout code nullable.

La stratégie que vous choisissez dépend de la quantité de développement actif qui se déroule dans votre projet. Plus votre projet est mature et stable, plus la deuxième stratégie est efficace. Plus les fonctionnalités sont développées, plus la première stratégie est efficace.

> **ⓘ Important**
>
> Le contexte global pouvant accepter la valeur Null ne s'applique pas aux fichiers de code générés. Dans l'une ou l'autre stratégie, le contexte pouvant accepter la valeur

Null est *désactivé* pour tout fichier source marqué comme généré. Cela signifie que des API dans les fichiers générés ne sont pas annotées. Il existe quatre façons de marquer un fichier comme généré :

1. Dans .editorconfig, spécifiez `generated_code = true` dans une section qui s'applique à ce fichier.
2. Placez `<auto-generated/>` ou `<auto-generated>` dans un commentaire en haut du fichier. Il peut se trouver sur n'importe quelle ligne de ce commentaire, mais le bloc de commentaires doit être le premier élément du fichier.
3. Le nom du fichier doit commencer par *TemporaryGeneratedFile_*.
4. Le nom du fichier doit se terminer par *.designer.cs*, *.generated.cs*, *.g.cs* ou *.g.i.cs*.

Les générateurs peuvent s'inscrire à l'aide de la directive de préprocesseur **#nullable**.

Comprendre les contextes et les avertissements

L'activation d'avertissements et d'annotations contrôle la façon dont le compilateur traite les types de référence et la nullabilité. Chaque type a l'une des trois capacités Null :

- *oblivious* : tous les types de références sont *inconscients* à la valeur Null lorsque le contexte d'annotation est désactivé.
- *nonnullable* : type de référence non annoté, `C` n'est *pas nullable* lorsque le contexte d'annotation est activé.
- *nullable* : type de référence annoté, `C?` est *nullable*, mais un avertissement peut être émis lorsque le contexte d'annotation est désactivé. Les variables déclarées avec `var` sont *nullables* lorsque le contexte d'annotation est activé.

Le compilateur génère des avertissements en fonction de cette nullabilité :

- Les types *nonnullable* provoquent des avertissements si une valeur potentielle `null` est affectée à ces derniers.
- Les types *nullable* provoquent des avertissements s'ils se déréférencent en *maybe-null*.
- Les types *oblivious* provoquent des avertissements s'ils se déréférencent en *maybe-null* et que le contexte d'avertissement est activé.

Chaque variable a un état nullable par défaut qui dépend de sa nullabilité :

- Les variables nullables ont comme *état null* par défaut *maybe-null*.

- Les variables non-nullables ont comme *état null* par défaut *not-null*.
- Les variables inconscientes nullables ont comme *état null* par défaut *not-null*.

Avant d'activer les types de référence nullables, toutes les déclarations de votre codebase sont *oblivious*. Cela est important, car cela signifie que tous les types de référence ont un *état null* par défaut *non null*.

Avertissements d'adresse

Si votre projet utilise Entity Framework Core, vous devez lire leurs instructions sur l'utilisation de typesde référence nullable.

Lorsque vous démarrez votre migration, vous devez commencer par activer uniquement les avertissements. Toutes les déclarations restent *oublivious*, mais vous verrez des avertissements lorsque vous déréférencerez une valeur après que son *état null* change en *maybe-null*. Lorsque vous traitez ces avertissements, vous vérifierez la valeur Null dans d'autres emplacements, et votre codebase deviendra plus résilient. Pour découvrir des techniques spécifiques pour différentes situations, consultez l'article sur les techniques de résolution des avertissements nullables.

Vous pouvez traiter les avertissements et activer les annotations dans chaque fichier ou classe avant de continuer avec d'autres codes. Toutefois, il est souvent plus efficace de traiter les avertissements générés pendant que le contexte est *avertissements* avant d'activer les annotations de type. De cette façon, tous les types sont *oblivious* jusqu'à ce que vous ayez traité le premier ensemble d'avertissements.

Activer les annotations de type

Après avoir adressé le premier ensemble d'avertissements, vous pouvez activer le *contexte d'annotation*. Cela modifie les types de référence d'*oblivious* à *nonnullable*. Toutes les variables déclarées avec `var` sont *nullables*. Cette modification introduit souvent de nouveaux avertissements. La première étape de l'adressage des avertissements du compilateur consiste à utiliser des annotations `?` sur les types de paramètres et de retour pour indiquer quand des arguments ou des valeurs de retour peuvent être `null`. En effectuant cette tâche, votre objectif n'est pas simplement de corriger les avertissements. L'objectif le plus important est de faire comprendre au compilateur votre intention pour les valeurs null potentielles.

Les attributs étendent les annotations de type

Plusieurs attributs ont été ajoutés pour exprimer des informations supplémentaires sur l'état null des variables. Les règles de vos API sont probablement plus complexes que *non-null* ou *maybe-null* pour tous les paramètres et valeurs de retour. La plupart de vos API ont des règles plus complexes pour les variables pouvant ou ne pouvant pas être `null`. Dans ces cas-là, vous utiliserez des attributs pour exprimer ces règles. Les attributs qui décrivent la sémantique de votre API se trouvent dans l'article sur les attributs affectant l'analyse nullable.

Étapes suivantes

Une fois que vous avez résolu tous les avertissements après l'activation des annotations, vous pouvez définir le contexte par défaut pour que votre projet soit *activé*. Si vous avez ajouté des pragmas dans votre code pour l'annotation nullable ou le contexte d'avertissement, vous pouvez les supprimer. Au fil du temps, vous pouvez voir de nouveaux avertissements. Vous pouvez écrire du code qui introduit des avertissements. Une dépendance de bibliothèque peut être mise à jour pour les types de référence nullables. Ces mises à jour modifient les types de cette bibliothèque d'*oblivious* à *nonnullable* ou *nullable*.

Vous pouvez également explorer ces concepts dans notre module Learn sur la sécurité des valeurs Null en C#.

Résoudre les avertissements nullables

Article • 21/03/2023

Cet article traite des avertissements suivantes du compilateur :

- CS8597 - *La valeur levée est peut-être null.*
- CS8600 - *Conversion de littéral ayant une valeur null ou d'une éventuelle valeur null en type non nullable.*
- CS8601 - *Possible affectation d'une référence null.*
- CS8602 - *Déréférencement d'une référence possiblement null.*
- CS8603 - *Retour d'une possible référence null.*
- CS8604 - *Possible argument de référence null pour le paramètre.*
- CS8605 - *Conversion unboxing d'une valeur possiblement null.*
- CS8607 - *Une possible valeur null ne peut pas être utilisée pour un type marqué avec* `[NotNull]` *ou* `[DisallowNull]`
- CS8608 - *La nullabilité des types référence dans le type ne correspond pas au membre surchargé.*
- CS8609 - *La nullabilité des types référence dans le type de retour ne correspond pas au membre surchargé.*
- CS8610 - *La nullabilité des types référence dans le type de paramètre ne correspond pas au membre surchargé.*
- CS8611 - *La nullabilité des types référence dans le type de paramètre ne correspond pas à la déclaration de méthode partielle.*
- CS8612 - *La nullabilité des types référence dans le type ne correspond pas au membre implémenté implicitement.*
- CS8613 - *La nullabilité des types référence dans le type de retour ne correspond pas au membre implémenté implicitement.*
- CS8614 - *La nullabilité des types référence dans le type de paramètre ne correspond pas au membre implémenté implicitement.*
- CS8615 - *La nullabilité des types référence dans le type ne correspond pas au membre implémenté.*
- CS8616 - *La nullabilité des types référence dans le type de retour ne correspond pas au membre implémenté.*
- CS8617 - *La nullabilité des types référence dans le type de paramètre ne correspond pas au membre implémenté.*
- CS8618 - *Une variable non nullable doit contenir une valeur non null au moment de la sortie du constructeur. Envisagez de la déclarer comme nullable.*
- CS8619 - *La nullabilité des types référence dans la valeur ne correspond pas au type cible.*

- CS8620 - *Impossible d'utiliser l'argument pour le paramètre, car il existe des différences dans la nullabilité des types référence.*
- CS8621 - *La nullabilité des types référence dans le type de retour ne correspond pas au délégué cible (possiblement en raison des attributs de nullabilité).*
- CS8622 - *La nullabilité des types référence dans le type de paramètre ne correspond pas au délégué cible (possiblement en raison des attributs de nullabilité).*
- CS8624 - *Impossible d'utiliser l'argument comme sortie, car il existe des différences dans la nullabilité des types référence.*
- CS8625 - *Impossible de convertir un littéral null en type référence non nullable.*
- CS8629 - *Le type valeur nullable est peut-être null.*
- CS8631 - *Impossible d'utiliser le type en tant que paramètre de type dans le type ou la méthode générique. La nullabilité de l'argument de type ne correspond pas au type de contrainte.*
- CS8633 - *La nullabilité dans les contraintes pour le paramètre de type de la méthode ne correspond pas aux contraintes pour le paramètre de type de la méthode d'interface. Utilisez à la place une implémentation d'interface explicite.*
- CS8634 - *Impossible d'utiliser le type en tant que paramètre de type dans le type ou la méthode générique. La nullabilité de l'argument de type ne correspond pas à la contrainte « class ».*
- CS8643 - *La nullabilité des types référence dans le spécificateur d'interface explicite ne correspond pas à l'interface implémentée par le type.*
- CS8644 - *Le type n'implémente pas le membre d'interface. La nullabilité des types référence dans l'interface implémentée par le type de base ne correspond pas.*
- CS8645 - *Le membre figure déjà dans la liste des interfaces sur le type avec une nullabilité différente des types référence.*
- CS8655 - *L'expression switch ne prend pas en charge certaines entrées null (elle n'est pas exhaustive).*
- CS8667 - *Les déclarations de méthodes partielles ont une nullabilité incohérente dans les contraintes pour le paramètre de type.*
- CS8670 - *L'initialiseur d'objet ou de collection déréférence implicitement le membre possiblement null.*
- CS8714 - *Impossible d'utiliser le type en tant que paramètre de type dans le type ou la méthode générique. La nullabilité de l'argument de type ne correspond pas à la contrainte « notnull ».*
- CS8762 - *Le paramètre doit avoir une valeur non null au moment de la sortie.*
- CS8763 - *Une méthode marquée `[DoesNotReturn]` ne doit pas retourner.*
- CS8764 - *La nullabilité du type de retour ne correspond pas au membre surchargé (possiblement en raison des attributs de nullabilité).*
- CS8765 - *La nullabilité de type du paramètre ne correspond pas au membre surchargé (possiblement en raison des attributs de nullabilité).*

- CS8766 - *La nullabilité des types référence dans le type de retour ne correspond pas au membre implémenté implicitement (possiblement en raison des attributs de nullabilité).*
- CS8767 - *La nullabilité des types référence dans le type de retour ne correspond pas au membre implémenté implicitement (possiblement en raison des attributs de nullabilité).*
- CS8768 - *La nullabilité des types référence dans le type de retour ne correspond pas au membre implémenté (possiblement en raison des attributs de nullabilité).*
- CS8769 - *La nullabilité des types référence dans le type de paramètre ne correspond pas au membre implémenté (possiblement en raison des attributs de nullabilité).*
- CS8770 - *La méthode n'a pas d'annotation* `[DoesNotReturn]` *correspondant au membre implémenté ou surchargé.*
- CS8774 - *Le paramètre doit avoir une valeur non null au moment de la sortie.*
- CS8776 - *Le membre ne peut pas être utilisé dans cet attribut.*
- CS8775 - *Le paramètre doit avoir une valeur non null au moment de la sortie.*
- CS8777 - *Le paramètre doit avoir une valeur non null au moment de la sortie.*
- CS8819 - *La nullabilité des types référence dans le type de retour ne correspond pas à la déclaration de méthode partielle.*
- CS8824 - *Le paramètre doit avoir une valeur non null au moment de la sortie, car le paramètre a une valeur non null.*
- CS8825 - *La valeur de retour doit être non null, car le paramètre est non null.*
- CS8847 - *L'expression switch ne gère pas certaines entrées null (elle n'est pas exhaustive). Cependant, un modèle avec une clause « when » pourrait établit une correspondance avec cette valeur.*

L'objectif des avertissements « nullables » est de réduire les risques que votre application lève une System.NullReferenceException lors de l'exécution. Pour atteindre cet objectif, le compilateur utilise l'analyse statique et émet des avertissements lorsque votre code a des constructions qui peuvent entraîner des exceptions de référence null. Vous fournissez au compilateur des informations pour son analyse statique en appliquant des attributs et des annotations de type. Ces annotations et attributs décrivent la nullabilité des arguments, paramètres et membres de vos types. Dans cet article, vous allez découvrir différentes techniques pour corriger les avertissements nullables que le compilateur génère à partir de son analyse statique. Les techniques décrites ici concernent le code C# général. Découvrez comment utiliser les types référence nullables et Entity Framework Core dans Utilisation des types référence nullables.

Vous résoudrez presque tous les avertissements à l'aide de l'une des quatre techniques suivantes :

- Ajout de contrôles de valeur null nécessaires.

- Ajout d'annotations nullables `?` ou `!`.
- Ajout d'attributs qui décrivent la sémantique null.
- Initialisation correcte des variables.

Déréférencement possible de null

Cet ensemble d'avertissements vous alerte du fait que vous déréférencez une variable dont l'*état de nullabilité* (null-state) est *peut-être null* (maybe-null). Ces avertissements sont les suivants :

- **CS8602** - *Déréférencement d'une référence possiblement null.*
- **CS8670** - *L'initialiseur d'objet ou de collection déréférence implicitement le membre possiblement null.*

Le code suivant montre un exemple de chacun des avertissements précédents :

C#

```
class Container
{
    public List<string>? States { get; set; }
}

internal void PossibleDereferenceNullExamples(string? message)
{
    Console.WriteLine(message.Length); // CS8602

    var c = new Container { States = { "Red", "Yellow", "Green" } }; //
CS8670
}
```

Dans l'exemple ci-dessus, l'avertissement est dû au fait que le `Container`, `c`, peut avoir une valeur null pour la propriété `States`. L'affectation de nouveaux états à une collection qui peut être null provoque l'avertissement.

Pour supprimer ces avertissements, vous devez ajouter du code pour changer le *null-state* de cette variable en *not-null* avant de la déréférencer. L'avertissement de l'initialiseur de collection peut être plus difficile à repérer. Le compilateur détecte que la collection *peut être null* quand l'initialiseur y ajoute des éléments.

Dans de nombreux cas, vous pouvez corriger ces avertissements en vérifiant qu'une variable n'est pas null avant de la déréférencer. Par exemple, l'exemple ci-dessus pourrait être réécrit comme ceci :

C#

```
void WriteMessageLength(string? message)
{
    if (message is not null)
    {
        Console.WriteLine(message.Length);
    }
}
```

D'autres cas où vous recevez ces avertissements peuvent être faux positifs. Vous pouvez avoir une méthode d'utilitaire privée qui teste la valeur null. Le compilateur ne sait pas que la méthode fournit un contrôle de valeur null. Prenons l'exemple suivant qui utilise une méthode d'utilitaire privée, `IsNotNull` :

C#

```
public void WriteMessage(string? message)
{
    if (IsNotNull(message))
        Console.WriteLine(message.Length);
}
```

Le compilateur avertit que vous pouvez déréférencer la valeur null lorsque vous écrivez la propriété `message.Length`, car son analyse statique détermine que `message` peut être `null`. Vous savez peut-être que `IsNotNull` fournit un contrôle de valeur null et quand celui-ci retourne `true`, le *null-state* de `message` doit être *not-null*. Vous devez indiquer ces faits au compilateur. Un moyen consiste à utiliser l'opérateur null forgiving, `!`. Vous pouvez changer l'instruction `WriteLine` pour la faire correspondre au code suivant :

C#

```
Console.WriteLine(message!.Length);
```

L'opérateur null forgiving rend l'expression *not-null* même si elle était *maybe-null* sans le `!` appliqué. Dans cet exemple, une meilleure solution consiste à ajouter un attribut à la signature de `IsNotNull` :

C#

```
private static bool IsNotNull([NotNullWhen(true)] object? obj) => obj !=
null;
```

System.Diagnostics.CodeAnalysis.NotNullWhenAttribute informe le compilateur que l'argument utilisé pour le paramètre `obj` est *not-null* lorsque la méthode retourne `true`.

Lorsque la méthode retourne `false`, l'argument a le même *null-state* qu'il avait avant l'appel de la méthode.

> **Conseil**
>
> Vous pouvez utiliser un ensemble complet d'attributs pour décrire la façon dont vos méthodes et propriétés affectent *null-state*. Vous les trouverez dans l'article de référence sur le langage, **Attributs d'analyse statique nullables**.

La correction d'un avertissement pour le déréférencement d'une variable *maybe-null* implique l'une des trois techniques suivantes :

- Ajouter un contrôle de valeur null manquant.
- Ajouter des attributs d'analyse null sur les API pour affecter l'analyse statique du *null-state* du compilateur. Ces attributs informent le compilateur lorsqu'une valeur de retour ou un argument doit être *maybe-null* ou *not-null* après l'appel de la méthode.
- Appliquer l'opérateur null forgiving `!` à l'expression pour forcer l'état à être *not-null*.

Valeur null possible attribuée à une référence non nullable

Cet ensemble d'avertissements vous alerte du fait que vous affectez une variable dont le type est non nullable à une expression dont *l'état de nullabilité* est *peut-être null*. Ces avertissements sont les suivants :

- **CS8597** - *La valeur levée est peut-être null.*
- **CS8600** - *Conversion de littéral ayant une valeur null ou d'une éventuelle valeur null en type non nullable.*
- **CS8601** - *Possible affectation d'une référence null.*
- **CS8603** - *Retour d'une possible référence null*
- **CS8604** - *Possible argument de référence null pour le paramètre.*
- **CS8605** - *Conversion unboxing d'une valeur possiblement null.*
- **CS8625** - *Impossible de convertir un littéral null en type référence non nullable.*
- **CS8629** - *Le type valeur nullable est peut-être null.*

Le compilateur émet ces avertissements lorsque vous tentez d'affecter une expression qui est *maybe-null* à une variable qui est non nullable. Par exemple :

```
C#
```

```
string? TryGetMessage(int id) => "";

string msg = TryGetMessage(42);  // Possible null assignment.
```

Les différents avertissements fournissent des détails sur le code, comme l'affectation, l'affectation de conversion unboxing, les instructions de retour, les arguments des méthodes et les expressions throw.

Vous pouvez effectuer l'une des trois actions pour résoudre ces avertissements. L'une consiste à ajouter l'annotation `?` pour faire de la variable un type référence nullable. Ce changement peut entraîner d'autres avertissements. Le changement d'une variable en remplaçant une référence non nullable par une référence nullable change son *null-state* par défaut, qui passe de *not-null* à *maybe-null*. L'analyse statique du compilateur peut trouver des instances où vous déréférencez une variable qui est *maybe-null*.

Les autres actions indiquent au compilateur que le côté droit de l'affectation est *not-null*. L'expression à droite peut faire l'objet d'un contrôle de valeur null avant l'affectation, comme illustré dans l'exemple suivant :

C#

```
string notNullMsg = TryGetMessage(42) ?? "Unknown message id: 42";
```

Les exemples précédents illustrent l'affectation de la valeur de retour d'une méthode. Vous pouvez annoter la méthode (ou la propriété) pour indiquer quand une méthode retourne une valeur not-null. System.Diagnostics.CodeAnalysis.NotNullIfNotNullAttribute spécifie souvent qu'une valeur de retour est *not-null* lorsqu'un argument d'entrée est *not-null*. Une autre alternative consiste à ajouter l'opérateur null forgiving, `!`, à droite :

C#

```
string msg = TryGetMessage(42)!;
```

La résolution d'un avertissement pour l'affectation d'une expression *not-null* à une variable *not-null* implique l'une des quatre techniques suivantes :

- Remplacer le côté gauche de l'affectation par un type nullable. Cette action peut introduire de nouveaux avertissements lorsque vous déréférencez cette variable.
- Fournir un contrôle de valeur null avant l'affectation.
- Annoter l'API qui produit le côté droit de l'affectation.
- Ajouter l'opérateur null forgiving à droite de l'affectation.

Référence non nullable non initialisée

Cet ensemble d'avertissements vous alerte du fait que vous affectez une variable dont le type est non nullable à une expression dont *l'état de nullabilité* est *peut-être null*. Ces avertissements sont les suivants :

- **CS8618** - *Une variable non nullable doit contenir une valeur non null au moment de la sortie du constructeur. Envisagez de la déclarer comme nullable.*
- **CS8762** - *Le paramètre doit avoir une valeur non null au moment de la sortie.*

Prenons la classe suivante comme exemple :

C#

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

Ni `FirstName` ni `LastName` ne sont garantis initialisés. Si ce code est nouveau, envisagez de changer l'interface publique. L'exemple ci-dessus pourrait être mis à jour comme ceci :

C#

```
public class Person
{
    public Person(string first, string last)
    {
        FirstName = first;
        LastName = last;
    }

    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

Si vous devez créer un objet `Person` avant de définir le nom, vous pouvez initialiser les propriétés à l'aide d'une valeur par défaut non null :

C#

```
public class Person
{
    public string FirstName { get; set; } = string.Empty;
```

```
    public string LastName { get; set; } = string.Empty;
}
```

Une autre alternative consiste à remplacer ces membres par des types référence nullables. La classe `Person` pourrait être définie comme suit si `null` devait être autorisé pour le nom :

C#

```
public class Person
{
    public string? FirstName { get; set; }
    public string? LastName { get; set; }
}
```

Le code existant peut demander d'autres changements pour informer le compilateur de la sémantique null pour ces membres. Vous avez peut-être créé plusieurs constructeurs et votre classe peut avoir une méthode d'assistance privée qui initialise un ou plusieurs membres. Vous pouvez déplacer le code d'initialisation dans un seul constructeur et vous assurer que tous les constructeurs appellent celui avec le code d'initialisation commun. Vous pouvez également utiliser les attributs System.Diagnostics.CodeAnalysis.MemberNotNullAttribute et System.Diagnostics.CodeAnalysis.MemberNotNullWhenAttribute. Ces attributs informent le compilateur qu'un membre est *not-null* une fois que la méthode a été appelée. Le code suivant montre un exemple de chaque. La classe `Person` utilise un constructeur commun appelé par tous les autres constructeurs. La classe `Student` a une méthode d'assistance annotée avec l'attribut System.Diagnostics.CodeAnalysis.MemberNotNullAttribute :

C#

```
using System.Diagnostics.CodeAnalysis;

public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

    public Person() : this("John", "Doe") { }
```

```
}

public class Student : Person
{
    public string Major { get; set; }

    public Student(string firstName, string lastName, string major)
        : base(firstName, lastName)
    {
        SetMajor(major);
    }

    public Student(string firstName, string lastName) :
        base(firstName, lastName)
    {
        SetMajor();
    }

    public Student()
    {
        SetMajor();
    }

    [MemberNotNull(nameof(Major))]
    private void SetMajor(string? major = default)
    {
        Major = major ?? "Undeclared";
    }
}
```

Enfin, vous pouvez utiliser l'opérateur null forgiving pour indiquer qu'un membre est initialisé dans un autre code. Pour avoir un autre exemple, prenons les classes suivantes représentant un modèle Entity Framework Core :

C#

```
public class TodoItem
{
    public long Id { get; set; }
    public string? Name { get; set; }
    public bool IsComplete { get; set; }
}

public class TodoContext : DbContext
{
    public TodoContext(DbContextOptions<TodoContext> options)
        : base(options)
    {
    }

    public DbSet<TodoItem> TodoItems { get; set; } = null!;
}
```

La propriété `DbSet` est initialisée à `null!`. Cela indique au compilateur que la propriété est définie sur une valeur *not-null*. En fait, le `DbContext` de base effectue l'initialisation de la définition. L'analyse statique du compilateur ne s'en occupe pas. Pour plus d'informations sur l'utilisation des types référence nullables et d'Entity Framework Core, consultez l'article Utilisation des types référence nullables dans EF Core.

La résolution d'un avertissement pour ne pas initialiser un membre non nullable implique l'une des quatre techniques suivantes :

- Changer les constructeurs ou les initialiseurs de champ pour vous assurer que tous les membres non nullables sont initialisés.
- Changer un ou plusieurs membres pour qu'ils soient des types nullables.
- Annoter toutes les méthodes d'assistance pour indiquer quels membres sont affectés.
- Ajouter un initialiseur à `null!` pour indiquer que le membre est initialisé dans un autre code.

Incompatibilité dans une déclaration de nullabilité

D'autres avertissements indiquent des non-correspondances de nullabilité entre les signatures pour les méthodes, les délégués ou les paramètres de type.

- **CS8608** - *La nullabilité des types référence dans le type ne correspond pas au membre surchargé.*
- **CS8609** - *La nullabilité des types référence dans le type de retour ne correspond pas au membre surchargé.*
- **CS8610** - *La nullabilité des types référence dans le type de paramètre ne correspond pas au membre surchargé.*
- **CS8611** - *La nullabilité des types référence dans le type de paramètre ne correspond pas à la déclaration de méthode partielle.*
- **CS8612** - *La nullabilité des types référence dans le type ne correspond pas au membre implémenté implicitement.*
- **CS8613** - *La nullabilité des types référence dans le type de retour ne correspond pas au membre implémenté implicitement.*
- **CS8614** - *La nullabilité des types référence dans le type de paramètre ne correspond pas au membre implémenté implicitement.*
- **CS8615** - *La nullabilité des types référence dans le type ne correspond pas au membre implémenté.*
- **CS8616** - *La nullabilité des types référence dans le type de retour ne correspond pas au membre implémenté.*

- **CS8617** - *La nullabilité des types référence dans le type de paramètre ne correspond pas au membre implémenté.*
- **CS8619** - *La nullabilité des types référence dans la valeur ne correspond pas au type cible.*
- **CS8620** - *Impossible d'utiliser l'argument pour le paramètre, car il existe des différences dans la nullabilité des types référence.*
- **CS8621** - *La nullabilité des types référence dans le type de retour ne correspond pas au délégué cible (possiblement en raison des attributs de nullabilité).*
- **CS8622** - *La nullabilité des types référence dans le type de paramètre ne correspond pas au délégué cible (possiblement en raison des attributs de nullabilité).*
- **CS8624** - *Impossible d'utiliser l'argument comme sortie, car il existe des différences dans la nullabilité des types référence.*
- **CS8631** - *Impossible d'utiliser le type en tant que paramètre de type dans le type ou la méthode générique. La nullabilité de l'argument de type ne correspond pas au type de contrainte.*
- **CS8633** - *La nullabilité dans les contraintes pour le paramètre de type de la méthode ne correspond pas aux contraintes pour le paramètre de type de la méthode d'interface. Utilisez à la place une implémentation d'interface explicite.*
- **CS8634** - *Impossible d'utiliser le type en tant que paramètre de type dans le type ou la méthode générique. La nullabilité de l'argument de type ne correspond pas à la contrainte « class ».*
- **CS8643** - *La nullabilité des types référence dans le spécificateur d'interface explicite ne correspond pas à l'interface implémentée par le type.*
- **CS8644** - *Le type n'implémente pas le membre d'interface. La nullabilité des types référence dans l'interface implémentée par le type de base ne correspond pas.*
- **CS8645** - *Le membre figure déjà dans la liste des interfaces sur le type avec une nullabilité différente des types référence.*
- **CS8667** - *Les déclarations de méthodes partielles ont une nullabilité incohérente dans les contraintes pour le paramètre de type.*
- **CS8714** - *Impossible d'utiliser le type en tant que paramètre de type dans le type ou la méthode générique. La nullabilité de l'argument de type ne correspond pas à la contrainte « notnull ».*
- **CS8764** - *La nullabilité du type de retour ne correspond pas au membre surchargé (possiblement en raison des attributs de nullabilité).*
- **CS8765** - *La nullabilité de type du paramètre ne correspond pas au membre surchargé (possiblement en raison des attributs de nullabilité).*
- **CS8766** - *La nullabilité des types référence dans le type de retour ne correspond pas au membre implémenté implicitement (possiblement en raison des attributs de nullabilité).*

- **CS8767** - *La nullabilité des types référence dans le type de retour ne correspond pas au membre implémenté implicitement (possiblement en raison des attributs de nullabilité).*
- **CS8768** - *La nullabilité des types référence dans le type de retour ne correspond pas au membre implémenté (possiblement en raison des attributs de nullabilité).*
- **CS8769** - *La nullabilité des types référence dans le type de paramètre ne correspond pas au membre implémenté (possiblement en raison des attributs de nullabilité).*
- **CS8819** - *La nullabilité des types référence dans le type de retour ne correspond pas à la déclaration de méthode partielle.*

Le code suivant illustre *CS8764* :

C#

```
public class B
{
    public virtual string GetMessage(string id) => string.Empty;
}
public class D : B
{
    public override string? GetMessage(string? id) => default;
}
```

L'exemple précédent montre une méthode `virtual` dans une classe de base et un `override` avec une nullabilité différente. La classe de base retourne une chaîne non nullable, mais la classe dérivée retourne une chaîne nullable. Si `string` et `string?` étaient inversés, cela serait autorisé parce que la classe dérivée est plus restrictive. De même, les déclarations de paramètres doivent correspondre. Les paramètres de la méthode override peuvent autoriser la valeur null même si la classe de base ne l'autorise pas.

D'autres situations peuvent générer ces avertissements. Vous pouvez avoir une incompatibilité dans une déclaration de méthode d'interface et l'implémentation de cette méthode. Ou un type de délégué et l'expression de ce délégué peuvent différer. Un paramètre de type et l'argument de type peuvent différer dans la nullabilité.

Pour corriger ces avertissements, mettez à jour la déclaration appropriée.

Le code ne correspond pas à la déclaration d'attribut

Les sections précédentes ont expliqué comment utiliser les attributs pour une analyse statique nullable afin d'informer le compilateur de la sémantique null de votre code. Le

compilateur vous avertit si le code ne respecte pas les promesses de cet attribut.

- **CS8607** - *Une possible valeur null ne peut pas être utilisée pour un type marqué avec* `[NotNull]` *ou* `[DisallowNull]`
- **CS8763** - *Une méthode marquée* `[DoesNotReturn]` *ne doit pas retourner.*
- **CS8770** - *La méthode n'a pas d'annotation* `[DoesNotReturn]` *correspondant au membre implémenté ou surchargé.*
- **CS8774** - *Le paramètre doit avoir une valeur non null au moment de la sortie.*
- **CS8775** - *Le paramètre doit avoir une valeur non null au moment de la sortie.*
- **CS8776** - *Le membre ne peut pas être utilisé dans cet attribut.*
- **CS8777** - *Le paramètre doit avoir une valeur non null au moment de la sortie.*
- **CS8824** - *Le paramètre doit avoir une valeur non null au moment de la sortie, car le paramètre a une valeur non null.*
- **CS8825** - *La valeur de retour doit être non null, car le paramètre est non null.*

Considérez la méthode suivante :

```C#
public bool TryGetMessage(int id, [NotNullWhen(true)] out string? message)
{
    message = null;
    return true;

}
```

Le compilateur génère un avertissement parce que le paramètre `message` est `null` *et* la méthode retourne `true`. L'attribut `NotNullWhen` indique que cela ne devrait pas se produire.

Pour corriger ces avertissements, mettez à jour votre code afin qu'il corresponde aux attentes des attributs que vous avez appliqués. Vous pouvez changer les attributs ou l'algorithme.

Expression switch exhaustive

Les expressions switch doivent être *exhaustives*, ce qui signifie que toutes les valeurs d'entrée doivent être gérées. Même pour les types référence non nullables, la valeur `null` doit être prise en compte. Le compilateur émet des avertissements quand la valeur null n'est pas gérée :

- **CS8655** - *L'expression switch ne prend pas en charge certaines entrées null (elle n'est pas exhaustive).*

- **CS8847** - *L'expression switch ne gère pas certaines entrées null (elle n'est pas exhaustive). Cependant, un modèle avec une clause « when » pourrait établit une correspondance avec cette valeur.*

L'exemple de code suivant illustre cette condition :

C#

```
int AsScale(string status) =>
    status switch
    {
        "Red" => 0,
        "Yellow" => 5,
        "Green" => 10,
        { } => -1
    };
```

L'expression d'entrée est un `string`, et non pas un `string?`. Le compilateur génère néanmoins cet avertissement. Le modèle `{ }` gère toutes les valeurs non null, mais pas `null`. Pour traiter ces erreurs, vous pouvez ajouter un cas `null` explicite ou remplacer le `{ }` par le modèle (discard) `_`. Le modèle discard traite les cas des valeurs null ainsi que de n'importe quelle autre valeur.

Méthodes en C#

Article • 07/04/2023

Une méthode est un bloc de code qui contient une série d'instructions. Un programme provoque l'exécution des instructions en appelant la méthode et en spécifiant les éventuels arguments de méthode requis. En C#, chaque instruction exécutée est effectuée dans le contexte d'une méthode. La méthode `Main` est le point d'entrée de chaque application C# et elle est appelée par le Common Language Runtime (CLR) au démarrage du programme.

> **Notes**
>
> Cette rubrique décrit les méthodes nommées. Pour plus d'informations sur les fonctions anonymes, consultez **Expressions lambda**.

Signatures de méthode

Les méthodes sont déclarées dans `class`, `record` ou `struct` en spécifiant :

- Un niveau d'accès facultatif, comme `public` ou `private`. Par défaut, il s'agit de `private`.
- Des modificateurs facultatifs, comme `abstract` ou `sealed`.
- La valeur de retour, ou `void` si la méthode n'en a pas.
- Nom de la méthode.
- Des paramètres de méthodes. Les paramètres de méthode sont placés entre parenthèses et séparés par des virgules. Des parenthèses vides indiquent que la méthode ne requiert aucun paramètre.

Ces parties forment ensemble la signature de la méthode.

> **Important**
>
> Un type de retour d'une méthode ne fait pas partie de la signature de la méthode à des fins de surcharge de méthode. Toutefois, il fait partie de la signature de la méthode lors de la détermination de la compatibilité entre un délégué et la méthode vers laquelle il pointe.

L'exemple suivant définit une classe nommée `Motorcycle` qui contient cinq méthodes :

C#

```
using System;

abstract class Motorcycle
{
    // Anyone can call this.
    public void StartEngine() {/* Method statements here */ }

    // Only derived classes can call this.
    protected void AddGas(int gallons) { /* Method statements here */ }

    // Derived classes can override the base class implementation.
    public virtual int Drive(int miles, int speed) { /* Method statements
here */ return 1; }

    // Derived classes can override the base class implementation.
    public virtual int Drive(TimeSpan time, int speed) { /* Method
statements here */ return 0; }

    // Derived classes must implement this.
    public abstract double GetTopSpeed();
}
```

La classe `Motorcycle` inclut une méthode surchargée, `Drive`. Deux méthodes ont le même nom, mais doivent être différenciées par leurs types de paramètre.

Appel de méthode

Les méthodes peuvent être *d'instance* ou *statiques*. L'appel d'une méthode d'instance nécessite l'instanciation d'un objet et l'appel de la méthode sur cet objet ; une méthode d'instance agit sur cette instance et sur ses données. Vous appelez une méthode statique en référençant le nom du type auquel la méthode appartient ; les méthodes statiques n'agissent pas sur les données d'une instance. Une tentative d'appeler une méthode statique via une instance d'objet génère une erreur du compilateur.

Appeler une méthode est comme accéder à un champ. Après le nom d'objet (si vous appelez une méthode d'instance) ou le nom de type (si vous appelez une méthode `static`), ajoutez un point, le nom de la méthode et des parenthèses. Les arguments sont répertoriés entre les parenthèses et séparés par des virgules.

La définition de la méthode spécifie les noms et types des paramètres requis. Quand un appelant appelle la méthode, il fournit des valeurs réelles, appelées « arguments », pour chaque paramètre. Les arguments doivent être compatibles avec le type de paramètre, mais le nom de l'argument, si un nom est utilisé dans le code appelant, ne doit pas nécessairement être le même que celui du paramètre défini dans la méthode. Dans

l'exemple suivant, la méthode `Square` comprend un seul paramètre de type `int` nommé *i*. Le premier appel de la méthode passe à la méthode `Square` une variable de type `int` nommée *num*, le deuxième appel passe une constante numérique et le troisième appel passe une expression.

C#

```
public class SquareExample
{
   public static void Main()
   {
      // Call with an int variable.
      int num = 4;
      int productA = Square(num);

      // Call with an integer literal.
      int productB = Square(12);

      // Call with an expression that evaluates to int.
      int productC = Square(productA * 3);
   }

   static int Square(int i)
   {
      // Store input argument in a local variable.
      int input = i;
      return input * input;
   }
}
```

La forme la plus courante des appels de méthode utilise des arguments de position ; elle fournit les arguments dans le même ordre que celui des paramètres de la méthode. Les méthodes de la classe `Motorcycle` peuvent donc être appelées comme dans l'exemple suivant. Par exemple, l'appel à la méthode `Drive` comprend deux arguments qui correspondent aux deux paramètres de la syntaxe de la méthode. Le premier devient la valeur du paramètre `miles` et le deuxième devient la valeur du paramètre `speed`.

C#

```
class TestMotorcycle : Motorcycle
{
    public override double GetTopSpeed()
    {
        return 108.4;
    }

    static void Main()
    {
```

```
        TestMotorcycle moto = new TestMotorcycle();

        moto.StartEngine();
        moto.AddGas(15);
        moto.Drive(5, 20);
        double speed = moto.GetTopSpeed();
        Console.WriteLine("My top speed is {0}", speed);
    }
}
```

Vous pouvez également utiliser des *arguments nommés* au lieu d'arguments positionnels lors de l'appel d'une méthode. Lors de l'utilisation d'arguments nommés, vous spécifiez le nom du paramètre suivi de deux-points (« : ») et de l'argument. Les arguments de la méthode peuvent apparaître dans n'importe quel ordre, dès lors que tous les arguments nécessaires sont présents. L'exemple suivant utilise des arguments nommés pour appeler la méthode `TestMotorcycle.Drive`. Dans cet exemple, les arguments nommés sont passés dans l'ordre inverse de celui de la liste des paramètres de la méthode.

C#

```
using System;

class TestMotorcycle : Motorcycle
{
    public override int Drive(int miles, int speed)
    {
        return (int)Math.Round(((double)miles) / speed, 0);
    }

    public override double GetTopSpeed()
    {
        return 108.4;
    }

    static void Main()
    {

        TestMotorcycle moto = new TestMotorcycle();
        moto.StartEngine();
        moto.AddGas(15);
        var travelTime = moto.Drive(speed: 60, miles: 170);
        Console.WriteLine("Travel time: approx. {0} hours", travelTime);
    }
}
// The example displays the following output:
//      Travel time: approx. 3 hours
```

Vous pouvez appeler une méthode en utilisant à la fois des arguments positionnels et des arguments nommés. Toutefois, les arguments positionnels ne peuvent suivre les arguments nommés que lorsque les arguments nommés se trouvent dans des positions correctes. L'exemple suivant appelle la méthode `TestMotorcycle.Drive` de l'exemple précédent en utilisant un argument positionnel et un argument nommé.

C#

```
var travelTime = moto.Drive(170, speed: 55);
```

Méthodes hérités et remplacées

En plus des membres qui sont définis explicitement dans un type, un type hérite des membres définis dans ses classes de base. Comme tous les types du système de types managés héritent directement ou indirectement de la classe Object, tous les types héritent ses membres, comme Equals(Object), GetType() et ToString(). L'exemple suivant définit une classe `Person`, instancie deux objets `Person` et appelle la méthode `Person.Equals` pour déterminer si les deux objets sont égaux. La méthode `Equals` n'est cependant pas définie dans la classe `Person` ; elle est héritée de Object.

C#

```
using System;

public class Person
{
   public String FirstName;
}

public class ClassTypeExample
{
   public static void Main()
   {
      var p1 = new Person();
      p1.FirstName = "John";
      var p2 = new Person();
      p2.FirstName = "John";
      Console.WriteLine("p1 = p2: {0}", p1.Equals(p2));
   }
}
// The example displays the following output:
//      p1 = p2: False
```

Vous pouvez remplacer des membres hérités par des types en utilisant le mot clé `override` et en fournissant une implémentation de la méthode remplacée. La signature

de la méthode doit être identique à celle de la méthode remplacée. L'exemple suivant est semblable au précédent, sauf qu'il remplace la méthode Equals(Object). (Il remplace aussi la méthode GetHashCode(), car les deux méthodes sont destinées à fournir des résultats cohérents.)

C#

```
using System;

public class Person
{
    public String FirstName;

    public override bool Equals(object obj)
    {
        var p2 = obj as Person;
        if (p2 == null)
            return false;
        else
            return FirstName.Equals(p2.FirstName);
    }

    public override int GetHashCode()
    {
        return FirstName.GetHashCode();
    }
}

public class Example
{
    public static void Main()
    {
        var p1 = new Person();
        p1.FirstName = "John";
        var p2 = new Person();
        p2.FirstName = "John";
        Console.WriteLine("p1 = p2: {0}", p1.Equals(p2));
    }
}
// The example displays the following output:
//       p1 = p2: True
```

Passage de paramètres

Les types en C# sont des *types valeur* ou des *types référence*. Pour obtenir la liste des types valeur intégrés, consultez Types. Par défaut, les types valeur et les types référence sont passés par valeur à une méthode.

Passage de paramètres par valeur

Quand un type valeur est passé par valeur à une méthode, c'est une copie de l'objet, et non pas l'objet lui-même, qui est passée à la méthode. Par conséquent, les modifications de l'objet dans la méthode appelée n'ont pas d'effet sur l'objet d'origine quand le contrôle retourne à l'appelant.

L'exemple suivant passe un type valeur à une méthode par valeur, et la méthode appelée tente de changer la valeur du type valeur. Il définit une variable de type `int`, qui est un type valeur, initialise sa valeur à 20 et la passe à une méthode nommée `ModifyValue` qui change la valeur de la variable en 30. Quand la méthode retourne, la valeur de la variable reste cependant inchangée.

C#

```
using System;

public class ByValueExample
{
   public static void Main()
   {
      int value = 20;
      Console.WriteLine("In Main, value = {0}", value);
      ModifyValue(value);
      Console.WriteLine("Back in Main, value = {0}", value);
   }

   static void ModifyValue(int i)
   {
      i = 30;
      Console.WriteLine("In ModifyValue, parameter value = {0}", i);
      return;
   }
}
// The example displays the following output:
//      In Main, value = 20
//      In ModifyValue, parameter value = 30
//      Back in Main, value = 20
```

Quand un objet d'un type référence est passé par valeur à une méthode, c'est une référence à l'objet qui est passée par valeur. Autrement dit, la méthode ne reçoit pas l'objet lui-même mais un argument qui indique l'emplacement de l'objet. Si vous modifiez un membre de l'objet en utilisant cette référence, la modification est répercutée dans l'objet quand le contrôle retourne à la méthode appelante. Cependant, remplacer l'objet passé à la méthode n'a pas d'effet sur l'objet d'origine quand le contrôle retourne à l'appelant.

L'exemple suivant définit une classe (qui est un type référence) nommée `SampleRefType`. Il instancie un objet `SampleRefType`, affecte la valeur 44 à son champ `value` et passe l'objet à la méthode `ModifyObject`. L'exemple fait essentiellement la même chose que l'exemple précédent : il passe un argument par valeur à une méthode. Cependant, comme il utilise un type référence, le résultat est différent. La modification apportée dans `ModifyObject` au champ `obj.value` change également en 33 le champ `value` de l'argument `rt` dans la méthode `Main`, comme le montre le résultat de l'exemple.

C#

```
using System;

public class SampleRefType
{
    public int value;
}

public class ByRefTypeExample
{
    public static void Main()
    {
        var rt = new SampleRefType();
        rt.value = 44;
        ModifyObject(rt);
        Console.WriteLine(rt.value);
    }

    static void ModifyObject(SampleRefType obj)
    {
        obj.value = 33;
    }
}
```

Passage de paramètres par référence

On passe un paramètre par référence quand on souhaite changer la valeur d'un argument dans une méthode et refléter cette modification lorsque le contrôle revient à la méthode appelante. Pour passer un paramètre par référence, vous utilisez le mot clé ref ou out. Vous pouvez également passer une valeur par référence pour éviter la copie tout en empêchant les modifications à l'aide du mot clé in.

L'exemple suivant est identique au précédent, sauf que la valeur est passée par référence à la méthode `ModifyValue`. Quand la valeur du paramètre est modifiée dans la méthode `ModifyValue`, le changement de la valeur est reflété quand le contrôle retourne à l'appelant.

C#

```
using System;

public class ByRefExample
{
   public static void Main()
   {
      int value = 20;
      Console.WriteLine("In Main, value = {0}", value);
      ModifyValue(ref value);
      Console.WriteLine("Back in Main, value = {0}", value);
   }

   static void ModifyValue(ref int i)
   {
      i = 30;
      Console.WriteLine("In ModifyValue, parameter value = {0}", i);
      return;
   }
}
// The example displays the following output:
//      In Main, value = 20
//      In ModifyValue, parameter value = 30
//      Back in Main, value = 30
```

Un modèle courant qui utilise des paramètres par référence implique la permutation des valeurs des variables. Vous passez deux variables à une méthode par référence, et la méthode permute leur contenu. L'exemple suivant permute des valeurs d'entier.

C#

```
using System;

public class RefSwapExample
{
   static void Main()
   {
      int i = 2, j = 3;
      System.Console.WriteLine("i = {0}  j = {1}" , i, j);

      Swap(ref i, ref j);

      System.Console.WriteLine("i = {0}  j = {1}" , i, j);
   }

   static void Swap(ref int x, ref int y)
   {
      int temp = x;
      x = y;
      y = temp;
   }
```

```
}
// The example displays the following output:
//      i = 2  j = 3
//      i = 3  j = 2
```

Passer un paramètre de type référence vous permet de changer la valeur de la référence elle-même, au lieu de la valeur de ses éléments ou champs individuels.

Tableaux de paramètres

Parfois, l'obligation de spécifier le nombre exact d'arguments de votre méthode est restrictive. En utilisant le mot clé `params` pour indiquer qu'un paramètre est un tableau de paramètres, vous permettez que votre méthode soit appelée avec un nombre variable d'arguments. Le paramètre marqué avec le mot clé `params` doit être un type tableau. Ce doit être le dernier paramètre de la liste de paramètres de la méthode.

Un appelant peut alors appeler la méthode de quatre façons :

- En passant un tableau du type approprié, qui contient le nombre d'éléments souhaité.
- En passant une liste d'arguments individuels séparés par des virgules et du type approprié à la méthode.
- En passant `null`.
- En ne fournissant pas d'argument au tableau de paramètres.

L'exemple suivant définit une méthode nommée `GetVowels` qui retourne toutes les voyelles à partir d'un tableau de paramètres. La méthode `Main` montre les quatre façons d'appeler la méthode. Les appelants n'ont pas nécessairement à fournir tous les arguments pour les paramètres qui incluent le modificateur `params`. Dans ce cas, le paramètre est un tableau vide.

C#

```
using System;
using System.Linq;

class ParamsExample
{
    static void Main()
    {
        string fromArray = GetVowels(new[] { "apple", "banana", "pear" });
        Console.WriteLine($"Vowels from array: '{fromArray}'");

        string fromMultipleArguments = GetVowels("apple", "banana", "pear");
        Console.WriteLine($"Vowels from multiple arguments:
'{fromMultipleArguments}'");
```

```
        string fromNull = GetVowels(null);
        Console.WriteLine($"Vowels from null: '{fromNull}'");

        string fromNoValue = GetVowels();
        Console.WriteLine($"Vowels from no value: '{fromNoValue}'");
    }

    static string GetVowels(params string[] input)
    {
        if (input == null || input.Length == 0)
        {
            return string.Empty;
        }

        var vowels = new char[] { 'A', 'E', 'I', 'O', 'U' };
        return string.Concat(
            input.SelectMany(
                word => word.Where(letter =>
vowels.Contains(char.ToUpper(letter)))));
    }
}

// The example displays the following output:
//     Vowels from array: 'aeaaaea'
//     Vowels from multiple arguments: 'aeaaaea'
//     Vowels from null: ''
//     Vowels from no value: ''
```

Paramètres et arguments facultatifs

Une définition de méthode peut spécifier que ses paramètres sont obligatoires ou facultatifs. Par défaut, les paramètres sont obligatoires. Vous spécifiez des paramètres facultatifs en incluant la valeur par défaut du paramètre dans la définition de la méthode. Quand la méthode est appelée, si aucun argument n'est fourni pour un paramètre facultatif, c'est la valeur par défaut qui est utilisée à la place.

La valeur par défaut du paramètre doit être affectée par un des types d'expressions suivants :

- Une constante, comme une chaîne littérale ou un nombre.
- Une expression de la forme `default(SomeType)`, où `SomeType` peut être un type valeur ou un type référence. S'il s'agit d'un type référence, il s'agit de la même chose que de spécifier `null`. Vous pouvez utiliser le littéral `default`, car le compilateur peut déduire le type à partir de la déclaration du paramètre.

- Une expression de la forme `new ValType()`, où `ValType` est un type valeur. Ceci appelle le constructeur sans paramètre implicite du type valeur, qui n'est pas un membre réel du type.

> **Notes**
>
> Dans C# 10 et versions ultérieures, lorsqu'une expression du formulaire `new ValType()` appelle le constructeur sans paramètre explicitement défini d'un type valeur, le compilateur génère une erreur, car la valeur de paramètre par défaut doit être une constante de compilation. Utilisez l'expression `default(ValType)` ou le littéral `default` pour fournir la valeur de paramètre par défaut. Pour plus d'informations sur les constructeurs sans paramètre, consultez la section **Initialisation de struct et valeurs par défaut** de l'article **Types de structure**.

Si une méthode comprend à la fois des paramètres obligatoires et des paramètres facultatifs, les paramètres facultatifs sont définis à la fin de la liste des paramètres, après tous les paramètres obligatoires.

L'exemple suivant définit une méthode, `ExampleMethod`, qui a un paramètre obligatoire et deux paramètres facultatifs.

C#

```
using System;

public class Options
{
    public void ExampleMethod(int required, int optionalInt = default,
                              string? description = default)
   {
        var msg = $"{description ?? "N/A"}: {required} + {optionalInt} =
{required + optionalInt}";
        Console.WriteLine(msg);
   }
}
```

Si une méthode avec plusieurs arguments facultatifs est appelée en utilisant des arguments positionnels, l'appelant doit fournir un argument pour tous les paramètres facultatifs, du premier au dernier pour lequel un argument est fourni. Par exemple, dans le cas de la méthode `ExampleMethod`, si l'appelant fournit un argument pour le paramètre `description`, il doit également en fournir un pour le paramètre `optionalInt`. `opt.ExampleMethod(2, 2, "Addition of 2 and 2");` est un appel de méthode valide.

`opt.ExampleMethod(2, , "Addition of 2 and 0");` génère une erreur du compilateur « Argument manquant ».

Si une méthode est appelée en utilisant des arguments nommés ou une combinaison d'arguments positionnels et nommés, l'appelant peut omettre les arguments qui suivent le dernier argument positionnel dans l'appel de la méthode.

L'exemple suivant appelle la méthode `ExampleMethod` trois fois. Les deux premiers appels de la méthode utilisent des arguments positionnels. Le premier omet les deux arguments facultatifs, tandis que le deuxième omet le dernier argument. Le troisième appel de la méthode fournit un argument positionnel pour le paramètre obligatoire, mais utilise un argument nommé pour fournir une valeur au paramètre `description` tout en omettant l'argument `optionalInt`.

C#

```
public class OptionsExample
{
   public static void Main()
   {
      var opt = new Options();
      opt.ExampleMethod(10);
      opt.ExampleMethod(10, 2);
      opt.ExampleMethod(12, description: "Addition with zero:");
   }
}
// The example displays the following output:
//      N/A: 10 + 0 = 10
//      N/A: 10 + 2 = 12
//      Addition with zero:: 12 + 0 = 12
```

L'utilisation de paramètres facultatifs affecte la *résolution de la surcharge*, qui est la façon dont le compilateur C# détermine quelle surcharge particulière doit être appelée par un appel de méthode, comme suit :

- Une méthode, un indexeur ou un constructeur est un candidat pour l'exécution si chacun de ses paramètres est facultatif ou correspond, par nom ou par position, à un seul argument dans l'instruction appelante, et que cet argument peut être converti vers le type du paramètre.
- Si plusieurs candidats sont trouvés, les règles de résolution de surcharge des conversions préférées sont appliquées aux arguments qui sont explicitement spécifiés. Les arguments omis pour les paramètres facultatifs sont ignorés.
- Si deux candidats sont jugés de qualité équivalente, la préférence va à celui qui n'a pas de paramètres facultatifs pour lesquels des arguments ont été omis dans

l'appel. Ceci est une conséquence d'une préférence générale dans la résolution de la surcharge pour les candidats qui ont le moins de paramètres.

Valeurs de retour

Les méthodes peuvent retourner une valeur à l'appelant. Si le type de retour (le type qui figure avant le nom de la méthode) n'est pas `void`, la méthode peut retourner la valeur en utilisant le mot clé `return`. Une instruction avec le mot clé `return` suivi d'une variable, d'une constante ou d'une expression qui correspond au type de retour retourne cette valeur à l'appelant de la méthode. Les méthodes dotées d'un type de retour non void doivent utiliser le mot clé `return` pour retourner une valeur. Le mot clé `return` arrête également l'exécution de la méthode.

Si le type de retour est `void`, une instruction `return` sans valeur est quand même utile pour arrêter l'exécution de la méthode. Sans le mot clé `return` , la méthode arrête de s'exécuter quand elle atteint la fin du bloc de code.

Par exemple, ces deux méthodes utilisent le mot clé `return` pour retourner des entiers :

C#

```
class SimpleMath
{
    public int AddTwoNumbers(int number1, int number2)
    {
        return number1 + number2;
    }

    public int SquareANumber(int number)
    {
        return number * number;
    }
}
```

Pour utiliser une valeur retournée à partir d'une méthode, la méthode d'appel peut utiliser l'appel de méthode proprement dit partout où une valeur du même type peut suffire. Vous pouvez également affecter la valeur de retour à une variable. Par exemple, les deux exemples de code suivants remplissent le même objectif :

C#

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

C#

```
result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));
// The result is 9.
Console.WriteLine(result);
```

L'utilisation d'une variable locale, dans cet exemple, `result`, pour stocker une valeur est facultative. Elle peut favoriser la lisibilité du code ou s'avérer nécessaire si vous avez besoin de stocker la valeur d'origine de l'argument pour la portée entière de la méthode.

Vous voulez parfois que votre méthode retourne plusieurs valeurs. Vous pouvez faire cela facilement en utilisant des *types tuple* et des *littéraux de tuple*. Le type tuple définit les types de données des éléments du tuple. Les littéraux de tuple fournissent les valeurs réelles du tuple retourné. Dans l'exemple suivant, `(string, string, string, int)` définit le type tuple retourné par la méthode `GetPersonalInfo`. L'expression `(per.FirstName, per.MiddleName, per.LastName, per.Age)` est le littéral de tuple ; la méthode retourne le prénom, le deuxième prénom et le nom, ainsi que l'âge pour un objet `PersonInfo`.

C#

```
public (string, string, string, int) GetPersonalInfo(string id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}
```

L'appelant peut alors consommer le tuple retourné avec du code comme celui-ci :

C#

```
var person = GetPersonalInfo("111111111");
Console.WriteLine($"{person.Item1} {person.Item3}: age = {person.Item4}");
```

Vous pouvez aussi affecter des noms aux éléments du tuple dans la définition du type tuple. L'exemple suivant montre une autre version de la méthode `GetPersonalInfo` qui utilise des éléments nommés :

C#

```
public (string FName, string MName, string LName, int Age)
GetPersonalInfo(string id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
```

```
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}
```

L'appel précédent de la méthode `GetPersonalInfo` peut alors être modifié comme suit :

C#

```
var person = GetPersonalInfo("111111111");
Console.WriteLine($"{person.FName} {person.LName}: age = {person.Age}");
```

Si un tableau est passé comme argument à une méthode et que celle-ci modifie la valeur d'éléments individuels, il n'est pas nécessaire que la méthode retourne le tableau, même si vous pouvez choisir de le faire pour la qualité du style ou pour obtenir un flux fonctionnel de valeurs. La raison en est que C# passe tous les types référence par valeur et que la valeur d'une référence de tableau est le pointeur vers ce tableau. Dans l'exemple suivant, les modifications apportées au contenu du tableau `values` qui sont effectuées dans la méthode `DoubleValues` sont observables par tout code ayant une référence au tableau.

C#

```
using System;

public class ArrayValueExample
{
   static void Main(string[] args)
   {
      int[] values = { 2, 4, 6, 8 };
      DoubleValues(values);
      foreach (var value in values)
         Console.Write("{0}  ", value);
   }

   public static void DoubleValues(int[] arr)
   {
      for (int ctr = 0; ctr <= arr.GetUpperBound(0); ctr++)
         arr[ctr] = arr[ctr] * 2;
   }
}
// The example displays the following output:
//       4  8  12  16
```

Méthodes d'extension

En règle générale, il existe deux façons d'ajouter une méthode à un type existant :

- Modifier le code source pour ce type. Vous ne pouvez évidemment pas le faire si vous ne disposez pas du code source du type. D'autre part, ceci devient une modification d'importance si vous ajoutez aussi des champs de données privés pour prendre en charge la méthode.
- Définir la nouvelle méthode dans une classe dérivée. Vous ne pouvez pas ajouter une méthode de cette façon en utilisant l'héritage pour d'autres types, comme des structures et des énumérations. Vous ne pouvez pas non plus l'utiliser pour « ajouter » une méthode à une classe scellée.

Les méthodes d'extension vous permettent d'« ajouter » une méthode à un type existant sans modifier le type lui-même, ou en implémentant la nouvelle méthode dans un type hérité. La méthode d'extension ne doit pas non plus nécessairement se trouver dans le même assembly que le type qu'elle étend. Vous appelez une méthode d'extension comme si elle était un membre défini d'un type.

Pour plus d'informations, consultez Méthodes d'extension.

Méthodes async

La fonctionnalité async vous permet d'appeler des méthodes asynchrones sans utiliser de rappels explicites ni fractionner manuellement votre code entre plusieurs méthodes ou expressions lambda.

Si vous marquez une méthode avec le modificateur async, vous pouvez utiliser l'opérateur await dans la méthode. Quand le contrôle atteint une expression `await` dans la méthode async, il retourne à l'appelant si la tâche attendue n'est pas terminée, et la progression dans la méthode avec le mot clé `await` est interrompue jusqu'à ce que la tâche attendue se termine. Quand la tâche est terminée, l'exécution peut reprendre dans la méthode.

> ⓘ **Notes**
>
> Une méthode asynchrone retourne à l'appelant quand elle rencontre le premier objet attendu qui n'est pas encore terminé ou quand elle atteint la fin de la méthode asynchrone, selon la première éventualité.

Une méthode asynchrone a généralement un type de retour Task<TResult>, Task, IAsyncEnumerable<T> ou `void` . Le type de retour `void` est essentiellement utilisé pour définir les gestionnaires d'événements, où un type de retour `void` est obligatoire. Une méthode async qui retourne `void` ne peut pas être attendue, et l'appelant d'une

méthode retournant void ne peut intercepter aucune exception levée par la méthode. Une méthode asynchrone peut avoir n'importe quel type de retour de type tâche.

Dans l'exemple suivant, `DelayAsync` est une méthode async contenant une instruction return qui retourne un entier. Comme il s'agit d'une méthode async, la déclaration de sa méthode doit avoir un type de retour `Task<int>`. Comme le type de retour est `Task<int>`, l'évaluation de l'expression `await` dans `DoSomethingAsync` produit un entier, comme l'instruction `int result = await delayTask` suivante le montre.

C#

```
class Program
{
    static Task Main() => DoSomethingAsync();

    static async Task DoSomethingAsync()
    {
        Task<int> delayTask = DelayAsync();
        int result = await delayTask;

        // The previous two statements may be combined into
        // the following statement.
        //int result = await DelayAsync();

        Console.WriteLine($"Result: {result}");
    }

    static async Task<int> DelayAsync()
    {
        await Task.Delay(100);
        return 5;
    }
}
// Example output:
//   Result: 5
```

Une méthode async ne peut pas déclarer de paramètres in, ref ou out, mais elle peut appeler des méthodes qui ont ces paramètres.

Pour plus d'informations sur les méthodes asynchrones, consultez Programmation asynchrone avec asynchrone et await et Types de retour asynchrones.

Membres expression-bodied

Il est courant d'avoir des définitions de méthode qui retournent tout simplement le résultat d'une expression immédiatement, ou qui ont une seule instruction en tant que

corps de la méthode. Il existe un raccourci de syntaxe pour définir de telles méthodes en utilisant `=>` :

```
C#
```

```
public Point Move(int dx, int dy) => new Point(x + dx, y + dy);
public void Print() => Console.WriteLine(First + " " + Last);
// Works with operators, properties, and indexers too.
public static Complex operator +(Complex a, Complex b) => a.Add(b);
public string Name => First + " " + Last;
public Customer this[long id] => store.LookupCustomer(id);
```

Si la méthode retourne `void` ou est une méthode async, alors le corps de la méthode doit être une expression d'instruction (comme avec les expressions lambda). En ce qui concerne les propriétés et indexeurs, ils doivent être en lecture seule et vous n'utilisez pas le mot clé d'accesseur `get`.

Iterators

Un itérateur exécute une itération personnalisée sur une collection, comme une liste ou un tableau. Un itérateur utilise l'instruction yield return pour retourner chaque élément un par un. Quand une instruction `yield return` est atteinte, l'emplacement actif est mémorisé pour que l'appelant puisse demander l'élément suivant dans la séquence.

Le type de retour d'un itérateur peut être IEnumerable, IEnumerable<T>, IAsyncEnumerable<T>, IEnumerator ou IEnumerator<T>.

Pour plus d'informations, consultez Itérateurs.

Voir aussi

- Modificateurs d'accès
- Classes statiques et membres de classe statique
- Héritage
- Classes abstract et sealed et membres de classe
- params
- out
- ref
- in
- Passage de paramètres

Propriétés

Article • 07/04/2023

Les propriétés sont des éléments de première classe dans C#. Le langage définit la syntaxe que les développeurs utilisent pour écrire du code qui exprime leur intention de conception avec précision.

Les propriétés auxquelles le code accède se comportent comme des champs. Toutefois, contrairement aux champs, les propriétés sont implémentées avec des accesseurs qui définissent quelles instructions sont exécutées au moment de l'accès à une propriété ou de son assignation.

Syntaxe des propriétés

La syntaxe des propriétés est une extension naturelle des champs. Un champ définit un emplacement de stockage :

C#

```
public class Person
{
    public string? FirstName;

    // Omitted for brevity.
}
```

Une définition de propriété contient les déclarations de l'accesseur `get`, qui récupère la valeur de cette propriété, et de l'accesseur `set`, qui assigne cette valeur :

C#

```
public class Person
{
    public string? FirstName { get; set; }

    // Omitted for brevity.
}
```

La syntaxe illustrée ci-dessus est la syntaxe *auto property*. Le compilateur génère l'emplacement de stockage pour le champ qui enregistre la propriété. Le compilateur implémente également le corps des accesseurs `get` et `set`.

Parfois, vous devez initialiser une propriété sur une valeur autre que la valeur par défaut pour son type. C# permet cette opération en définissant une valeur après l'accolade fermante de la propriété. Vous pouvez choisir comme valeur initiale pour la propriété `FirstName` une chaîne vide au lieu de `null`. Vous pouvez le spécifier comme indiqué ci-dessous :

C#

```
public class Person
{
    public string FirstName { get; set; } = string.Empty;

    // Omitted for brevity.
}
```

Une initialisation spécifique est pratique surtout pour les propriétés en lecture seule, comme vous le verrez plus loin dans cet article.

Vous pouvez aussi définir le stockage vous-même, de la manière suivante :

C#

```
public class Person
{
    public string? FirstName
    {
        get { return _firstName; }
        set { _firstName = value; }
    }
    private string? _firstName;

    // Omitted for brevity.
}
```

Quand une implémentation de propriété est une expression unique, vous pouvez utiliser des *membres expression-bodied* pour l'accesseur Get ou Set :

C#

```
public class Person
{
    public string? FirstName
    {
        get => _firstName;
        set => _firstName = value;
    }
    private string? _firstName;
```

```
    // Omitted for brevity.
}
```

Cette syntaxe simplifiée est utilisée partout où elle est applicable dans cet article.

La définition de propriété présentée ci-dessus est une propriété en lecture-écriture. Notez la présence du mot clé `value` dans l'accesseur set. L'accesseur `set` a toujours un seul paramètre nommé `value`. L'accesseur `get` doit retourner une valeur convertible dans le type de la propriété (`string`, dans cet exemple).

Nous venons de voir les éléments de base de la syntaxe. Il existe beaucoup de variantes de cette syntaxe, qui sont adaptées aux divers idiomes de conception. Nous allons les explorer et découvrir les options syntaxiques de chacune.

Validation

Les exemples ci-dessus ont montré un des cas les plus simples de définition de propriété, à savoir une propriété en lecture-écriture sans validation. En écrivant le code souhaité dans les accesseurs `get` et `set`, vous pouvez créer de nombreux scénarios différents.

Vous pouvez écrire du code dans l'accesseur `set` pour garantir que les valeurs représentées par une propriété sont toujours valides. Par exemple, définissez une règle pour la classe `Person` qui spécifie que le nom ne peut pas être vide, ni contenir d'espace blanc. Le code à écrire est le suivant :

C#

```
public class Person
{
    public string? FirstName
    {
        get => _firstName;
        set
        {
            if (string.IsNullOrWhiteSpace(value))
                throw new ArgumentException("First name must not be blank");
            _firstName = value;
        }
    }
    private string? _firstName;

    // Omitted for brevity.
}
```

L'exemple précédent peut être simplifié en utilisant une expression `throw` dans le cadre de la validation de la méthode setter de propriété :

C#

```
public class Person
{
    public string? FirstName
    {
        get => _firstName;
        set => _firstName = (!string.IsNullOrWhiteSpace(value)) ? value :
throw new ArgumentException("First name must not be blank");
    }
    private string? _firstName;

    // Omitted for brevity.
}
```

L'exemple ci-dessus applique la règle selon laquelle le nom ne doit pas être vide, ni contenir d'espace blanc. Supposons qu'un développeur écrive cette ligne de code :

C#

```
hero.FirstName = "";
```

Cette assignation lève une exception `ArgumentException`. Étant donné qu'un accesseur set de propriété doit avoir un type de retour void, vous signalez les erreurs dans l'accesseur set en levant une exception.

Vous pouvez employer cette même syntaxe pour valider d'autres éléments dans votre scénario. Vous pouvez notamment vérifier les relations entre plusieurs propriétés ou effectuer une validation par rapport à des conditions externes. Toute instruction C# valide peut être utilisée dans un accesseur de propriété.

Contrôle d'accès

Jusqu'ici, nous avons vu uniquement des définitions de propriétés qui sont en lecture-écriture dans des accesseurs publics. Ce n'est pas la seule accessibilité valide pour les propriétés. Vous pouvez créer des propriétés en lecture seule, ou assigner une accessibilité différente aux accesseurs set et get. Supposons que votre classe `Person` doit uniquement autoriser la modification de la valeur de la propriété `FirstName` à partir des autres méthodes de cette classe. Vous pouvez alors assigner l'accessibilité `private` au lieu de `public` à l'accesseur set :

C#

```
public class Person
{
    public string? FirstName { get; private set; }

    // Omitted for brevity.
}
```

À présent, la propriété `FirstName` est accessible à partir de n'importe quel code, mais elle peut uniquement être assignée à partir de code dans la classe `Person`.

Vous pouvez ajouter n'importe quel modificateur d'accès restrictif à l'accesseur set ou get. Le modificateur d'accès que vous ajoutez à un accesseur doit être plus restrictif que le modificateur d'accès spécifié dans la définition de propriété. Le code ci-dessus est autorisé, car la propriété `FirstName` est `public`, mais l'accesseur set est `private`. En revanche, vous ne pouvez pas déclarer une propriété `private` avec un accesseur `public`. Les propriétés peuvent également être déclarées comme `protected`, `internal`, `protected internal` ou même `private`.

Placer le modificateur le plus restrictif sur l'accesseur `get` est également autorisé. Par exemple, vous pouvez avoir une propriété `public`, mais restreindre l'accesseur `get` à `private`. Ce scénario s'observe rarement dans la pratique.

Lecture seule

Vous pouvez également limiter les modifications apportées à une propriété pour qu'elle puisse uniquement être définie dans un constructeur. Vous pouvez modifier la classe `Person`, comme suit :

C#

```
public class Person
{
    public Person(string firstName) => FirstName = firstName;

    public string FirstName { get; }

    // Omitted for brevity.
}
```

Init uniquement

L'exemple précédent exige que les appelants utilisent le constructeur qui inclut le paramètre `FirstName`. Les appelants ne peuvent pas utiliser d'initialiseurs d'objet pour affecter une valeur à la propriété. Pour prendre en charge les initialiseurs, vous pouvez faire de l'accesseur `set` un accesseur `init`, comme indiqué dans le code suivant :

C#

```
public class Person
{
    public Person() { }
    public Person(string firstName) => FirstName = firstName;

    public string? FirstName { get; init; }

    // Omitted for brevity.
}
```

L'exemple précédent permet à un appelant de créer un `Person` à l'aide du constructeur par défaut, même lorsque ce code ne définit pas la propriété `FirstName`. À compter de C# 11, vous pouvez *demander* aux appelants de définir cette propriété :

C#

```
public class Person
{
    public Person() { }

    [SetsRequiredMembers]
    public Person(string firstName) => FirstName = firstName;

    public required string FirstName { get; init; }

    // Omitted for brevity.
}
```

Le code précédent ajoute deux ajouts à la classe `Person`. Tout d'abord, la déclaration de propriété `FirstName` inclut le modificateur `required`. Cela signifie que tout code qui crée un nouveau `Person` doit définir cette propriété. Deuxièmement, le constructeur qui prend un paramètre `firstName` a l'attribut System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute. Cet attribut informe le compilateur que ce constructeur définit *tous* `required` les membres.

> ⓘ **Important**

Ne confondez pas `required` avec *non-nullable*. Définir une `required` propriété sur `null` ou `default` est valide. Si le type est non-nullable, comme `string` dans ces exemples, le compilateur émet un avertissement.

Les appelants doivent soit utiliser le constructeur avec `SetsRequiredMembers`, soit définir la propriété à l'aide `FirstName` d'un initialiseur d'objet, comme indiqué dans le code suivant :

C#

```
var person = new VersionNinePoint2.Person("John");
person = new VersionNinePoint2.Person{ FirstName = "John"};
// Error CS9035: Required member `Person.FirstName` must be set:
//person = new VersionNinePoint2.Person();
```

Propriétés calculées

Une propriété peut faire plus que simplement retourner la valeur d'un champ de membre. Vous pouvez créer des propriétés qui retournent une valeur calculée. L'objet `Person` est étendu pour retourner le nom complet, calculé en concaténant le nom et le prénom :

C#

```
public class Person
{
    public string? FirstName { get; set; }

    public string? LastName { get; set; }

    public string FullName { get { return $"{FirstName} {LastName}"; } }
}
```

L'exemple ci-dessus utilise la fonctionnalité d'interpolation de chaîne pour créer la chaîne mise en forme du nom complet.

Vous pouvez également utiliser un *membre expression-bodied*, qui constitue un moyen plus succinct de créer la propriété `FullName` calculée :

C#

```
public class Person
{
    public string? FirstName { get; set; }
```

```
    public string? LastName { get; set; }

    public string FullName => $"{FirstName} {LastName}";
}
```

Les *membres expression-bodied* utilisent la syntaxe des *expressions lambda* pour définir des méthodes qui contiennent une seule expression. Ici, cette expression retourne le nom complet de l'objet person.

Propriétés évaluées avec mise en cache

Vous pouvez combiner le concept d'une propriété calculée avec le stockage et créer une *propriété évaluée avec mise en cache*. Par exemple, vous pouvez mettre à jour la propriété `FullName` pour que la chaîne soit mise en forme uniquement lors du premier accès à cette propriété :

C#

```
public class Person
{
    public string? FirstName { get; set; }

    public string? LastName { get; set; }

    private string? _fullName;
    public string FullName
    {
        get
        {
            if (_fullName is null)
                _fullName = $"{FirstName} {LastName}";
            return _fullName;
        }
    }
}
```

Le code ci-dessus contient toutefois un bogue. Si le code met à jour la valeur de la propriété `FirstName` ou `LastName`, le champ `fullName` qui a été précédemment évalué n'est plus valide. Vous modifiez les accesseurs `set` des propriétés `FirstName` et `LastName` pour que le champ `fullName` soit recalculé :

C#

```
public class Person
{
    private string? _firstName;
```

```
    public string? FirstName
    {
        get => _firstName;
        set
        {
            _firstName = value;
            _fullName = null;
        }
    }

    private string? _lastName;
    public string? LastName
    {
        get => _lastName;
        set
        {
            _lastName = value;
            _fullName = null;
        }
    }

    private string? _fullName;
    public string FullName
    {
        get
        {
            if (_fullName is null)
                _fullName = $"{FirstName} {LastName}";
            return _fullName;
        }
    }
}
```

Dans cette version finale du code, la propriété `FullName` est évaluée uniquement si cela est nécessaire. Si la version précédemment calculée est valide, elle est utilisée. Si elle n'est plus valide en raison d'un changement d'état, la version est recalculée. Les développeurs peuvent utiliser cette classe sans connaître les détails de l'implémentation. Ces modifications internes n'ont pas d'impact sur l'utilisation de l'objet Person. C'est l'un des principaux avantages d'utiliser des propriétés pour exposer les membres de données d'un objet.

Attachement d'attributs à des propriétés implémentées automatiquement

Les attributs de champ peuvent être attachés au champ de stockage généré par le compilateur dans les propriétés implémentées automatiquement. Par exemple, considérez une révision de la classe `Person` qui ajoute une propriété `Id` unique de type

entier. Vous écrivez la propriété `Id` en utilisant une propriété implémentée automatiquement, mais votre conception n'effectue pas d'appel pour le stockage de la propriété `Id`. NonSerializedAttribute peut être attaché seulement à des champs, et pas à des propriétés. Vous pouvez attacher NonSerializedAttribute au champ de stockage pour la propriété `Id` en utilisant le spécificateur `field:` sur l'attribut, comme illustré dans l'exemple suivant :

C#

```
public class Person
{
    public string? FirstName { get; set; }

    public string? LastName { get; set; }

    [field:NonSerialized]
    public int Id { get; set; }

    public string FullName => $"{FirstName} {LastName}";
}
```

Cette technique fonctionne pour tout attribut que vous voulez attacher au champ de stockage sur la propriété implémentée automatiquement.

Implémentation de INotifyPropertyChanged

Il existe un dernier scénario où vous devrez écrire du code dans un accesseur de propriété : pour prendre en charge l'interface INotifyPropertyChanged, qui notifie les changements de valeurs aux clients de liaison de données. Quand la valeur d'une propriété change, l'objet déclenche l'événement INotifyPropertyChanged.PropertyChanged pour signaler le changement. Les bibliothèques de liaison de données mettent ensuite à jour les éléments d'affichage en fonction de cette modification. Le code ci-dessous montre comment implémenter `INotifyPropertyChanged` pour la propriété `FirstName` de la classe Person.

C#

```
public class Person : INotifyPropertyChanged
{
    public string? FirstName
    {
        get => _firstName;
        set
        {
            if (string.IsNullOrWhiteSpace(value))
                throw new ArgumentException("First name must not be blank");
```

```
            if (value != _firstName)
            {
                _firstName = value;
                PropertyChanged?.Invoke(this,
                    new PropertyChangedEventArgs(nameof(FirstName)));
            }
        }
    }
    private string? _firstName;

    public event PropertyChangedEventHandler? PropertyChanged;
}
```

L'opérateur `?.` est appelé *opérateur conditionnel Null*. Il recherche une référence null avant d'évaluer le côté droit de l'opérateur. Au final, s'il n'y a pas d'abonné à l'événement `PropertyChanged`, le code devant déclencher l'événement n'est pas exécuté. Dans ce cas précis, il lèverait une exception `NullReferenceException` sans cette vérification. Pour plus d'informations, consultez events. Cet exemple utilise également le nouvel opérateur `nameof` pour convertir le symbole de nom de propriété en sa représentation textuelle. L'utilisation de `nameof` peut vous éviter des erreurs dues à la saisie incorrecte du nom de propriété.

L'implémentation de INotifyPropertyChanged est un autre exemple de cas où vous pouvez écrire du code dans vos accesseurs pour prendre en charge les scénarios souhaités.

Résumé

Les propriétés sont une forme de champs intelligents dans une classe ou un objet. De l'extérieur de l'objet, elles apparaissent sous la forme de champs dans l'objet. Toutefois, les propriétés peuvent être implémentées avec toutes les fonctionnalités C#. Vous pouvez écrire du code qui remplit les exigences de validation, d'accessibilité, d'évaluation différée ou toute autre exigence requise dans vos scénarios.

Indexeurs

Article • 10/05/2023

Les *indexeurs* sont similaires aux propriétés. Les indexeurs sont souvent construits à l'aide des mêmes fonctionnalités de langage que les propriétés. Il permettent l'utilisation de propriétés *indexées*, qui sont référencées à l'aide d'un ou plusieurs arguments. Ces arguments fournissent un index dans une collection de valeurs.

Syntaxe de l'indexeur

Vous pouvez accéder à un indexeur avec un nom de variable et des crochets. Vous devez placer les arguments de l'indexeur entre crochets :

C#

```
var item = someObject["key"];
someObject["AnotherKey"] = item;
```

Déclarez les indexeurs en utilisant le mot clé `this` comme nom de propriété et en déclarant les arguments entre crochets. Cette déclaration correspond à l'utilisation abordée dans le paragraphe précédent :

C#

```
public int this[string key]
{
    get { return storage.Find(key); }
    set { storage.SetAt(key, value); }
}
```

Dans ce premier exemple, vous pouvez voir la relation entre la syntaxe des propriétés et celle des indexeurs. Cette analogie s'applique à la plupart des règles de syntaxe des indexeurs. Les indexeurs peuvent avoir n'importe quel modificateur d'accès valide (public, protected internal, protected, internal, private ou private protected). Ils peuvent être sealed, virtual ou abstract. Comme pour les propriétés, vous pouvez spécifier des modificateurs d'accès différents pour les accesseurs get et set d'un indexeur. Vous pouvez également spécifier des indexeurs en lecture seule (en omettant l'accesseur set) ou des indexeurs en écriture seule (en omettant l'accesseur get).

Vous pouvez appliquer aux indexeurs quasiment tout ce que vous avez appris de l'utilisation des propriétés. La seule exception à cette règle sont les *propriétés*

implémentées automatiquement. Le compilateur ne peut pas toujours générer le stockage adapté à l'indexeur.

C'est la présence d'arguments référençant un élément dans un ensemble d'éléments qui distingue les indexeurs des propriétés. Vous pouvez définir plusieurs indexeurs sur un type, tant que chaque indexeur a sa propre liste d'arguments. Nous allons explorer différents scénarios où vous pourrez utiliser un ou plusieurs indexeurs dans une définition de classe.

Scénarios

Vous définissez des *indexeurs* dans votre type quand son API modélise une collection dans laquelle vous définissez des arguments. Vos indexeurs peuvent ou non être mappés directement aux types de collection qui font partie du framework .NET Core. Votre type peut avoir d'autres responsabilités en plus de la modélisation d'une collection. Les indexeurs vous permettent de fournir l'API qui correspond à l'abstraction de votre type sans exposer les détails internes concernant le stockage ou le calcul des valeurs de cette abstraction.

Examinons quelques-uns des scénarios courants d'utilisation des *indexeurs*. Vous pouvez accéder au dossier d'exemples d'indexeurs. Pour obtenir des instructions de téléchargement, consultez Exemples et didacticiels.

Tableaux et vecteurs

L'un des scénarios de création d'indexeur les plus courants est lorsque votre type modélise un tableau ou un vecteur. Vous pouvez créer un indexeur pour modéliser une liste de données triées.

L'avantage de créer votre propre indexeur est que vous pouvez définir le stockage de cette collection en fonction de vos besoins. Imaginez un scénario où votre type modélise des données d'historique qui sont trop volumineuses pour être chargées en une seule fois dans la mémoire. Vous devez charger et décharger des sections de la collection selon leur utilisation. L'exemple suivant modélise ce comportement. Il signale le nombre de points de données. Il crée à la demande des pages contenant des sections de données. Il supprime des pages de la mémoire afin de libérer de l'espace pour les pages qui ont fait l'objet de demandes récentes.

```
C#

public class DataSamples
{
```

```
    private class Page
    {
        private readonly List<Measurements> pageData = new
List<Measurements>();
        private readonly int startingIndex;
        private readonly int length;
        private bool dirty;
        private DateTime lastAccess;

        public Page(int startingIndex, int length)
        {
            this.startingIndex = startingIndex;
            this.length = length;
            lastAccess = DateTime.Now;

            // This stays as random stuff:
            var generator = new Random();
            for(int i=0; i < length; i++)
            {
                var m = new Measurements
                {
                    HiTemp = generator.Next(50, 95),
                    LoTemp = generator.Next(12, 49),
                    AirPressure = 28.0 + generator.NextDouble() * 4
                };
                pageData.Add(m);
            }
        }
        public bool HasItem(int index) =>
            ((index >= startingIndex) &&
            (index < startingIndex + length));

        public Measurements this[int index]
        {
            get
            {
                lastAccess = DateTime.Now;
                return pageData[index - startingIndex];
            }
            set
            {
                pageData[index - startingIndex] = value;
                dirty = true;
                lastAccess = DateTime.Now;
            }
        }

        public bool Dirty => dirty;
        public DateTime LastAccess => lastAccess;
    }

    private readonly int totalSize;
    private readonly List<Page> pagesInMemory = new List<Page>();

    public DataSamples(int totalSize)
```

```
    {
        this.totalSize = totalSize;
    }

    public Measurements this[int index]
    {
        get
        {
            if (index < 0)
                throw new IndexOutOfRangeException("Cannot index less than
0");
            if (index >= totalSize)
                throw new IndexOutOfRangeException("Cannot index past the
end of storage");

            var page = updateCachedPagesForAccess(index);
            return page[index];
        }
        set
        {
            if (index < 0)
                throw new IndexOutOfRangeException("Cannot index less than
0");
            if (index >= totalSize)
                throw new IndexOutOfRangeException("Cannot index past the
end of storage");
            var page = updateCachedPagesForAccess(index);

            page[index] = value;
        }
    }

    private Page updateCachedPagesForAccess(int index)
    {
        foreach (var p in pagesInMemory)
        {
            if (p.HasItem(index))
            {
                return p;
            }
        }
        var startingIndex = (index / 1000) * 1000;
        var newPage = new Page(startingIndex, 1000);
        addPageToCache(newPage);
        return newPage;
    }

    private void addPageToCache(Page p)
    {
        if (pagesInMemory.Count > 4)
        {
            // remove oldest non-dirty page:
            var oldest = pagesInMemory
                .Where(page => !page.Dirty)
                .OrderBy(page => page.LastAccess)
```

```
                .FirstOrDefault();
            // Note that this may keep more than 5 pages in memory
            // if too much is dirty
            if (oldest != null)
                pagesInMemory.Remove(oldest);
        }
        pagesInMemory.Add(p);
    }
}
```

Vous pouvez suivre cet idiome de conception afin de modéliser toute sorte de collection pour laquelle il existe de bonnes raisons de ne pas charger l'intégralité des données en mémoire. Notez que la classe `Page` est une classe privée imbriquée qui ne fait pas partie de l'interface publique. Ces détails sont masqués pour tous les utilisateurs de cette classe.

Dictionnaires

Un autre scénario courant est lorsque vous avez besoin de modéliser un dictionnaire ou un mappage. Dans ce cas, votre type stocke des valeurs en fonction des clés, généralement des clés de texte. Cet exemple crée un dictionnaire qui mappe les arguments de ligne de commande à des expressions lambda qui gèrent ces options. L'exemple suivant montre deux classes : une classe `ArgsActions` qui mappe une option de ligne de commande à un délégué `Action`, et un `ArgsProcessor` qui utilise `ArgsActions` pour exécuter chaque `Action` quand il rencontre cette option.

C#

```
public class ArgsProcessor
{
    private readonly ArgsActions actions;

    public ArgsProcessor(ArgsActions actions)
    {
        this.actions = actions;
    }

    public void Process(string[] args)
    {
        foreach(var arg in args)
        {
            actions[arg]?.Invoke();
        }
    }

}
public class ArgsActions
{
```

```
    readonly private Dictionary<string, Action> argsActions = new
Dictionary<string, Action>();

    public Action this[string s]
    {
        get
        {
            Action action;
            Action defaultAction = () => {} ;
            return argsActions.TryGetValue(s, out action) ? action :
defaultAction;
        }
    }

    public void SetOption(string s, Action a)
    {
        argsActions[s] = a;
    }
}
```

Dans cet exemple, la collection `ArgsAction` correspond étroitement à la collection sous-jacente. `get` détermine si une option donnée a été configurée. Si c'est le cas, il retourne le `Action` associé à cette option. Sinon, il retourne un `Action` qui n'a aucun effet. L'accesseur public n'inclut pas d'accesseur `set`, Au lieu de cela, il comprend la conception qui utilise une méthode publique pour définir des options.

Mappages multidimensionnels

Vous pouvez créer des indexeurs qui utilisent plusieurs arguments. En outre, ces arguments ne sont pas contraints à être du même type. Examinons ces deux exemples.

Le premier exemple montre une classe qui génère des valeurs pour un ensemble de Mandelbrot. Pour plus d'informations sur les mathématiques impliquées par cet ensemble, lisez cet article. L'indexeur utilise deux doubles pour définir un point dans le plan X, Y. L'accesseur get calcule le nombre d'itérations jusqu'à un point déterminé pour ne pas se trouver dans l'ensemble. Si le nombre maximal d'itérations est atteint, le point se trouve dans l'ensemble, et la valeur maxIterations de la classe est retournée. (Les images générées par ordinateur popularisées par l'ensemble de Mandelbrot définissent des couleurs pour le nombre d'itérations qui sont nécessaires pour déterminer qu'un point se trouve en dehors de l'ensemble.)

C#

```
public class Mandelbrot
{
    readonly private int maxIterations;
```

```
    public Mandelbrot(int maxIterations)
    {
        this.maxIterations = maxIterations;
    }

    public int this [double x, double y]
    {
        get
        {
            var iterations = 0;
            var x0 = x;
            var y0 = y;

            while ((x*x + y * y < 4) &&
                (iterations < maxIterations))
            {
                var newX = x * x - y * y + x0;
                y = 2 * x * y + y0;
                x = newX;
                iterations++;
            }
            return iterations;
        }
    }
}
```

L'ensemble de Mandelbrot définit des valeurs à chaque coordonnée (x, y) pour des valeurs de nombres réels. Cela définit un dictionnaire qui peut contenir un nombre infini de valeurs. Par conséquent, aucun stockage n'est nécessaire. Cette classe calcule la valeur de chaque point lorsque le code appelle l'accesseur `get`. Aucun stockage sous-jacent n'est utilisé.

Examinons une dernière utilisation d'indexeur, dans laquelle l'indexeur accepte plusieurs arguments de types différents. Prenons un programme qui gère des données d'historique des températures. Cet indexeur utilise une ville et une date pour définir ou obtenir les températures minimales et maximales de la ville en question :

C#

```
using DateMeasurements =
    System.Collections.Generic.Dictionary<System.DateTime,
IndexersSamples.Common.Measurements>;
using CityDataMeasurements =
    System.Collections.Generic.Dictionary<string,
System.Collections.Generic.Dictionary<System.DateTime,
IndexersSamples.Common.Measurements>>;

public class HistoricalWeatherData
{
    readonly CityDataMeasurements storage = new CityDataMeasurements();
```

```
    public Measurements this[string city, DateTime date]
    {
        get
        {
            var cityData = default(DateMeasurements);

            if (!storage.TryGetValue(city, out cityData))
                throw new ArgumentOutOfRangeException(nameof(city), "City
not found");

            // strip out any time portion:
            var index = date.Date;
            var measure = default(Measurements);
            if (cityData.TryGetValue(index, out measure))
                return measure;
            throw new ArgumentOutOfRangeException(nameof(date), "Date not
found");
        }
        set
        {
            var cityData = default(DateMeasurements);

            if (!storage.TryGetValue(city, out cityData))
            {
                cityData = new DateMeasurements();
                storage.Add(city, cityData);
            }

            // Strip out any time portion:
            var index = date.Date;
            cityData[index] = value;
        }
    }
}
```

Cet exemple crée un indexeur qui mappe des données météorologiques sur deux arguments différents : une ville (représentée par un `string`) et une date (représentée par un `DateTime`). Le stockage interne utilise deux classes `Dictionary` pour représenter le dictionnaire à deux dimensions. L'API publique ne représente plus le stockage sous-jacent. Les fonctionnalités de langage des indexeurs vous permettent de créer une interface publique qui représente votre abstraction, même si le stockage sous-jacent doit utiliser des types de collections de base différents.

Ce code comprend deux sections avec lesquelles vous pouvez ne pas être familier. Ces deux `using` directives :

```
C#
```

```
using DateMeasurements =
System.Collections.Generic.Dictionary<System.DateTime,
IndexersSamples.Common.Measurements>;
using CityDataMeasurements = System.Collections.Generic.Dictionary<string,
System.Collections.Generic.Dictionary<System.DateTime,
IndexersSamples.Common.Measurements>>;
```

créent un *alias* pour un type générique construit. Ces instructions permettent au code d'utiliser plus tard les noms plus descriptifs que sont `DateMeasurements` et `CityDataMeasurements`, au lieu de la construction générique de `Dictionary<DateTime, Measurements>` et `Dictionary<string, Dictionary<DateTime, Measurements> >`. Cette construction nécessite l'utilisation de noms de types qualifiés complets à droite du signe `=`.

La deuxième technique consiste à supprimer les sections de date et heure de tous les objets `DateTime` utilisés pour indexer des collections. .NET n'inclut pas de type date uniquement. Les développeurs utilisent le type `DateTime`, mais utilisent la propriété `Date` pour s'assurer qu'il n'existe pas d'objets `DateTime` égaux pour ce jour-là.

Récapitulatif

Vous devez créer des indexeurs chaque fois que vous avez un élément de type propriété dans votre classe, où cette propriété ne représente pas une valeur unique, mais une collection de valeurs dans laquelle chaque élément est identifié par un ensemble d'arguments. Ces arguments peuvent identifier quel élément de la collection doit être référencé. Les indexeurs étendent le concept de propriété, où un membre est considéré comme un élément de données extérieur à la classe, mais comme une méthode à l'intérieur. Les indexeurs autorisent les arguments à rechercher un élément d'une propriété qui représente un ensemble d'éléments.

Iterators

Article • 10/05/2023

Presque chaque programme que vous écrivez doit itérer au sein d'une collection. Vous allez écrire du code qui examine chaque élément d'une collection.

Vous allez également créer des méthodes d'itérateur, c'est-à-dire des méthodes qui produisent un *itérateur* pour les éléments de cette classe. Un *itérateur* est un objet qui parcourt un conteneur, notamment des listes. Les itérateurs peuvent être utilisés pour :

- Effectuer une action sur chaque élément d'une collection.
- Énumérer une collection personnalisée.
- Étendre LINQ ou d'autres bibliothèques.
- Créer un pipeline de données où les données circulent efficacement via des méthodes d'itérateur.

Le langage C# fournit des fonctionnalités permettant de générer et de consommer des séquences. Ces séquences peuvent être produites et consommées de manière synchrone ou asynchrone. Cet article présente une vue d'ensemble de ces fonctionnalités.

Itération avec foreach

L'énumération d'une collection est simple : le mot clé `foreach` énumère une collection en exécutant l'instruction incorporée une fois pour chaque élément de la collection :

C#

```
foreach (var item in collection)
{
    Console.WriteLine(item?.ToString());
}
```

C'est tout. Pour itérer au sein du contenu d'une collection, l'instruction `foreach` suffit. L'instruction `foreach` n'est cependant pas magique. Cette méthode s'appuie sur deux interfaces génériques définies dans la bibliothèque .NET Core afin de générer le code nécessaire à l'itération d'une collection : `IEnumerable<T>` et `IEnumerator<T>`. Ce mécanisme est expliqué plus en détail ci-dessous.

Ces deux interfaces ont également des contreparties non génériques : `IEnumerable` et `IEnumerator`. Les versions génériques conviennent mieux pour un code moderne.

Lorsqu'une séquence est générée de manière asynchrone, vous pouvez utiliser l'instruction `await foreach` pour consommer la séquence de manière asynchrone :

C#

```
await foreach (var item in asyncSequence)
{
Console.WriteLine(item?.ToString());
}
```

Lorsqu'une séquence correspond à System.Collections.Generic.IEnumerable<T>, vous utilisez `foreach`. Lorsqu'une séquence correspond à System.Collections.Generic.IAsyncEnumerable<T>, vous utilisez `await foreach`. Dans ce dernier cas, la séquence est générée de manière asynchrone.

Sources d'énumération avec des méthodes d'itérateur

Une autre fonctionnalité intéressante du langage C# vous permet de construire des méthodes qui créent une source pour une énumération. Ces méthodes sont appelées des *méthodes d'itérateur*. Une méthode d'itérateur définit comment générer les objets dans une séquence sur demande. Vous utilisez les mots clés contextuels `yield return` pour définir une méthode d'itérateur.

Vous pouvez par exemple écrire cette méthode pour produire la séquence d'entiers de 0 à 9 :

C#

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    yield return 0;
    yield return 1;
    yield return 2;
    yield return 3;
    yield return 4;
    yield return 5;
    yield return 6;
    yield return 7;
    yield return 8;
    yield return 9;
}
```

Le code ci-dessus montre des instructions `yield return` distinctes pour mettre en évidence le fait que vous pouvez utiliser plusieurs instructions `yield return` discrètes dans une méthode d'itérateur. Vous pouvez (et c'est souvent le cas) utiliser d'autres constructions du langage pour simplifier le code d'une méthode d'itérateur. La définition de méthode ci-dessous produit exactement la même séquence de nombres :

C#

```
public IEnumerable<int> GetSingleDigitNumbersLoop()
{
    int index = 0;
    while (index < 10)
        yield return index++;
}
```

Vous n'êtes pas obligé de choisir l'une ou l'autre. Vous pouvez avoir autant d'instructions `yield return` que nécessaire pour répondre aux besoins de votre méthode :

C#

```
public IEnumerable<int> GetSetsOfNumbers()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    index = 100;
    while (index < 110)
        yield return index++;
}
```

Tous les exemples précédents auraient un équivalent asynchrone. Dans chaque cas, vous devez remplacer le type de retour de `IEnumerable<T>` par `IAsyncEnumerable<T>`. Par exemple, la version asynchrone de l'exemple précédent serait la suivante :

C#

```
public async IAsyncEnumerable<int> GetSetsOfNumbersAsync()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    await Task.Delay(500);
```

```
    yield return 50;

    await Task.Delay(500);

    index = 100;
    while (index < 110)
        yield return index++;
}
```

Il s'agit de la syntaxe des itérateurs synchrones et asynchrones. Prenons un exemple concret. Imaginez que vous êtes sur un projet IoT et que les capteurs d'un appareil génèrent un flux de données très important. Pour obtenir un aperçu des données, vous pouvez écrire une méthode qui échantillonne les éléments de données à un certain intervalle. Cette petite méthode d'itérateur peut effectuer ce travail :

C#

```
public static IEnumerable<T> Sample<T>(this IEnumerable<T> sourceSequence,
int interval)
{
    int index = 0;
    foreach (T item in sourceSequence)
    {
        if (index++ % interval == 0)
            yield return item;
    }
}
```

Si la lecture à partir de l'appareil IoT produit une séquence asynchrone, vous devez modifier la méthode comme indiqué dans la méthode suivante :

C#

```
public static async IAsyncEnumerable<T> Sample<T>(this IAsyncEnumerable<T>
sourceSequence, int interval)
{
    int index = 0;
    await foreach (T item in sourceSequence)
    {
        if (index++ % interval == 0)
            yield return item;
    }
}
```

Il existe une restriction importante concernant les méthodes d'itérateur : vous ne pouvez pas avoir à la fois une instruction `return` et une instruction `yield return` dans la même méthode. Le code suivant ne sera pas compilé :

C#

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    // generates a compile time error:
    var items = new int[] {100, 101, 102, 103, 104, 105, 106, 107, 108, 109
};
    return items;
}
```

Cette restriction n'est normalement pas un problème. Vous avez le choix entre utiliser `yield return` dans toute la méthode ou de diviser la méthode d'origine en plusieurs méthodes, certaines utilisant `return` et d'autres utilisant `yield return`.

Vous pouvez modifier légèrement la dernière méthode pour utiliser `yield return` partout :

C#

```
public IEnumerable<int> GetFirstDecile()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    var items = new int[] {100, 101, 102, 103, 104, 105, 106, 107, 108, 109
};
    foreach (var item in items)
        yield return item;
}
```

Parfois, la bonne réponse est de fractionner une méthode d'itérateur en deux méthodes différentes. Une qui utilise `return` et l'autre qui utilise `yield return`. Imaginons que vous vouliez renvoyer une collection vide, ou les cinq premiers nombres impairs, en vous basant sur un argument booléen. Vous pouvez écrire ceci sous la forme de ces deux méthodes :

C#

```
public IEnumerable<int> GetSingleDigitOddNumbers(bool getCollection)
{
    if (getCollection == false)
        return new int[0];
    else
        return IteratorMethod();
}

private IEnumerable<int> IteratorMethod()
{
    int index = 0;
    while (index < 10)
    {
        if (index % 2 == 1)
            yield return index;
        index++;
    }
}
```

Examinez les méthodes ci-dessus. La première utilise l'instruction `return` standard pour retourner une collection vide ou l'itérateur créé par la deuxième méthode. La deuxième méthode utilise l'instruction `yield return` pour créer la séquence demandée.

Aller plus loin avec `foreach`

L'instruction `foreach` se développe en un idiome standard qui utilise les interfaces `IEnumerable<T>` et `IEnumerator<T>` pour itérer à travers tous les éléments d'une collection. Cette méthode réduit également les erreurs que font les développeurs en ne gérant pas correctement les ressources.

Le compilateur traduit la boucle `foreach` présentée dans le premier exemple en quelque chose de similaire à cette construction :

C#

```
IEnumerator<int> enumerator = collection.GetEnumerator();
while (enumerator.MoveNext())
{
    var item = enumerator.Current;
    Console.WriteLine(item.ToString());
}
```

Le code exact généré par le compilateur est plus compliqué et gère les cas où l'objet renvoyé par `GetEnumerator()` implémente l'interface `IDisposable`. L'expansion complète génère un code qui ressemble davantage à celui-ci :

C#

```
{
    var enumerator = collection.GetEnumerator();
    try
    {
        while (enumerator.MoveNext())
        {
            var item = enumerator.Current;
            Console.WriteLine(item.ToString());
        }
    }
    finally
    {
        // dispose of enumerator.
    }
}
```

Le compilateur traduit le premier exemple asynchrone en quelque chose de similaire à cette construction :

C#

```
{
    var enumerator = collection.GetAsyncEnumerator();
    try
    {
        while (await enumerator.MoveNextAsync())
        {
            var item = enumerator.Current;
            Console.WriteLine(item.ToString());
        }
    }
    finally
    {
        // dispose of async enumerator.
    }
}
```

La manière dont l'énumérateur est éliminé dépend des caractéristiques du type d'`enumerator`. Dans le cas synchrone général, la clause `finally` se développe en :

C#

```
finally
{
   (enumerator as IDisposable)?.Dispose();
}
```

Le cas asynchrone général se développe en :

```
C#

finally
{
    if (enumerator is IAsyncDisposable asyncDisposable)
        await asyncDisposable.DisposeAsync();
}
```

Cependant, si le type d'`enumerator` est un type sealed et qu'il n'existe pas de conversion implicite du type d'`enumerator` en `IDisposable` ou `IAsyncDisposable`, la clause `finally` se développe en un bloc vide :

```
C#

finally
{
}
```

S'il existe une conversion implicite du type d'`enumerator` en `IDisposable` et si `enumerator` est un type valeur qui n'autorise pas les valeurs Null, la clause `finally` se développe en :

```
C#

finally
{
   ((IDisposable)enumerator).Dispose();
}
```

Heureusement, vous n'avez pas besoin de mémoriser tous ces détails. L'instruction `foreach` gère toutes ces subtilités pour vous. Le compilateur génère le code correct pour toutes ces constructions.

Présentation des délégués et des événements en C#

Article • 10/05/2023

Les délégués fournissent un mécanisme de *liaison tardive* dans .NET. La liaison tardive signifie que vous créez un algorithme où l'appelant fournit également au moins une méthode qui implémente une partie de l'algorithme.

Par exemple, prenez le tri d'une liste d'étoiles dans une application d'astronomie. Vous pouvez choisir de trier les étoiles d'après leur distance par rapport à la terre, d'après leur magnitude ou d'après leur luminosité perçue.

Dans tous ces cas-là, la méthode Sort() effectue essentiellement la même chose : elle organise les éléments dans la liste en se basant sur une comparaison. Le code qui compare deux étoiles est différent pour chacun des ordres de tri.

Ces genres de solutions sont utilisés dans les logiciels depuis un demi siècle. Le concept de délégué du langage C# fournit une prise en charge linguistique de première classe et la sécurité de type autour du concept.

Comme vous le verrez plus loin dans cette série, le code C# que vous écrivez pour des algorithmes comme celui-ci est de type sécurisé. Le compilateur garantit que les types correspondent aux arguments et aux types de retour.

Les pointeurs de fonction ont été ajoutés à C# 9 pour des scénarios similaires, où vous avez besoin d'un contrôle supplémentaire sur la convention d'appel. Le code associé à un délégué est appelé à l'aide d'une méthode virtuelle ajoutée à un type délégué. À l'aide de pointeurs de fonction, vous pouvez spécifier différentes conventions.

Objectifs de conception de langage pour les délégués

Les concepteurs de langage ont énuméré plusieurs objectifs pour la fonctionnalité qui est finalement devenue celle des délégués.

L'équipe souhaitait une construction de langage commune pouvant être utilisée pour tout algorithme à liaison tardive. Les délégués permettent aux développeurs d'apprendre un concept et d'utiliser ce même concept dans de nombreux problèmes logiciels différents.

Ensuite, l'équipe souhaitait prendre en charge les appels de méthodes uniques et multicasts (Les délégués multidiffusion sont des délégués qui chaînent plusieurs appels de méthode. Vous verrez des exemples plus loin dans cette série.)

L'équipe souhaitait que les délégués prennent en charge la même sécurité de type que celle attendue par les développeurs pour toutes les constructions C#.

Pour finir, l'équipe a reconnu qu'un modèle d'événement était un modèle spécifique où les délégués (ou tout algorithme de liaison tardive) étaient utiles. L'équipe souhaitait s'assurer que le code pour les délégués pouvait fournir la base pour le modèle d'événement .NET.

Le résultat de tout ce travail est la prise en charge des délégués et des événements en C# et .NET.

Les autres articles de cette série traitent des fonctionnalités de langage, de la prise en charge de la bibliothèque, et des idiomes courants utilisés quand vous travaillez avec des délégués et événements. Voici ce que vous allez apprendre :

- Mot clé `delegate` et code qu'il génère.
- Les fonctionnalités de la classe `System.Delegate` et la façon dont elles sont utilisées.
- Comment créer des délégués de type sécurisé.
- Comment créer des méthodes qui peuvent être appelées par le biais de délégués.
- Comment utiliser des délégués et des événements à l'aide d'expressions lambda.
- Où les délégués deviennent l'un des blocs de construction de LINQ.
- Les délégués sont la base du modèle d'événement .NET, et comment ils diffèrent.

Commençons.

Next

System.Delegate et le mot clé `delegate`

Article • 10/05/2023

Précédent

Cet article décrit les classes .NET qui prennent en charge les délégués et comment celles-ci sont mappées au mot clé `delegate`.

Définir des types délégués

Commençons par le mot clé 'delegate', car c'est l'élément principal que vous utilisez quand vous travaillez avec des délégués. Le code que le compilateur génère quand vous utilisez le mot clé `delegate` mappe aux appels de méthode qui appellent des membres des classes Delegate et MulticastDelegate.

Vous définissez un type délégué à l'aide d'une syntaxe similaire à la définition d'une signature de méthode. Vous venez d'ajouter le mot clé `delegate` à la définition.

Continuons à utiliser la méthode List.Sort() comme dans notre exemple. La première étape consiste à créer un type pour le délégué de comparaison :

```
C#

// From the .NET Core library

// Define the delegate type:
public delegate int Comparison<in T>(T left, T right);
```

Le compilateur génère une classe, dérivée de `System.Delegate`, qui correspond à la signature utilisée (dans le cas présent, une méthode qui retourne un entier et qui a deux arguments). Le type de ce délégué est `Comparison`. Le type délégué `Comparison` est un type générique. Pour plus d'informations sur les génériques, cliquez ici.

Notez que la syntaxe peut sembler déclarer une variable, alors qu'elle déclare en fait un *type*. Vous pouvez définir des types délégués dans des classes, directement dans des espaces de noms, ou même dans l'espace de noms global.

> ⓘ **Notes**
>
> La déclaration de types délégués (ou d'autres types) directement dans l'espace de noms global n'est pas recommandée.

Le compilateur génère également des gestionnaires d'ajout et de suppression pour ce nouveau type, afin que les clients de cette classe puissent ajouter et supprimer des méthodes dans la liste d'invocation d'une instance. Le compilateur exige que la signature de la méthode ajoutée ou supprimée corresponde à la signature utilisée lors de la déclaration de la méthode.

Déclarer des instances de délégués

Après avoir défini le délégué, vous pouvez créer une instance de ce type. Comme pour toutes les variables en C#, vous ne pouvez pas déclarer d'instances de délégué directement dans un espace de noms, ni dans l'espace de noms global.

C#

```
// inside a class definition:

// Declare an instance of that type:
public Comparison<T> comparator;
```

Le type de la variable est le type délégué défini précédemment, `Comparison<T>`. Le nom de la variable est `comparator`.

Cet extrait de code ci-dessus a déclaré une variable membre à l'intérieur d'une classe. Vous pouvez également déclarer des variables de délégués qui sont des variables locales, ou bien des arguments de méthodes.

Appeler des délégués

Vous appelez les méthodes qui se trouvent dans la liste d'invocation d'un délégué en appelant ce dernier. À l'intérieur de la méthode `Sort()`, le code appelle la méthode de comparaison pour déterminer l'ordre dans lequel placer les objets :

C#

```
int result = comparator(left, right);
```

Dans la ligne ci-dessus, le code *appelle* la méthode attachée au délégué. Vous traitez la variable comme un nom de méthode et vous l'appelez à l'aide de la syntaxe d'appel de méthode normale.

Cette ligne de code effectue une hypothèse hasardeuse : il n'existe aucune garantie qu'une cible a été ajoutée au délégué. Si aucune cible n'a été attachée, la ligne ci-dessus

entraîne la levée de `NullReferenceException`. Les idiomes utilisés pour résoudre ce problème sont plus compliqués qu'un simple contrôle de valeur Null. Ils sont traités plus loin dans cette série.

Attribuer, ajouter et supprimer des cibles d'appel

Voyons comment un type délégué est défini et comment les instances de délégué sont déclarées et appelées.

Les développeurs qui souhaitent utiliser la méthode `List.Sort()` doivent définir une méthode dont la signature correspond à la définition du type délégué, puis l'assigner au délégué utilisé par la méthode sort. Cette assignation ajoute la méthode à la liste d'invocation de cet objet délégué.

Supposons que vous souhaitiez trier une liste de chaînes en fonction de leur longueur. Votre fonction de comparaison peut être la suivante :

C#

```
private static int CompareLength(string left, string right) =>
    left.Length.CompareTo(right.Length);
```

La méthode est déclarée en tant que méthode privée. C'est parfait. Vous ne voulez peut-être pas que cette méthode fasse partie de votre interface publique. Elle peut toujours être utilisée comme méthode de comparaison quand elle est attachée à un délégué. Cette méthode est attachée à la liste cible de l'objet délégué dans le code appelant, qui peut y accéder par l'intermédiaire de ce délégué.

Vous créez cette relation en passant cette méthode à la méthode `List.Sort()` :

C#

```
phrases.Sort(CompareLength);
```

Notez que le nom de la méthode est utilisé sans parenthèses. L'utilisation de la méthode comme argument indique au compilateur de convertir la référence de méthode en une référence pouvant être utilisée comme cible d'invocation de délégué, puis d'attacher cette méthode comme cible d'invocation.

Vous pouvez également avoir déclaré explicitement une variable de type `Comparison<string>` et effectué une assignation :

```
C#

Comparison<string> comparer = CompareLength;
phrases.Sort(comparer);
```

Quand la méthode utilisée comme cible du délégué est petite, il est courant d'utiliser la syntaxe des expressions lambda pour effectuer l'assignation :

```
C#

Comparison<string> comparer = (left, right) =>
left.Length.CompareTo(right.Length);
phrases.Sort(comparer);
```

L'utilisation d'expressions lambda pour les cibles de délégués est traitée plus en détail dans une section ultérieure.

L'exemple Sort() attache généralement une méthode cible unique au délégué. Toutefois, les objets délégués prennent en charge les listes d'invocation comprenant plusieurs méthodes cibles attachées à un objet délégué.

Classes Delegate et MulticastDelegate

La prise en charge du langage décrite ci-dessus fournit les fonctionnalités et la prise en charge généralement nécessaires pour utiliser des délégués. Ces fonctionnalités sont basées sur deux classes du framework .NET Core : Delegate et MulticastDelegate.

La classe `System.Delegate` et son unique sous-classe directe `System.MulticastDelegate` fournissent la prise en charge du framework pour la création de délégués, l'inscription de méthodes comme cibles de délégués et l'appel de toutes les méthodes inscrites comme cible de délégué.

Il est intéressant de noter que les classes `System.Delegate` et `System.MulticastDelegate` ne sont pas elles-mêmes des types délégués. Elles servent de base à tous les types délégués spécifiques. Ce même processus de conception du langage a stipulé que vous ne pouvez pas déclarer une classe qui dérive de `Delegate` ou de `MulticastDelegate`. Les règles du langage C# l'interdisent.

À la place, le compilateur C# crée des instances d'une classe dérivée de `MulticastDelegate` quand vous utilisez le mot clé du langage C# pour déclarer des types délégués.

Cette conception trouve ses racines dans la première version Release de C# et du .NET. L'équipe de conception avait pour objectif de s'assurer que le langage appliquait la cohérence des types lors de l'utilisation de délégués. Il était important de garantir que les délégués étaient appelés avec le type et le nombre d'arguments appropriés. Et que tout type de retour était correctement indiqué au moment de la compilation. Les délégués faisaient partie de la version Release du .NET 1.0, qui était antérieure aux génériques.

La création, par le compilateur, des classes déléguées concrètes qui représentaient la signature de méthode utilisée constituait la meilleure façon d'appliquer la cohérence des types.

Même si vous ne pouvez pas créer des classes dérivées directement, vous utiliserez les méthodes définies sur ces classes. Examinons les méthodes les plus courantes que vous utiliserez quand vous travaillerez avec des délégués.

Le premier et le plus important point à retenir est que chaque délégué avec lequel vous travaillez est dérivé de `MulticastDelegate`. Un délégué multicast signifie que plusieurs méthodes cibles peuvent être appelées lors d'un appel par l'intermédiaire d'un délégué. La conception d'origine envisageait de distinguer les délégués où une seule méthode cible pouvait être attachée et appelée, et les délégués où plusieurs méthodes cibles pouvaient être attachées et appelées. En réalité, cette distinction s'est avérée moins utile que prévu. Les deux différentes classes étaient déjà créées et sont présentes dans le framework depuis la première version Release publique de celui-ci.

Les méthodes que vous utiliserez le plus avec les délégués sont `Invoke()` et `BeginInvoke()` / `EndInvoke()`. `Invoke()` appelle toutes les méthodes qui ont été attachées à une instance de délégué particulière. Comme vous l'avez vu ci-dessus, vous appelez généralement des délégués à l'aide de la syntaxe d'appel de méthode sur la variable de délégué. Comme vous le verrez plus loin dans cette série, il existe des modèles qui fonctionnent directement avec ces méthodes.

Maintenant que vous avez vu la syntaxe du langage et les classes qui prennent en charge les délégués, examinons la façon dont les délégués fortement typés sont utilisés, créés et appelés.

Next

Délégués fortement typés

Article • 10/05/2023

Précédent

Dans l'article précédent, vous avez vu que vous pouviez créer des types délégués spécifiques à l'aide du mot clé `delegate`.

La classe abstraite Delegate fournit l'infrastructure pour l'invocation et le couplage faible. Les types délégués concrets deviennent beaucoup plus utiles en adoptant et en appliquant la sécurité de type pour les méthodes qui sont ajoutées à la liste d'invocation d'un objet délégué. Quand vous utilisez le mot clé `delegate` et que vous définissez un type délégué concret, le compilateur génère ces méthodes.

Dans la pratique, cela conduirait à la création de types délégués chaque fois que vous avez besoin d'une signature de méthode différente. Ce travail peut devenir fastidieux au bout d'un moment. Chaque nouvelle fonctionnalité nécessite de nouveaux types délégués.

Heureusement, cela n'est pas nécessaire. Le framework .NET Core contient plusieurs types que vous pouvez réutiliser chaque fois que vous avez besoin de types délégués. Il s'agit de définitions génériques. Vous pouvez ainsi déclarer des personnalisations quand vous avez besoin de nouvelles déclarations de méthode.

Le premier de ces types est le type Action et plusieurs variantes :

C#

```
public delegate void Action();
public delegate void Action<in T>(T arg);
public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);
// Other variations removed for brevity.
```

Le modificateur `in` sur l'argument de type générique est traité dans l'article sur la covariance.

Il existe des variantes du délégué `Action` qui contiennent jusqu'à 16 arguments, comme Action<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16>. Il est important que ces définitions utilisent différents arguments génériques pour chacun des arguments de délégués : cela offre une flexibilité maximale. Les arguments de méthode ne doivent pas obligatoirement être du même type, mais il peuvent l'être.

Utilisez l'un des types `Action` pour tout type délégué ayant un type de retour void.

Le framework inclut également plusieurs types délégués génériques que vous pouvez utiliser pour les types délégués qui retournent des valeurs :

```
C#
```

```
public delegate TResult Func<out TResult>();
public delegate TResult Func<in T1, out TResult>(T1 arg);
public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
// Other variations removed for brevity
```

Le modificateur `out` sur l'argument de type générique de résultat est traité dans l'article sur la covariance.

Il existe des variantes du délégué `Func` qui contiennent jusqu'à 16 arguments d'entrée, comme Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16,TResult>. Par convention, le type du résultat est toujours le dernier paramètre de type dans toutes les déclarations `Func`.

Utilisez l'un des types `Func` pour tout type délégué qui retourne une valeur.

Il existe aussi un type Predicate<T> spécialisé pour un délégué qui retourne un test sur une seule valeur :

```
C#
```

```
public delegate bool Predicate<in T>(T obj);
```

Vous remarquerez que pour tout type `Predicate`, il existe un type `Func` structurellement équivalent, par exemple :

```
C#
```

```
Func<string, bool> TestForString;
Predicate<string> AnotherTestForString;
```

Ces deux types peuvent vous sembler équivalents, mais ils ne le sont pas. Ces deux variables ne sont pas interchangeables. Une variable d'un type ne peut pas être assignée à l'autre type. Le système de type C# utilise les noms des types définis, pas la structure.

Toutes ces définitions de types délégués dans la bibliothèque .NET Core doivent signifier que vous n'avez pas besoin de définir un nouveau type délégué pour toute nouvelle fonctionnalité que vous créez et qui nécessite des délégués. Ces définitions génériques doivent fournir tous les types délégués dont vous avez besoin dans la plupart des situations. Il vous suffit d'instancier l'un de ces types avec les paramètres de type

nécessaires. Dans le cas des algorithmes qui peuvent être rendus génériques, ces délégués peuvent être utilisés en tant que types génériques.

Cela devrait vous procurer un gain de temps et réduire le nombre de nouveaux types que vous devez créer pour travailler avec des délégués.

Dans l'article suivant, vous verrez plusieurs schémas d'utilisation courants des délégués.

Next

Modèles courants pour les délégués

Article • 07/04/2023

Précédent

Les délégués fournissent un mécanisme qui autorise des conceptions logicielles impliquant un couplage minimal entre les composants.

LINQ constitue un excellent exemple de ce genre de conception. Le modèle d'expression de requête LINQ s'appuie sur des délégués pour toutes ses fonctionnalités. Prenons l'exemple simple suivant :

```
C#

var smallNumbers = numbers.Where(n => n < 10);
```

Il filtre la séquence de nombres pour ne sélectionner que ceux qui sont inférieurs à 10. La méthode `Where` utilise un délégué qui détermine les éléments d'une séquence qui correspondent au filtre. Quand vous créez une requête LINQ, vous fournissez l'implémentation du délégué à cette fin spécifique.

Le prototype de la méthode Where est le suivant :

```
C#

public static IEnumerable<TSource> Where<TSource> (this IEnumerable<TSource>
source, Func<TSource, bool> predicate);
```

Cet exemple est répété avec toutes les méthodes qui font partie de LINQ. Elles s'appuient toutes sur des délégués pour le code qui gère la requête spécifique. Ce modèle de conception d'API est puissant, une fois appris et compris.

Cet exemple simple montre que les délégués nécessitent très peu de couplage entre les composants. Vous n'avez pas besoin de créer de classe qui dérive d'une classe de base particulière. Vous n'avez pas besoin d'implémenter une interface spécifique. La seule exigence consiste à fournir l'implémentation d'une méthode qui est essentielle à la tâche en question.

Créer vos propres composants avec des délégués

Étendons cet exemple en créant un composant à l'aide d'une conception qui s'appuie sur des délégués.

Nous allons définir un composant qui peut être utilisé pour des messages de journaux dans un système de grande taille. Les composants de bibliothèque pourraient être utilisés dans de nombreux environnements différents, sur plusieurs plateformes différentes. Il existe de nombreuses fonctionnalités courantes dans le composant qui gère les journaux. Il doit accepter des messages en provenance de tout composant du système. Ces messages auront des priorités différentes, que le composant principal peut gérer. Les messages doivent avoir des horodatages dans leur forme finale archivée. Pour des scénarios avancés, vous pouvez filtrer les messages en fonction du composant source.

Il y a un aspect de la fonctionnalité qui changera souvent : l'emplacement d'écriture des messages. Dans certains environnements, ils pourront être écrits dans la console des erreurs. Dans d'autres, il pourra s'agir d'un fichier. Les autres possibilités incluent le stockage de base de données, les journaux des événements du système d'exploitation ou autre stockage de documents.

Il existe également des combinaisons de sortie qui peuvent être utilisées dans différents scénarios. Vous souhaiterez peut-être écrire des messages dans la console et dans un fichier.

Une conception basée sur des délégués offre une très grande souplesse et facilite la prise en charge des mécanismes de stockage qui pourront être ajoutés à l'avenir.

Dans cette conception, le composant de journal principal peut être une classe non virtuelle, voire sealed. Vous pouvez incorporer n'importe quel jeu de délégués pour écrire les messages sur différents supports de stockage. La prise en charge intégrée des délégués multicast facilite la prise en charge des scénarios où les messages doivent être écrits à plusieurs emplacements (dans un fichier et une console).

Une première implémentation

Commençons par quelque chose de simple : l'implémentation initiale acceptera les nouveaux messages et les écrira à l'aide de n'importe quel délégué attaché. Vous pouvez démarrer avec un délégué qui écrit des messages dans la console.

C#

```
public static class Logger
{
    public static Action<string>? WriteMessage;
```

```
    public static void LogMessage(string msg)
    {
        if (WriteMessage is not null)
            WriteMessage(msg);
    }
}
```

La classe statique ci-dessus est la chose la plus simple qui peut fonctionner. Nous devons écrire l'implémentation unique pour la méthode qui écrit des messages dans la console :

C#

```
public static class LoggingMethods
{
    public static void LogToConsole(string message)
    {
        Console.Error.WriteLine(message);
    }
}
```

Pour finir, nous devons raccorder le délégué en l'attachant au délégué WriteMessage déclaré dans l'enregistreur d'événements :

C#

```
Logger.WriteMessage += LoggingMethods.LogToConsole;
```

Pratiques

Jusqu'à présent, notre exemple est assez simple, mais il illustre quand même certaines instructions importantes pour les conceptions impliquant des délégués.

L'utilisation des types délégués définis dans l'infrastructure principale permet aux utilisateurs de travailler facilement avec les délégués. Vous n'avez pas besoin de définir de nouveaux types, et les développeurs qui utilisent votre bibliothèque n'ont pas besoin d'apprendre de nouveaux types délégués spécialisés.

Les interfaces utilisées sont le plus minimal et flexible possible : pour créer un enregistreur d'événements de sortie, vous devez créer une méthode. Il peut s'agir d'une méthode statique ou d'une méthode d'instance. Elle peut avoir n'importe quel accès.

Mettre en forme la sortie

Nous allons maintenant rendre cette première version un peu plus robuste et commencer à créer d'autres mécanismes de journalisation.

Ensuite, nous ajouterons quelques arguments à la méthode `LogMessage()` pour que notre classe de journal crée des messages plus structurés :

C#

```
public enum Severity
{
    Verbose,
    Trace,
    Information,
    Warning,
    Error,
    Critical
}
```

C#

```
public static class Logger
{
    public static Action<string>? WriteMessage;

    public static void LogMessage(Severity s, string component, string msg)
    {
        var outputMsg = $"{DateTime.Now}\t{s}\t{component}\t{msg}";
        if (WriteMessage is not null)
            WriteMessage(outputMsg);
    }
}
```

Maintenant, utilisons cet argument `Severity` pour filtrer les messages envoyés à la sortie du journal.

C#

```
public static class Logger
{
    public static Action<string>? WriteMessage;

    public static Severity LogLevel { get; set; } = Severity.Warning;

    public static void LogMessage(Severity s, string component, string msg)
    {
        if (s < LogLevel)
            return;

        var outputMsg = $"{DateTime.Now}\t{s}\t{component}\t{msg}";
        if (WriteMessage is not null)
```

```
            WriteMessage(outputMsg);
        }
    }
```

Pratiques

Nous avons ajouté de nouvelles fonctionnalités à l'infrastructure de journalisation. Le composant enregistreur d'événements étant très faiblement couplé à tout mécanisme de sortie, ces nouvelles fonctionnalités peuvent être ajoutées sans aucun impact sur le code qui implémente le délégué enregistreur d'événements.

Au fil du temps, vous verrez d'autres d'exemples montrant comment ce couplage faible autorise une plus grande souplesse dans la mise à jour des parties du site sans aucune modification des autres emplacements. En fait, dans une application plus grande, les classes de sortie de l'enregistreur d'événements peuvent être dans un autre assembly et ne même pas avoir besoin d'être reconstruites.

Générer un deuxième moteur de sortie

Le composant journal commence à prendre forme. Ajoutons un autre moteur de sortie qui enregistre les messages dans un fichier. Ce moteur de sortie est légèrement plus complexe. Il s'agit d'une classe qui encapsule les opérations de fichiers et garantit que le fichier est toujours fermé après chaque écriture. Cela permet de s'assurer que toutes les données sont vidées sur le disque après chaque génération de message.

Voici cet enregistreur d'événements basé sur fichier :

C#

```
public class FileLogger
{
    private readonly string logPath;
    public FileLogger(string path)
    {
        logPath = path;
        Logger.WriteMessage += LogMessage;
    }

    public void DetachLog() => Logger.WriteMessage -= LogMessage;
    // make sure this can't throw.
    private void LogMessage(string msg)
    {
        try
        {
            using (var log = File.AppendText(logPath))
            {
```

```
                log.WriteLine(msg);
                log.Flush();
            }
        }
        catch (Exception)
        {
            // Hmm. We caught an exception while
            // logging. We can't really log the
            // problem (since it's the log that's failing).
            // So, while normally, catching an exception
            // and doing nothing isn't wise, it's really the
            // only reasonable option here.
        }
    }
}
```

Une fois que nous avons créé cette classe, nous pouvons l'instancier, et elle attache sa méthode LogMessage au composant enregistreur d'événements :

C#

```
var file = new FileLogger("log.txt");
```

Ces deux méthodes ne s'excluent pas mutuellement. Nous pourrions attacher les deux méthodes de journalisation et générer des messages dans la console et dans un fichier :

C#

```
var fileOutput = new FileLogger("log.txt");
Logger.WriteMessage += LoggingMethods.LogToConsole; // LoggingMethods is the
static class we utilized earlier
```

Plus tard, même dans la même application, nous pourrions supprimer l'un des délégués sans provoquer d'autre problème dans le système :

C#

```
Logger.WriteMessage -= LoggingMethods.LogToConsole;
```

Pratiques

Nous avons maintenant ajouté un deuxième gestionnaire de sortie pour le sous-système de journalisation. Celui-ci a besoin d'un peu plus d'infrastructure pour correctement prendre en charge le système de fichiers. Le délégué est une méthode d'instance. Il

s'agit aussi d'une méthode privée. Une plus grande accessibilité n'est pas nécessaire, car l'infrastructure de délégué peut connecter les délégués.

Ensuite, la conception basée sur les délégués autorise plusieurs méthodes de sortie sans code supplémentaire. Vous n'avez pas besoin de créer d'infrastructure supplémentaire pour prendre en charge plusieurs méthodes de sortie. Elles viennent simplement s'ajouter comme autre méthode dans la liste d'invocation.

Faites particulièrement attention au code de la méthode de sortie de journalisation de fichier. Cette méthode est codée pour s'assurer qu'elle ne lève pas d'exception. Même si ce n'est pas toujours strictement nécessaire, cette pratique est souvent recommandée. Si l'une des méthodes délégué lève une exception, les délégués restants qui sont sur la liste d'invocation ne sont pas appelés.

Dernière remarque : l'enregistreur d'événements de fichier doit gérer ses ressources en ouvrant et en fermant le fichier à chaque message de journal. Vous pouvez choisir de laisser le fichier ouvert et d'implémenter `IDisposable` pour fermer le fichier une fois que vous avez terminé. Les deux méthodes ont leurs avantages et leurs inconvénients. Elles créent toutes les deux un peu plus de couplage entre les classes.

Pour prendre en charge ces deux scénarios, il est inutile de mettre à jour le code de la classe `Logger`.

Gérer les délégués null

Pour finir, nous allons mettre à jour la méthode LogMessage afin qu'elle soit robuste pour les cas où aucun mécanisme de sortie n'est sélectionné. L'implémentation actuelle lève une `NullReferenceException` quand aucune liste d'invocation n'est attachée au délégué `WriteMessage`. Vous préférerez peut-être une conception qui continue en mode silencieux quand aucune méthode n'a été attachée. Pour cela, vous pouvez utiliser l'opérateur conditionnel null, combiné avec la méthode `Delegate.Invoke()` :

C#

```
public static void LogMessage(string msg)
{
    WriteMessage?.Invoke(msg);
}
```

L'opérateur conditionnel null (`?.`) court-circuite quand l'opérande gauche (ici, `WriteMessage`) est null, ce qui signifie que le système ne tente pas de consigner un message dans le journal.

Vous ne trouverez pas la méthode `Invoke()` répertoriée dans la documentation de `System.Delegate` ou `System.MulticastDelegate`. Le compilateur génère une méthode `Invoke` de type sécurisé pour tout type de délégué déclaré. Dans cet exemple, cela signifie que `Invoke` prend un seul argument `string` et a un type de retour void.

Récapitulatif des méthodes

Nous venons de voir le début d'un composant de journal qui peut être étendu avec d'autres enregistreurs, ainsi que d'autres fonctionnalités. Grâce à l'utilisation de délégués dans la conception, ces différents composants sont faiblement couplés. Cela présente plusieurs avantages. Il est aisé de créer de nouveaux mécanismes de sortie et de les attacher au système. Ces autres mécanismes n'ont besoin que d'une seule méthode : celle qui écrit le message du journal. Cette conception est résiliente lors de l'ajout de nouvelles fonctionnalités. Le contrat requis pour tout enregistreur consiste à implémenter une méthode. Il peut s'agir d'une méthode d'instance ou statique. Elle peut être publique, privée ou avoir tout autre accès légal.

La classe Logger peut apporter des améliorations ou des modifications sans introduire de modifications avec rupture. Comme toute classe, vous ne pouvez pas modifier l'API publique sans risque de modifications avec rupture. Toutefois, le couplage entre l'enregistreur d'événements et les moteurs de sortie étant uniquement par l'intermédiaire du délégué, aucun autre type (tel que les interfaces ou classes de base) n'est appelé. Le couplage est aussi réduit que possible.

Next

Introduction aux événements

Article • 10/05/2023

Précédent

Les événements sont, comme les délégués, un mécanisme de *liaison tardive*. En fait, les événements reposent sur la prise en charge linguistique des délégués.

Ils permettent à un objet de signaler (à tous les composants du système intéressés), que quelque chose s'est produit. Tous les autres composants peuvent s'abonner à l'événement et être averti quand un événement est déclenché.

Vous avez sans doute déjà utilisé des événements durant vos tâches de programmation. De nombreux systèmes graphiques ont un modèle d'événement pour signaler l'interaction utilisateur. Ces événements signalent les mouvements de la souris, les pressions sur des boutons et autres interactions similaires. C'est l'un des scénarios les plus courants, mais certainement pas le seul, où des événements sont utilisés.

Vous pouvez définir les événements qui doivent être déclenchés pour vos classes. Quand vous travaillez avec des événements, vous devez prendre en compte le fait qu'il se peut qu'aucun objet ne soit inscrit pour un événement particulier. Vous devez écrire votre code pour qu'il ne déclenche pas d'événement quand aucun détecteur n'est configuré.

L'abonnement à un événement crée également un couplage entre deux objets (la source de l'événement et le récepteur d'événements). Vous devez vérifier que le récepteur d'événements annule l'abonnement à la source de l'événement quand il n'est plus intéressé par les événements.

Objectifs de conception pour la prise en charge des événements

En matière d'événements, la conception du langage vise les objectifs suivants :

- Un couplage minimal entre une source d'événements et un récepteur d'événements doit être possible. Ces deux composants peuvent ne pas avoir été écrits par la même organisation, et peuvent même être mis à jour d'après une planification totalement différente.

- La procédure d'abonnement à un événement et de désabonnement de celui-ci doit être très simple.

- Les sources d'événements doivent prendre en charge plusieurs abonnés aux événements. Elles doivent aussi prendre en charge les scénarios où aucun abonné aux événements n'est attaché.

Vous pouvez constater que les objectifs pour les événements sont très similaires aux objectifs pour les délégués. C'est pourquoi la prise en charge linguistique des événements repose sur la prise en charge linguistique des délégués.

Prise en charge du langage des événements

La syntaxe de définition des événements, et d'abonnement ou d'annulation des abonnements, est une extension de la syntaxe des délégués.

Pour définir un événement, vous devez utiliser le mot clé `event` :

```
C#

public event EventHandler<FileListArgs> Progress;
```

Le type de l'événement (`EventHandler<FileListArgs>` dans cet exemple) doit être un type délégué. Vous devez respecter plusieurs conventions lors de la déclaration d'un événement. En général, le type délégué d'événement a un retour void. Les déclarations d'événements doivent être un verbe ou une phrase verbale. Utilisez le passé lorsque l'événement signale quelque chose qui s'est produit. Utilisez un verbe au présent (par exemple, `Closing`) pour signaler quelque chose qui est sur le point de se produire. Souvent, le présent indique que votre classe prend en charge un type de comportement de personnalisation. L'un des scénarios les plus courants consiste à prendre en charge l'annulation. Par exemple, un événement `Closing` peut inclure un argument qui indique si l'opération de fermeture doit continuer ou non. D'autres scénarios peuvent permettre aux appelants de modifier le comportement en mettant à jour les propriétés des arguments de l'événement. Vous pouvez déclencher un événement pour indiquer une action suivante qu'un algorithme effectuera. Le gestionnaire d'événements peut imposer une action différente en modifiant les propriétés de l'argument de l'événement.

Quand vous souhaitez déclencher l'événement, vous appelez les gestionnaires d'événements à l'aide de la syntaxe d'appel de délégué :

```
C#

Progress?.Invoke(this, new FileListArgs(file));
```

Comme indiqué dans la section sur les délégués, l'opérateur ?. permet de s'assurer facilement que vous n'essayez pas de déclencher l'événement quand il n'y a aucun abonné à cet événement.

Vous vous abonnez à un événement à l'aide de l'opérateur `+=` :

```
C#

EventHandler<FileListArgs> onProgress = (sender, eventArgs) =>
    Console.WriteLine(eventArgs.FoundFile);

fileLister.Progress += onProgress;
```

La méthode du gestionnaire comporte en général le préfixe « On » suivi du nom de l'événement, comme indiqué ci-dessus.

Vous pouvez vous désinscrire à l'aide de l'opérateur `-=` :

```
C#

fileLister.Progress -= onProgress;
```

Vous devez impérativement déclarer une variable locale pour l'expression qui représente le gestionnaire d'événements. Cela garantit que le désabonnement supprime le gestionnaire. Si au lieu de cela vous utilisez le corps de l'expression lambda, vous essayez de supprimer un gestionnaire qui n'a jamais été attaché, ce qui ne fait rien.

Dans l'article suivant, vous en apprendrez davantage sur les modèles d'événements les plus courants et sur les différentes variantes de cet exemple.

Next

Modèles d'événement .NET standard

Article • 10/05/2023

Précédent

Les événements .NET respectent en général quelques modèles connus. La normalisation d'après ces modèles signifie que les développeurs peuvent exploiter les connaissances de ces modèles standard, qui peuvent être appliqués à n'importe quel programme d'événement .NET.

Un examen de ces modèles standard vous permettra d'acquérir toutes les connaissances nécessaires pour créer des sources d'événements standard, et pour traiter et vous abonner à des événements standard dans votre code.

Signatures de délégués d'événements

La signature standard d'un délégué d'événement .NET est la suivante :

C#

```
void EventRaised(object sender, EventArgs args);
```

Le type de retour est void. Les événements sont basés sur des délégués et il s'agit de délégués multicast. Cela permet de prendre en charge plusieurs abonnés pour toute source d'événement. La valeur de retour unique d'une méthode ne s'étend pas à plusieurs abonnés aux événements. Quelle valeur de retour la source de l'événement voit-elle après le déclenchement d'un événement ? Plus loin dans cet article, vous verrez comment créer des protocoles d'événements qui prennent en charge les abonnés aux événements qui signalent des informations à la source de l'événement.

La liste d'arguments contient deux arguments : l'expéditeur et les arguments de l'événement. Le type au moment de compilation de `sender` est `System.Object`, même si vous connaissez probablement un type plus dérivé qui serait toujours correct. Par convention, utilisez `object`.

Le deuxième argument était traditionnellement un type dérivé de `System.EventArgs`. (Vous verrez dans la section suivante que cette convention n'est plus appliquée.) Si votre type d'événement n'a pas besoin d'autres arguments, vous fournirez quand même ces deux arguments. Il existe une valeur spéciale, `EventArgs.Empty`, que vous devez utiliser pour indiquer que votre événement ne contient pas d'informations supplémentaires.

Commençons par créer une classe qui répertorie les fichiers contenus dans un répertoire ou dans l'un de ses sous-répertoires qui suivent un modèle. Ce composant déclenche un événement pour chaque fichier détecté qui correspond au modèle.

L'utilisation d'un modèle d'événement offre certains avantages en matière de conception. Vous pouvez créer plusieurs détecteurs d'événements qui effectuent des actions différentes quand un fichier recherché est trouvé. La combinaison des différents détecteurs permet de créer des algorithmes plus robustes.

Voici la déclaration d'argument d'événement initiale pour trouver un fichier recherché :

C#

```
public class FileFoundArgs : EventArgs
{
    public string FoundFile { get; }

    public FileFoundArgs(string fileName) => FoundFile = fileName;
}
```

Bien que ce type ressemble à un petit type « données uniquement », vous devez suivre la convention et faire de lui un type référence (`class`). Cela signifie que l'objet d'argument sera passé par référence, et que les mises à jour des données seront visibles par tous les abonnés. La première version est un objet immuable. Il vaut mieux rendre immuables les propriétés dans votre argument d'événement. Ainsi, un abonné ne peut pas changer les valeurs avant qu'un autre abonné ne les voit. (Il existe des exceptions, comme vous le verrez ci-dessous.)

Ensuite, nous devons créer la déclaration d'événement dans la classe FileSearcher. L'utilisation du type `EventHandler<T>` nous évite de devoir créer une autre définition de type. Il suffit d'utiliser une spécialisation générique.

Nous allons remplir la classe FileSearcher pour rechercher les fichiers qui correspondent à un modèle et déclencher l'événement approprié quand une correspondance est détectée.

C#

```
public class FileSearcher
{
    public event EventHandler<FileFoundArgs>? FileFound;

    public void Search(string directory, string searchPattern)
    {
        foreach (var file in Directory.EnumerateFiles(directory,
searchPattern))
```

```
        {
            RaiseFileFound(file);
        }
    }

    private void RaiseFileFound(string file) =>
        FileFound?.Invoke(this, new FileFoundArgs(file));
}
```

Définir et déclencher des événements de type champ

Pour ajouter un événement à votre classe, le plus simple consiste à déclarer cet événement en tant que champ public, comme dans l'exemple précédent :

C#

```
public event EventHandler<FileFoundArgs>? FileFound;
```

Ce code semble déclarer un champ public, ce qui semble être une mauvaise pratique orientée objet. Vous devez protéger l'accès aux données par l'intermédiaire des propriétés ou méthodes. Bien que cela semble être une mauvaise pratique, le code généré par le compilateur crée en fait des wrappers pour que les objets d'événements soient accessibles uniquement de manière sécurisée. Les seules opérations disponibles sur un événement de type champ sont l'ajout de gestionnaire :

C#

```
var fileLister = new FileSearcher();
int filesFound = 0;

EventHandler<FileFoundArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    filesFound++;
};

fileLister.FileFound += onFileFound;
```

et la suppression de gestionnaire :

C#

```
fileLister.FileFound -= onFileFound;
```

Notez qu'il existe une variable locale pour le gestionnaire. Si vous utilisiez le corps de l'expression lambda, la suppression ne fonctionnerait pas correctement. Il s'agirait d'une autre instance du délégué, et l'opération ne ferait rien en mode silencieux.

Le code en dehors de la classe ne peut pas déclencher l'événement et ne peut pas non plus effectuer d'autres opérations.

Valeurs renvoyées à partir des abonnés aux événements

Notre version simple fonctionne correctement. Ajoutons maintenant une autre fonctionnalité : l'annulation.

Quand vous déclenchez l'événement détecté, les détecteurs doivent pouvoir arrêter le traitement si ce fichier est le dernier recherché.

Les gestionnaires d'événements ne retournant pas de valeur, vous devez communiquer cela par un autre moyen. Le modèle d'événement standard utilise l'objet `EventArgs` pour inclure des champs que les abonnés aux événements peuvent utiliser pour signaler une annulation.

Deux modèles différents peuvent être utilisés, en fonction de la sémantique du contrat d'annulation. Dans les deux cas, vous ajouterez un champ booléen aux arguments d'événements pour l'événement de fichier trouvé.

L'un des modèles autorise n'importe quel abonné à annuler l'opération. Pour ce modèle, le nouveau champ est initialisé avec la valeur `false`. Tout abonné peut le changer et lui affecter la valeur `true`. Une fois que tous les abonnés ont vu l'événement déclenché, le composant FileSearcher examine la valeur booléenne et effectue une action.

Le deuxième modèle annule l'opération uniquement si tous les abonnés souhaitent l'annuler. Dans ce modèle, le nouveau champ est initialisé pour indiquer que l'opération doit être annulée, et n'importe quel abonné peut le changer pour indiquer que l'opération doit continuer. Une fois que tous les abonnés ont vu l'événement déclenché, le composant FileSearcher examine la valeur booléenne et effectue une action. Il existe une étape supplémentaire dans ce modèle : le composant doit savoir si des abonnés ont vu l'événement. S'il n'y a pas d'abonnés, le champ indique incorrectement une annulation.

Implémentons la première version pour cet exemple. Vous devez ajouter un champ booléen nommé `CancelRequested` au type `FileFoundArgs` :

C#

```
public class FileFoundArgs : EventArgs
{
    public string FoundFile { get; }
    public bool CancelRequested { get; set; }

    public FileFoundArgs(string fileName) => FoundFile = fileName;
}
```

Ce nouveau champ est automatiquement initialisé sur `false`, la valeur par défaut pour un champ `Boolean`, pour éviter tout risque d'annulation accidentelle. Le seul autre changement à apporter au composant consiste à vérifier l'indicateur après le déclenchement de l'événement, pour voir si l'un des abonnés a demandé une annulation :

C#

```
private void SearchDirectory(string directory, string searchPattern)
{
    foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
    {
        FileFoundArgs args = RaiseFileFound(file);
        if (args.CancelRequested)
        {
            break;
        }
    }
}

private FileFoundArgs RaiseFileFound(string file)
{
    var args = new FileFoundArgs(file);
    FileFound?.Invoke(this, args);
    return args;
}
```

L'un des avantages de ce modèle est qu'il ne s'agit pas d'une modification avec rupture. Aucun des abonnés n'a demandé d'annulation auparavant, et ils n'en demandent toujours pas. Aucun code d'abonné ne nécessite de mise à jour, sauf s'il souhaite prendre en charge le nouveau protocole d'annulation. Le couplage est très faible.

Nous allons maintenant mettre à jour l'abonné pour qu'il demande une annulation dès qu'il trouve le premier exécutable :

C#

```
EventHandler<FileFoundArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    eventArgs.CancelRequested = true;
};
```

Ajout d'une autre déclaration d'événement

Nous allons ajouter une autre fonctionnalité et illustrer d'autres idiomes de langage pour les événements. Ajoutons une surcharge de la méthode `Search` qui parcourt tous les sous-répertoires à la recherche de fichiers.

Cette opération pourrait prendre beaucoup de temps dans un répertoire contenant de nombreux sous-répertoires. Ajoutons un événement déclenché au début de chaque nouvelle recherche dans un répertoire. Cela permet aux abonnés de suivre la progression et de tenir l'utilisateur à jour. Tous les exemples que nous avons créés jusqu'à présent sont publics. Faisons de celui-ci un événement interne. Cela signifie que nous pouvons aussi rendre internes les types utilisés pour les arguments.

Nous allons commencer par créer la nouvelle classe dérivée EventArgs pour signaler le nouveau répertoire et la progression.

C#

```
internal class SearchDirectoryArgs : EventArgs
{
    internal string CurrentSearchDirectory { get; }
    internal int TotalDirs { get; }
    internal int CompletedDirs { get; }

    internal SearchDirectoryArgs(string dir, int totalDirs, int
completedDirs)
    {
        CurrentSearchDirectory = dir;
        TotalDirs = totalDirs;
        CompletedDirs = completedDirs;
    }
}
```

Là encore, nous pouvons suivre les recommandations pour créer un type référence immuable pour les arguments d'événements.

Maintenant, définissons l'événement. Cette fois-ci, nous utiliserons une syntaxe différente. En plus d'utiliser la syntaxe du champ, nous pouvons créer explicitement la propriété, avec des gestionnaires d'ajout et de suppression. Dans cet exemple, nous

n'aurons pas besoin de code supplémentaire dans ces gestionnaires, mais cet exemple montre comment les créer.

C#

```
internal event EventHandler<SearchDirectoryArgs> DirectoryChanged
{
    add { _directoryChanged += value; }
    remove { _directoryChanged -= value; }
}
private EventHandler<SearchDirectoryArgs>? _directoryChanged;
```

Le code que nous écrivons ici reflète en grande partie le code généré par le compilateur pour les définitions d'événements de champs que nous avons vu précédemment. Nous créons l'événement à l'aide d'une syntaxe très similaire à celle utilisée pour les propriétés. Notez que les gestionnaires ont des noms différents : `add` et `remove`. Il sont appelés pour s'abonner à l'événement ou pour annuler un abonnement. Notez que vous devez également déclarer un champ de stockage privé pour stocker la variable d'événement. Il est initialisé avec la valeur null.

Ensuite, nous allons ajouter la surcharge de la méthode `Search` qui parcourt les sous-répertoires et déclenche les deux événements. Le moyen le plus simple consiste à utiliser un argument par défaut pour indiquer que nous souhaitons rechercher dans tous les répertoires :

C#

```
public void Search(string directory, string searchPattern, bool
searchSubDirs = false)
{
    if (searchSubDirs)
    {
        var allDirectories = Directory.GetDirectories(directory, "*.*",
SearchOption.AllDirectories);
        var completedDirs = 0;
        var totalDirs = allDirectories.Length + 1;
        foreach (var dir in allDirectories)
        {
            RaiseSearchDirectoryChanged(dir, totalDirs, completedDirs++);
            // Search 'dir' and its subdirectories for files that match the
search pattern:
            SearchDirectory(dir, searchPattern);
        }
        // Include the Current Directory:
        RaiseSearchDirectoryChanged(directory, totalDirs, completedDirs++);

        SearchDirectory(directory, searchPattern);
    }
    else
```

```
    {
        SearchDirectory(directory, searchPattern);
    }
}

private void SearchDirectory(string directory, string searchPattern)
{
    foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
    {
        FileFoundArgs args = RaiseFileFound(file);
        if (args.CancelRequested)
        {
            break;
        }
    }
}

private void RaiseSearchDirectoryChanged(
    string directory, int totalDirs, int completedDirs) =>
    _directoryChanged?.Invoke(
        this,
            new SearchDirectoryArgs(directory, totalDirs, completedDirs));

private FileFoundArgs RaiseFileFound(string file)
{
    var args = new FileFoundArgs(file);
    FileFound?.Invoke(this, args);
    return args;
}
```

À ce stade, nous pouvons exécuter l'application qui appelle la surcharge pour rechercher dans tous les sous-répertoires. Il n'existe aucun abonné sur le nouvel événement `DirectoryChanged`, mais l'utilisation de l'idiome `?.Invoke()` garantit que cela fonctionne correctement.

Ajoutons un gestionnaire pour écrire une ligne qui affiche la progression dans la fenêtre de la console.

C#

```
fileLister.DirectoryChanged += (sender, eventArgs) =>
{
    Console.Write($"Entering '{eventArgs.CurrentSearchDirectory}'.");
    Console.WriteLine($" {eventArgs.CompletedDirs} of {eventArgs.TotalDirs}
completed...");
};
```

Nous avons vu des modèles qui sont suivis dans tout l'écosystème .NET. En apprenant ces modèles et ces conventions, vous écrirez rapidement du code C# et .NET idiomatique.

Voir aussi

- Introduction aux événements
- Conception d'événements
- Gérer et déclencher des événements

Dans le prochain article, nous allons voir quelques changements apportés à ces modèles dans la version la plus récente de .NET.

Next

Modèle d'événement .NET Core mis à jour

Article • 10/05/2023

Précédent

L'article précédent décrit les modèles d'événements les plus courants. .NET Core a un modèle plus souple. Dans cette version, la définition `EventHandler<TEventArgs>` n'a plus la contrainte selon laquelle `TEventArgs` doit être une classe dérivée de `System.EventArgs`.

Ceci accroît la flexibilité pour vous et offre une compatibilité descendante. Commençons par la flexibilité. La classe System.EventArgs introduit une méthode : `MemberwiseClone()`, qui crée une copie superficielle de l'objet. Cette méthode doit utiliser la réflexion afin d'implémenter ses fonctionnalités pour n'importe quelle classe dérivée de `EventArgs`. Ces fonctionnalités sont plus faciles à créer dans une classe dérivée spécifique. Cela signifie concrètement que la dérivation depuis System.EventArgs est une contrainte qui limite vos conceptions, mais n'offre aucun avantage supplémentaire. Vous pouvez en fait changer les définitions de `FileFoundArgs` et de `SearchDirectoryArgs` de façon à ce qu'ils ne dérivent pas de `EventArgs`. Le programme fonctionnera exactement de la même façon.

Vous pouvez également changer les `SearchDirectoryArgs` en un struct si vous faites une modification de plus :

C#

```
internal struct SearchDirectoryArgs
{
    internal string CurrentSearchDirectory { get; }
    internal int TotalDirs { get; }
    internal int CompletedDirs { get; }

    internal SearchDirectoryArgs(string dir, int totalDirs, int
completedDirs) : this()
    {
        CurrentSearchDirectory = dir;
        TotalDirs = totalDirs;
        CompletedDirs = completedDirs;
    }
}
```

La modification supplémentaire consiste à appeler le constructeur sans paramètre avant d'entrer dans le constructeur qui initialise tous les champs. Sans cet ajout, les règles de C# signalent que les propriétés sont accessibles avant qu'une valeur leur soit affectée.

Vous ne devez pas changer les `FileFoundArgs` d'une classe (type référence) en un struct (type valeur). La raison en est que le protocole de gestion d'annulation nécessite que les arguments de l'événement soient passés par référence. Si vous aviez apporté cette même modification, la classe de recherche de fichier ne pourrait jamais observer les modifications apportées par les abonnés à l'événement. Une nouvelle copie de la structure serait utilisée pour chaque abonné, et cette copie serait une copie différente de celle vue par l'objet de recherche de fichier.

Voyons ensuite comment cette modification peut avoir une compatibilité descendante. La suppression de la contrainte n'affecte aucun code existant. Les types d'arguments d'événement existants dérivent toujours de `System.EventArgs`. La compatibilité descendante est une des raisons principales pour lesquelles ils continueront à dériver de `System.EventArgs`. Les abonnés à l'événement existants seront les abonnés à un événement qui ont suivi le modèle classique.

Selon une logique similaire, aucun type d'argument d'événement créé maintenant n'aurait aucun abonné dans aucun des codes base existants. Les nouveaux types d'événements qui ne dérivent pas de `System.EventArgs` ne vont pas arrêter ces codes base.

Événements avec des abonnés asynchrones

Vous avez un modèle final à découvrir : comment écrire correctement des abonnés à l'événement qui appellent du code asynchrone. La difficulté est décrite dans l'article consacré à async et await. Les méthodes asynchrones peuvent avoir un type de retour void, mais ceci est fortement déconseillé. Quand votre code pour l'abonné à l'événement appelle une méthode asynchrone, vous n'avez d'autre choix que de créer une méthode `async void`. La signature de gestionnaire d'événements en a besoin.

Vous devez rapprocher ces deux nécessités opposées. D'une façon ou d'une autre, vous devez créer une méthode `async void` sécurisée. Les principes de base du modèle que vous devez implémenter sont les suivants :

C#

```
worker.StartWorking += async (sender, eventArgs) =>
{
    try
    {
```

```
        await DoWorkAsync();
    }
    catch (Exception e)
    {
        //Some form of logging.
        Console.WriteLine($"Async task failure: {e.ToString()}");
        // Consider gracefully, and quickly exiting.
    }
};
```

Notez d'abord que le gestionnaire est marqué en tant que gestionnaire asynchrone. Comme il est affecté à un type délégué de gestionnaire d'événements, il a un type de retour void. Cela signifie que vous devez suivre le modèle indiqué dans le gestionnaire et interdire la levée d'exceptions en dehors du contexte du gestionnaire asynchrone. Comme il ne retourne pas une tâche, aucune tâche ne peut signaler l'erreur en passant à l'état d'erreur. Comme la méthode est asynchrone, elle ne peut tout simplement pas lever l'exception. (La méthode appelante a une exécution continue, car elle est `async`.) Le comportement du runtime sera défini différemment pour différents environnements. Il peut arrêter le thread ou le processus propriétaire du thread ou laisser le processus dans un état indéterminé. Tous ces résultats potentiels sont très indésirables.

C'est pourquoi vous devez encapsuler l'instruction await pour la tâche asynchrone dans votre propre bloc try. Si elle entraîne une erreur dans une tâche, vous pouvez consigner cette erreur. Si c'est une erreur dont votre application ne peut pas récupérer, vous pouvez quitter le programme rapidement et normalement.

Ce sont les principales mises à jour du modèle d'événement .NET. Vous pouvez voir de nombreux exemples des versions précédentes dans les bibliothèques avec lesquelles vous travaillez. Il est cependant important de bien comprendre ce que sont les derniers modèles.

Le prochain article de cette série vous permet de faire la distinction entre l'utilisation de `delegates` et de `events` dans vos conceptions. Ces sont des concepts similaires : cet article vous aide à prendre la bonne décision pour vos programmes.

Next

Différenciation des délégués et des événements

Article • 10/05/2023

Précédent

Les développeurs qui découvrent la plateforme .NET Core éprouvent souvent des difficultés quand il faut choisir entre une conception basée sur les `delegates` et une conception basée sur les `events`. Choisir entre des délégués ou des événements est souvent difficile, car les deux fonctionnalités linguistiques sont similaires. Les événements sont même générés à l'aide de la prise en charge linguistique des délégués.

Elles offrent toutes deux un scénario de liaison tardive : elles activent les scénarios où un composant communique en appelant une méthode connue uniquement au moment de l'exécution. Toutes deux prennent en charge les méthodes à abonnés uniques et multiples. On utilise parfois le terme de prise en charge singlecast et multicast. Toutes deux prennent en charge une syntaxe similaire pour l'ajout et la suppression de gestionnaires. Pour finir, le déclenchement d'un événement et l'appel d'un délégué utilisent exactement la même syntaxe d'appel de méthode. Qui plus est, elles prennent toutes deux en charge la même syntaxe de méthode `Invoke()` pour une utilisation avec l'opérateur `?.`.

Avec tous ces similitudes, il est facile d'avoir des difficultés à déterminer quand utiliser l'une ou l'autre.

La détection d'événements est facultative

Pour déterminer laquelle de ces deux fonctionnalités utiliser, il convient avant tout de savoir s'il doit y avoir un abonné attaché ou non. Si votre code doit appeler le code fourni par l'abonné, vous devez utiliser une conception basée sur les délégués lorsque vous devez implémenter le rappel. Si votre code peut effectuer tout son travail sans appeler les abonnés, vous devez utiliser un modèle basé sur les événements.

Examinez les exemples fournis dans cette section. Le code que vous avez créé à l'aide de `List.Sort()` doit disposer d'une fonction de comparaison pour trier correctement les éléments. Des requêtes LINQ doivent être fournies avec les délégués pour identifier les éléments à retourner. Tous deux ont une conception basée sur des délégués.

Prenez l'événement `Progress`. Il signale la progression d'une tâche. La tâche se poursuit même s'il n'y a aucun détecteur. `FileSearcher` constitue un autre exemple. Il trouvera

tous les fichiers recherchés même si aucun abonné aux événements n'est attaché. Les contrôles de l'expérience utilisateur fonctionneront toujours correctement, même si aucun abonné n'est à l'écoute des événements. Ils utilisent tous deux des conceptions basées sur les événements.

Les valeurs de retour nécessitent des délégués

Une autre considération à prendre en compte est le prototype de méthode souhaité pour votre méthode déléguée. Comme nous l'avons vu, les délégués utilisés pour les événements ont tous un type de retour void. Nous avons également vu qu'il existait des idiomes pour créer des gestionnaires d'événements qui repassent des informations aux sources d'événements en modifiant des propriétés de l'objet d'argument d'événement. Bien que ces idiomes fonctionnent, ils ne sont pas aussi naturels que le retour d'une valeur à partir d'une méthode.

Notez que ces heuristiques peuvent souvent être toutes deux présentes : si votre méthode déléguée retourne une valeur, elle aura probablement un impact sur l'algorithme.

Les événements ont un appel privé

Les classes autres que celles dans lesquelles un événement est contenu peuvent uniquement ajouter et supprimer des écouteurs d'événements ; seule la classe contenant l'événement peut appeler l'événement. Les événements sont généralement des membres de classe publique. Par comparaison, les délégués sont souvent transmis en tant que paramètres et stockés en tant que membres de classe privée, voire ne sont pas du tout stockés.

Les détecteurs d'événements ont souvent des durées de vie plus longues

Le fait que ces écouteurs d'événements aient des durées de vie plus longues est une justification légèrement moins pertinente. Toutefois, vous constaterez peut-être que les conceptions basées sur les événements sont plus naturelles quand la source de l'événement déclenche des événements durant une période prolongée. Vous pouvez voir des exemples de conception basée sur des événements pour les contrôles d'expérience utilisateur sur de nombreux systèmes. Une fois que vous vous êtes abonné à un événement, la source de l'événement peut déclencher des événements pendant

toute la durée de vie du programme. (Vous pouvez vous désabonner à des événements quand vous n'en avez plus besoin.)

Comparez cela à de nombreuses conceptions basées sur les délégués, où un délégué est utilisé comme argument d'une méthode et le délégué n'est pas utilisé après le retour de cette méthode.

Évaluez avec soin

Les considérations ci-dessus ne sont pas des règles absolues. Il s'agit plutôt de conseils qui vous aideront à décider de ce qui convient le mieux à votre utilisation particulière. Les deux fonctionnalités étant similaires, vous pouvez même les prototyper et évaluer celle dont l'utilisation serait plus naturelle. Elles gèrent également toutes deux les scénarios de liaison tardive. Utilisez celle qui correspond le mieux à votre conception.

LINQ (Language-Integrated Query)

Article • 24/02/2023

LINQ (Language-Integrated Query) est le nom d'un ensemble de technologies basé sur l'intégration de fonctions de requête directement dans le langage C#. En règle générale, les requêtes de données sont exprimées comme de simples chaînes, sans vérification de type au moment de l'exécution, ni prise en charge IntelliSense. En outre, vous devez apprendre un langage de requête différent pour chaque type de source de données : bases de données SQL, documents XML, services web, etc. Avec LINQ, une requête est une construction de langage de premier ordre, comme les classes, les méthodes et les événements.

Pour un développeur qui écrit des requêtes, la partie « intégrée au langage » la plus visible de LINQ est l'expression de requête. Les expressions de requête sont écrites selon une *syntaxe de requête* déclarative. En utilisant la syntaxe de requête, vous pouvez effectuer des opérations de filtrage, de classement et de regroupement sur des sources de données avec un minimum de code. Vous utilisez les mêmes modèles d'expression de requête de base pour interroger et transformer des données dans les bases de données SQL, les jeux de données ADO .NET, les flux et documents XML et les collections .NET.

L'exemple suivant montre l'opération de requête complète. L'opération complète comprend la création d'une source de données, la définition de l'expression de requête et l'exécution de la requête dans une instruction `foreach`.

C#

```
// Specify the data source.
int[] scores = { 97, 92, 81, 60 };

// Define the query expression.
IEnumerable<int> scoreQuery =
    from score in scores
    where score > 80
    select score;

// Execute the query.
foreach (int i in scoreQuery)
{
    Console.Write(i + " ");
}

// Output: 97 92 81
```

Vue d'ensemble des expressions de requête

- Les expressions de requête peuvent être utilisées pour interroger et transformer des données à partir de n'importe quelle source de données compatible LINQ. Par exemple, une même requête peut récupérer des données d'une base de données SQL et générer un flux XML en sortie.

- Les expressions de requête sont faciles à saisir car elles utilisent de nombreuses constructions familières du langage C#.

- Les variables d'une expression de requête sont toutes fortement typées, même si, dans de nombreux cas, il n'est pas nécessaire de fournir le type explicitement, car le compilateur peut le déduire. Pour plus d'informations, consultez la page Relations entre les types dans les opérations de requête LINQ.

- Une requête ne s'exécute pas tant que vous n'avez pas itéré la variable de requête, par exemple dans une instruction `foreach`. Pour plus d'informations, consultez Introduction aux requêtes LINQ.

- Lors de la compilation, les expressions de requête sont converties en appels de méthode d'opérateur de requête standard en fonction des règles définies dans la spécification du langage C#. Toute requête exprimable avec la syntaxe de requête peut également être exprimée avec la syntaxe de méthode. Toutefois, dans la plupart des cas, la syntaxe de requête est plus lisible et plus concise. Pour plus d'informations, consultez les pages Spécification du langage C# et Vue d'ensemble des opérateurs de requête standard.

- En règle générale, lorsque vous écrivez des requêtes LINQ, nous vous recommandons d'utiliser la syntaxe de requête dans la mesure du possible et la syntaxe de méthode si nécessaire. Il n'y a aucune différence de sémantique ou de performances entre les deux formats. Les expressions de requête sont souvent plus lisibles que les expressions équivalentes écrites avec la syntaxe de méthode.

- Certaines opérations de requête, comme Count ou Max, n'ont pas d'expression de requête équivalente et doivent par conséquent être exprimées sous la forme d'un appel de méthode. La syntaxe de méthode peut être combinée avec la syntaxe de requête de différentes manières. Pour plus d'informations, consultez la page Syntaxe de requête et syntaxe de méthode dans LINQ.

- Les expressions de requête peuvent être compilées dans des arborescences d'expressions ou des délégués, selon le type auquel la requête est appliquée. Les requêtes IEnumerable<T> sont compilées en délégués. Les requêtes IQueryable et

IQueryable<T> sont compilées en arborescences d'expression. Pour plus d'informations, consultez la page Arborescences d'expressions.

Étapes suivantes

Pour approfondir votre connaissance de LINQ, commencez par vous familiariser avec certains concepts de base expliqués sur la page Principes de base des expressions de requête, puis lisez la documentation relative à la technologie LINQ qui vous intéresse :

- Documents XML : LINQ to XML
- ADO.NET Entity Framework : LINQ to Entities
- Collections, fichiers, chaînes, etc. .NET : LINQ to Objects

Pour approfondir votre compréhension de LINQ en général, consultez la page LINQ en C#.

Pour commencer à travailler avec LINQ en C#, consultez le didacticiel Travailler avec LINQ.

Concepts de base des expressions de requête

Article • 07/04/2023

Cet article présente les concepts fondamentaux liés aux expressions de requête en C#.

Qu'est-ce qu'une requête et quel est son rôle ?

Une *requête* est un jeu d'instructions qui décrit quelles données doivent être récupérées à partir d'une source (ou de sources) de données fournie, et quelles forme et organisation les données retournées doivent avoir. Une requête est distincte des résultats qu'elle génère.

De manière générale, les données sources sont organisées logiquement comme une séquence d'éléments du même type. Par exemple, une table de base de données SQL contient une séquence de lignes. Un fichier XML contient une « séquence » d'éléments XML (bien que ceux-ci soient organisés hiérarchiquement dans une arborescence). Une collection en mémoire contient une séquence d'objets.

Du point de vue d'une application, le type et la structure spécifiques des données sources d'origine ne sont pas importants. L'application voit toujours les données sources comme une collection IEnumerable<T> ou IQueryable<T>. Par exemple, dans LINQ to XML, les données sources sont rendues visibles sous la forme d'un `IEnumerable`<XElement>.

Compte tenu de cette séquence source, une requête peut effectuer l'une des trois actions suivantes :

- Récupérer un sous-ensemble des éléments pour générer une nouvelle séquence sans modifier les éléments individuels. La requête peut alors trier ou regrouper la séquence retournée de plusieurs façons, comme indiqué dans l'exemple suivant (supposons que `scores` est un `int[]`) :

  C#

  ```
  IEnumerable<int> highScoresQuery =
      from score in scores
      where score > 80
      orderby score descending
      select score;
  ```

- Récupérer une séquence d'éléments comme dans l'exemple précédent, mais les transformer en un nouveau type d'objet. Par exemple, une requête peut récupérer uniquement les noms de certains enregistrements de clients dans une source de données. Elle peut aussi récupérer l'enregistrement complet, puis l'utiliser pour construire un autre type d'objet en mémoire, voire des données XML, avant de générer la séquence de résultat final. L'exemple suivant affiche une projection d'un `int` en `string`. Notez le nouveau type de `highScoresQuery`.

  ```
  C#

  IEnumerable<string> highScoresQuery2 =
      from score in scores
      where score > 80
      orderby score descending
      select $"The score is {score}";
  ```

- Récupérer une valeur singleton sur les données sources, par exemple :

  - Nombre d'éléments qui correspondent à une certaine condition.
  - Élément qui a la valeur la plus grande ou la plus petite.
  - Premier élément qui correspond à une condition, ou somme de valeurs particulières dans un jeu d'éléments spécifié. Par exemple, la requête suivante retourne le nombre de notes supérieures à 80 à partir du tableau d'entiers `scores` :

    ```
    C#

    int highScoreCount = (
        from score in scores
        where score > 80
        select score
    ).Count();
    ```

    Dans l'exemple précédent, notez l'utilisation de parenthèses autour de l'expression de requête avant l'appel à la méthode `Count`. Vous pouvez également exprimer ceci en utilisant une nouvelle variable pour stocker le résultat concret. Cette technique est plus explicite, car elle conserve la variable qui stocke la requête à un emplacement distinct de la requête qui stocke un résultat.

    ```
    C#
    ```

```
IEnumerable<int> highScoresQuery3 =
    from score in scores
    where score > 80
    select score;

int scoreCount = highScoresQuery3.Count();
```

Dans l'exemple précédent, la requête est exécutée dans l'appel à `Count`, car `Count` doit effectuer une itération sur les résultats afin de déterminer le nombre d'éléments retournés par `highScoresQuery`.

Qu'est-ce qu'une expression de requête ?

Une *expression de requête* est une requête exprimée dans la syntaxe de la requête. Une expression de requête est une construction de langage de premier ordre. Elle est similaire à toute autre expression et peut être utilisée dans tous les contextes dans lesquels une expression C# est valide. Une expression de requête se compose d'un jeu de clauses écrit dans une syntaxe déclarative semblable à du SQL ou du XQuery. Chaque clause contient à son tour une ou plusieurs expressions C#, et ces expressions peuvent elles-mêmes être une expression de requête ou contenir une expression de requête.

Une expression de requête doit commencer par une clause from et doit se terminer par une clause select ou group. Entre la première clause `from` et la dernière clause `select` ou `group`, elle peut contenir une ou plusieurs clauses facultatives : where, orderby, join, let et même d'autres clauses from. Vous pouvez également utiliser le mot clé into pour que le résultat d'une clause `join` ou `group` puisse servir de source pour des clauses de requête supplémentaires dans la même expression de requête.

Variable de requête

Dans LINQ, une variable de requête correspond à n'importe quelle variable qui stocke une *requête* au lieu des *résultats* d'une requête. Plus spécifiquement, une variable de requête est toujours un type énumérable qui produit une séquence d'éléments quand elle est itérée dans une instruction `foreach` ou un appel direct à sa méthode `IEnumerator.MoveNext`.

L'exemple de code suivant montre une expression de requête simple avec une source de données, une clause de filtrage, une clause de classement et aucune transformation des éléments sources. La clause `select` termine la requête.

```
C#
```

```
// Data source.
int[] scores = { 90, 71, 82, 93, 75, 82 };

// Query Expression.
IEnumerable<int> scoreQuery = //query variable
    from score in scores //required
    where score > 80 // optional
    orderby score descending // optional
    select score; //must end with select or group

// Execute the query to produce the results
foreach (int testScore in scoreQuery)
{
    Console.WriteLine(testScore);
}

// Output: 93 90 82 82
```

Dans l'exemple précédent, `scoreQuery` est une *variable de requête* qui est parfois désignée simplement sous le nom de *requête*. La variable de requête ne stocke aucune donnée de résultat réelle, générée dans la boucle `foreach`. De plus, quand l'instruction `foreach` s'exécute, les résultats de la requête ne sont pas retournés via la variable de requête `scoreQuery`. Au lieu de cela, ils sont retournés via la variable d'itération `testScore`. La variable `scoreQuery` peut être itérée dans une deuxième boucle `foreach`. Elle générera les mêmes résultats tant que ni elle ni la source de données ne sont modifiées.

Une variable de requête peut stocker une requête exprimée dans la syntaxe de requête ou dans la syntaxe de méthode, ou dans une combinaison des deux. Dans les exemples suivants, `queryMajorCities` et `queryMajorCities2` sont des variables de requête :

C#

```
//Query syntax
IEnumerable<City> queryMajorCities =
    from city in cities
    where city.Population > 100000
    select city;

// Method-based syntax
IEnumerable<City> queryMajorCities2 = cities.Where(c => c.Population >
100000);
```

En revanche, les deux exemples suivants affichent des variables qui ne sont pas des variables de requête, même si chacune d'entre elles est initialisée avec une requête. Il ne s'agit pas de variables de requête, car elles stockent des résultats :

C#

```
int highestScore = (
    from score in scores
    select score
).Max();

// or split the expression
IEnumerable<int> scoreQuery =
    from score in scores
    select score;

int highScore = scoreQuery.Max();
// the following returns the same result
highScore = scores.Max();
```

C#

```
List<City> largeCitiesList = (
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city
).ToList();

// or split the expression
IEnumerable<City> largeCitiesQuery =
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city;

List<City> largeCitiesList2 = largeCitiesQuery.ToList();
```

Pour plus d'informations sur les différents modes d'expression de requêtes, consultez Syntaxe de requête et syntaxe de méthode dans LINQ.

Types explicites et implicites de variables de requête

Cette documentation fournit généralement le type explicite de la variable de requête pour afficher la relation de type entre la variable de requête et la clause select. Toutefois, vous pouvez également utiliser le mot clé var pour indiquer au compilateur de déduire le type d'une variable de requête (ou toute autre variable locale) au moment de la compilation. Par exemple, l'exemple de requête expliqué précédemment dans cette rubrique peut également être exprimé à l'aide de types implicites :

C#

```
// Use of var is optional here and in all queries.
// queryCities is an IEnumerable<City> just as
// when it is explicitly typed.
var queryCities =
    from city in cities
    where city.Population > 100000
    select city;
```

Pour plus d'informations, consultez Variables locales implicitement typées et Relations des types dans des opérations de requête LINQ.

Démarrage d'une expression de requête

Une expression de requête doit commencer par une clause `from`. Elle spécifie une source de données avec une variable de portée. La variable de portée représente chaque élément consécutif dans la séquence source comme la séquence source parcourue. La variable de portée est fortement typée selon le type d'éléments dans la source de données. Dans l'exemple suivant, comme `countries` est un tableau d'objets `Country`, la variable de portée est également de type `Country`. Étant donné que la variable de portée est fortement typée, vous pouvez utiliser l'opérateur point pour accéder à tous les membres disponibles du type.

C#

```
IEnumerable<Country> countryAreaQuery =
    from country in countries
    where country.Area > 500000 //sq km
    select country;
```

La variable de portée est dans la portée tant que la requête n'est pas quittée avec un point-virgule ou une clause continuation.

Une expression de requête peut contenir plusieurs clauses `from`. Utilisez des clauses `from` supplémentaires quand chaque élément de la séquence source est lui-même une collection ou contient une collection. Par exemple, supposons que vous avez une collection d'objets `Country` et que chacun d'entre eux contient une collection d'objets `City` nommés `Cities`. Pour interroger les objets `City` dans chaque `Country`, utilisez deux clauses `from` comme indiqué ici :

C#

```
IEnumerable<City> cityQuery =
    from country in countries
```

```
    from city in country.Cities
    where city.Population > 10000
    select city;
```

Pour plus d'informations, consultez from, clause.

Fin d'une expression de requête

Une expression de requête doit se terminer par une clause `group` ou une clause `select`.

group, clause

Utilisez la clause `group` pour générer une séquence de groupes organisée par une clé que vous spécifiez. La clé peut correspondre à tout type de données. Par exemple, la requête suivante crée une séquence de groupes qui contient un ou plusieurs objets `Country` et dont la clé est un type `char` avec la valeur étant la première lettre des noms des pays.

C#

```
var queryCountryGroups =
    from country in countries
    group country by country.Name[0];
```

Pour plus d'informations sur le regroupement, consultez group, clause.

select, clause

Utilisez la clause `select` pour générer tous les autres types de séquences. Une clause `select` simple génère simplement une séquence du même type d'objets que les objets contenus dans la source de données. Dans cet exemple, la source de données contient des objets `Country`. La clause `orderby` trie simplement les éléments dans un nouvel ordre et la clause `select` génère une séquence des objets `Country` réorganisés.

C#

```
IEnumerable<Country> sortedQuery =
    from country in countries
    orderby country.Area
    select country;
```

La clause `select` peut être utilisée pour transformer des données sources en séquences de nouveaux types. Cette transformation est également appelée *projection*. Dans l'exemple suivant, la clause `select` *projette* une séquence de types anonymes qui contient uniquement un sous-ensemble des champs dans l'élément d'origine. Notez que les nouveaux objets sont initialisés à l'aide d'un initialiseur d'objet.

C#

```
// Here var is required because the query
// produces an anonymous type.
var queryNameAndPop =
    from country in countries
    select new
    {
        Name = country.Name,
        Pop = country.Population
    };
```

Pour plus d'informations sur toutes les façons d'utiliser une clause `select` pour transformer des données sources, consultez select, clause.

Continuations avec *into*

Vous pouvez utiliser le mot clé `into` dans une clause `select` ou `group` pour créer un identificateur temporaire qui stocke une requête. Procédez ainsi quand vous devez effectuer des opérations de requête supplémentaires sur une requête après une opération de regroupement ou de sélection. Dans l'exemple suivant, les `countries` sont regroupés en fonction du remplissage, par plages de 10 millions. Une fois ces groupes créés, des clauses supplémentaires éliminent des groupes par filtrage, puis trient les groupes par ordre croissant. Pour effectuer ces opérations supplémentaires, la continuation représentée par `countryGroup` est requise.

C#

```
// percentileQuery is an IEnumerable<IGrouping<int, Country>>
var percentileQuery =
    from country in countries
    let percentile = (int)country.Population / 10_000_000
    group country by percentile into countryGroup
    where countryGroup.Key >= 20
    orderby countryGroup.Key
    select countryGroup;

// grouping is an IGrouping<int, Country>
foreach (var grouping in percentileQuery)
{
```

```
    Console.WriteLine(grouping.Key);
    foreach (var country in grouping)
    {
        Console.WriteLine(country.Name + ":" + country.Population);
    }
}
```

Pour plus d'informations, consultez into.

Filtrage, classement et jointure

Entre la clause `from` initiale et la clause `select` ou `group` finale, toutes les autres clauses (`where`, `join`, `orderby`, `from`, `let`) sont facultatives. Chacune des clauses facultatives peut être utilisée zéro fois ou plusieurs fois dans un corps de requête.

where, clause

Utilisez la clause `where` pour éliminer par filtrage des éléments des données sources selon une ou plusieurs expressions de prédicat. La clause `where` de l'exemple suivant a un prédicat avec deux conditions.

C#

```
IEnumerable<City> queryCityPop =
    from city in cities
    where city.Population < 200000 && city.Population > 100000
    select city;
```

Pour plus d'informations, consultez where, clause.

orderby, clause

Utilisez la clause `orderby` pour trier les résultats par ordre croissant ou décroissant. Vous pouvez également spécifier des ordres de tri secondaires. L'exemple suivant effectue un tri principal sur les objets `country` en utilisant la propriété `Area`. Il effectue ensuite un tri secondaire en utilisant la propriété `Population`.

C#

```
IEnumerable<Country> querySortedCountries =
    from country in countries
    orderby country.Area, country.Population descending
    select country;
```

Le mot clé `ascending` est facultatif ; il s'agit de l'ordre de tri par défaut si aucun ordre n'est spécifié. Pour plus d'informations, consultez orderby, clause.

join, clause

Utilisez la clause `join` pour associer et/ou combiner des éléments d'une source de données avec des éléments d'une autre source de données en fonction d'une comparaison d'égalité entre des clés spécifiées dans chaque élément. Dans LINQ, les opérations de jointure sont effectuées sur les séquences des objets dont les éléments sont des types différents. Après avoir joint deux séquences, vous devez utiliser une instruction `select` ou `group` pour spécifier quel élément stocker dans la séquence de sortie. Vous pouvez également utiliser un type anonyme pour combiner des propriétés de chaque jeu d'éléments associés dans un nouveau type pour la séquence de sortie. L'exemple suivant associe des objets `prod` dont la propriété `Category` correspond à l'une des catégories dans le tableau de chaînes `categories`. Les produits dont `Category` ne correspond à aucune chaîne dans `categories` sont filtrés. L'instruction `select` projette un nouveau type dont les propriétés sont extraites à la fois de `cat` et de `prod`.

C#

```
var categoryQuery =
    from cat in categories
    join prod in products on cat equals prod.Category
    select new
    {
        Category = cat,
        Name = prod.Name
    };
```

Vous pouvez également effectuer une jointure groupée en stockant les résultats de l'opération `join` dans une variable temporaire à l'aide du mot clé into. Pour plus d'informations, consultez join, clause.

let, clause

Utilisez la clause `let` pour stocker le résultat d'une expression telle qu'un appel de méthode dans une nouvelle variable de portée. Dans l'exemple suivant, la variable de portée `firstName` stocke le premier élément du tableau de chaînes retourné par `Split`.

C#

```
string[] names = { "Svetlana Omelchenko", "Claire O'Donnell", "Sven
Mortensen", "Cesar Garcia" };
```

```
IEnumerable<string> queryFirstNames =
    from name in names
    let firstName = name.Split(' ')[0]
    select firstName;

foreach (string s in queryFirstNames)
{
    Console.Write(s + " ");
}

//Output: Svetlana Claire Sven Cesar
```

Pour plus d'informations, consultez let, clause.

Sous-requêtes dans une expression de requête

Une clause de requête peut elle-même contenir une expression de requête, qui est parfois connue sous le nom de *sous-requête*. Chaque sous-requête démarre avec sa propre clause `from` qui ne pointe pas nécessairement vers la même source de données dans la première clause `from`. Par exemple, la requête suivante montre une expression de requête utilisée dans l'instruction select pour récupérer les résultats d'une opération de regroupement.

C#

```
var queryGroupMax =
    from student in students
    group student by student.Year into studentGroup
    select new
    {
        Level = studentGroup.Key,
        HighestScore = (
            from student2 in studentGroup
            select student2.ExamScores.Average()
        ).Max()
    };
```

Pour plus d'informations, consultez le Guide pratique pour effectuer une sous-requête sur une opération de regroupement.

Voir aussi

- Guide de programmation C#
- LINQ (Language-Integrated Query)
- Mots clés de requête (LINQ)

- Vue d’ensemble des opérateurs de requête standard

LINQ (Language-Integrated Query)

Article • 24/02/2023

LINQ (Language-Integrated Query) est le nom d'un ensemble de technologies basé sur l'intégration de fonctions de requête directement dans le langage C#. En règle générale, les requêtes de données sont exprimées comme de simples chaînes, sans vérification de type au moment de l'exécution, ni prise en charge IntelliSense. En outre, vous devez apprendre un langage de requête différent pour chaque type de source de données : bases de données SQL, documents XML, services web, etc. Avec LINQ, une requête est une construction de langage de premier ordre, comme les classes, les méthodes et les événements.

Pour un développeur qui écrit des requêtes, la partie « intégrée au langage » la plus visible de LINQ est l'expression de requête. Les expressions de requête sont écrites selon une *syntaxe de requête* déclarative. En utilisant la syntaxe de requête, vous pouvez effectuer des opérations de filtrage, de classement et de regroupement sur des sources de données avec un minimum de code. Vous utilisez les mêmes modèles d'expression de requête de base pour interroger et transformer des données dans les bases de données SQL, les jeux de données ADO .NET, les flux et documents XML et les collections .NET.

L'exemple suivant montre l'opération de requête complète. L'opération complète comprend la création d'une source de données, la définition de l'expression de requête et l'exécution de la requête dans une instruction `foreach`.

C#

```
// Specify the data source.
int[] scores = { 97, 92, 81, 60 };

// Define the query expression.
IEnumerable<int> scoreQuery =
    from score in scores
    where score > 80
    select score;

// Execute the query.
foreach (int i in scoreQuery)
{
    Console.Write(i + " ");
}

// Output: 97 92 81
```

Vue d'ensemble des expressions de requête

- Les expressions de requête peuvent être utilisées pour interroger et transformer des données à partir de n'importe quelle source de données compatible LINQ. Par exemple, une même requête peut récupérer des données d'une base de données SQL et générer un flux XML en sortie.

- Les expressions de requête sont faciles à saisir car elles utilisent de nombreuses constructions familières du langage C#.

- Les variables d'une expression de requête sont toutes fortement typées, même si, dans de nombreux cas, il n'est pas nécessaire de fournir le type explicitement, car le compilateur peut le déduire. Pour plus d'informations, consultez la page Relations entre les types dans les opérations de requête LINQ.

- Une requête ne s'exécute pas tant que vous n'avez pas itéré la variable de requête, par exemple dans une instruction `foreach`. Pour plus d'informations, consultez Introduction aux requêtes LINQ.

- Lors de la compilation, les expressions de requête sont converties en appels de méthode d'opérateur de requête standard en fonction des règles définies dans la spécification du langage C#. Toute requête exprimable avec la syntaxe de requête peut également être exprimée avec la syntaxe de méthode. Toutefois, dans la plupart des cas, la syntaxe de requête est plus lisible et plus concise. Pour plus d'informations, consultez les pages Spécification du langage C# et Vue d'ensemble des opérateurs de requête standard.

- En règle générale, lorsque vous écrivez des requêtes LINQ, nous vous recommandons d'utiliser la syntaxe de requête dans la mesure du possible et la syntaxe de méthode si nécessaire. Il n'y a aucune différence de sémantique ou de performances entre les deux formats. Les expressions de requête sont souvent plus lisibles que les expressions équivalentes écrites avec la syntaxe de méthode.

- Certaines opérations de requête, comme Count ou Max, n'ont pas d'expression de requête équivalente et doivent par conséquent être exprimées sous la forme d'un appel de méthode. La syntaxe de méthode peut être combinée avec la syntaxe de requête de différentes manières. Pour plus d'informations, consultez la page Syntaxe de requête et syntaxe de méthode dans LINQ.

- Les expressions de requête peuvent être compilées dans des arborescences d'expressions ou des délégués, selon le type auquel la requête est appliquée. Les requêtes IEnumerable<T> sont compilées en délégués. Les requêtes IQueryable et

IQueryable<T> sont compilées en arborescences d'expression. Pour plus d'informations, consultez la page Arborescences d'expressions.

Étapes suivantes

Pour approfondir votre connaissance de LINQ, commencez par vous familiariser avec certains concepts de base expliqués sur la page Principes de base des expressions de requête, puis lisez la documentation relative à la technologie LINQ qui vous intéresse :

- Documents XML : LINQ to XML
- ADO.NET Entity Framework : LINQ to Entities
- Collections, fichiers, chaînes, etc. .NET : LINQ to Objects

Pour approfondir votre compréhension de LINQ en général, consultez la page LINQ en C#.

Pour commencer à travailler avec LINQ en C#, consultez le didacticiel Travailler avec LINQ.

Écrire des requêtes LINQ en C#

Article • 20/07/2023

La plupart des requêtes de la documentation de présentation de Langage Integrated Query (LINQ) sont écrites à l'aide de la syntaxe de requête déclarative de LINQ. Toutefois, la syntaxe de requête doit être traduite en appels de méthode pour le Common Langage Runtime (CLR) .NET lorsque le code est compilé. Ces appels de méthode appellent les opérateurs de requête standard, qui ont des noms tels que `Where`, `Select`, `GroupBy`, `Join`, `Max` et `Average`. Vous pouvez les appeler directement en utilisant la syntaxe de méthode à la place de la syntaxe de requête.

La syntaxe de requête et la syntaxe de méthode sont identiques sémantiquement, mais de nombreuses personnes trouvent la syntaxe de requête plus simple et plus facile à lire. Certaines requêtes doivent être exprimées en tant qu'appels de méthode. Par exemple, vous devez utiliser un appel de méthode pour exprimer une requête qui récupère le nombre d'éléments qui correspondent à une condition spécifiée. Vous devez également utiliser un appel de méthode pour une requête qui récupère dans une séquence source l'élément qui a la valeur maximale. En général, la documentation de référence des opérateurs de requête standard dans l'espace de noms System.Linq utilise la syntaxe de méthode. Par conséquent, même lorsque vous commencez à écrire des requêtes LINQ, il est utile de se familiariser avec l'utilisation de la syntaxe de méthode dans les requêtes et dans les expressions de requête elles-mêmes.

Méthodes d'extension d'opérateur de requête standard

L'exemple suivant présente une *expression de requête* simple et la requête sémantiquement équivalente écrite en tant que *requête fondée sur une méthode*.

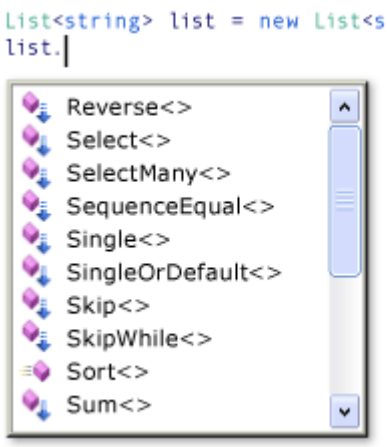
C#

```
class QueryVMethodSyntax
{
    static void Main()
    {
        int[] numbers = { 5, 10, 8, 3, 6, 12};

        //Query syntax:
        IEnumerable<int> numQuery1 =
            from num in numbers
            where num % 2 == 0
            orderby num
            select num;
```

```
        //Method syntax:
        IEnumerable<int> numQuery2 = numbers.Where(num => num % 2 ==
0).OrderBy(n => n);

        foreach (int i in numQuery1)
        {
            Console.Write(i + " ");
        }
        Console.WriteLine(System.Environment.NewLine);
        foreach (int i in numQuery2)
        {
            Console.Write(i + " ");
        }

        // Keep the console open in debug mode.
        Console.WriteLine(System.Environment.NewLine);
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/*
    Output:
    6 8 10 12
    6 8 10 12
 */
```

La sortie des deux exemples est identique. Vous pouvez voir que le type de la variable de requête est le même dans les deux formes : IEnumerable<T>.

Pour comprendre la requête basée sur une méthode, examinons-la de plus près. Du côté droit de l'expression, remarquez que la clause `where` est maintenant exprimée comme une méthode d'instance sur l'objet `numbers`, qui, comme vous vous en rappelez, a un type `IEnumerable<int>`. Si vous êtes connaissez bien l'interface IEnumerable<T> générique, vous savez qu'elle n'a pas de méthode `Where`. Toutefois, si vous appelez la liste de saisie semi-automatique IntelliSense dans l'IDE de Visual Studio, vous ne verrez pas seulement une méthode `Where`, mais de nombreuses autres méthodes telles que `Select`, `SelectMany`, `Join` et `Orderby`. Il s'agit de tous les opérateurs de requête standard.



Même s'il semble que IEnumerable<T> a été redéfini pour inclure ces méthodes supplémentaires, ce n'est pas le cas, en réalité. Les opérateurs de requête standard sont implémentés comme un nouveau type de méthode, appelé *méthodes d'extension*. Les méthodes d'extension « étendent » un type existant. Elles peuvent être appelées comme s'il s'agissait de méthodes d'instance sur le type. Les opérateurs de requête standard étendent IEnumerable<T>, si bien que vous pouvez écrire `numbers.Where(...)`.

Pour commencer à utiliser LINQ, tout ce que vous devez savoir sur les méthodes d'extension est la façon de les mettre à portée de votre application à l'aide des directives `using` appropriées. Du point de vue de votre application, une méthode d'extension et une méthode d'instance normale sont identiques.

Pour plus d'informations sur les méthodes d'extension, consultez Méthodes d'extension. Pour plus d'informations sur les opérateurs de requête standard, consultez Présentation des opérateurs de requête standard (C#). Certains fournisseurs LINQ, tels que LINQ to SQL ET LINQ to XML, implémentent leurs propres opérateurs de requête standard et des méthodes d'extension supplémentaires pour d'autres types que IEnumerable<T>.

Expressions lambda

Dans l'exemple précédent, notez que l'expression conditionnelle (`num % 2 == 0`) est passée comme argument de ligne à la méthode `Where` : `Where(num => num % 2 == 0).` Cette expression inline est appelée expression lambda. C'est un moyen pratique d'écrire du code qui devrait autrement être écrit sous une forme plus fastidieuse, comme une méthode anonyme, un délégué générique ou une arborescence d'expressions. En C#, `=>` est l'opérateur lambda, qui se lit « conduit à ». Le `num` situé à gauche de l'opérateur est la variable d'entrée qui correspond à `num` dans l'expression de requête. Le compilateur peut déduire le type de `num`, car il sait que `numbers` est un type IEnumerable<T> générique. Le corps de l'expression lambda est identique à l'expression dans la syntaxe de requête ou dans toute autre expression ou instruction C#. Il peut

inclure des appels de méthode et une autre logique complexe. La « valeur de retour » est simplement le résultat de l'expression.

Vous n'avez pas besoin d'utiliser fréquemment les expressions lambda pour commencer à utiliser LINQ. Toutefois, certaines requêtes peuvent être exprimées uniquement dans la syntaxe de méthode. Parmi elles, certaines nécessitent des expressions lambda. Une fois que vous serez plus familiarisé avec les expressions lambda, vous constaterez qu'elles sont un outil puissant et flexible dans votre boîte à outils LINQ. Pour plus d'informations, consultez Expressions lambda.

Composabilité des requêtes

Dans l'exemple de code précédent, notez que la méthode `OrderBy` est appelée à l'aide de l'opérateur point sur l'appel à `Where`. `Where` produit une séquence filtrée, puis `Orderby` agit sur cette séquence en la triant. Étant donné que les requêtes retournent un `IEnumerable`, vous les composez dans la syntaxe de méthode en chaînant les appels de méthode ensemble. Le compilateur effectue cette opération en arrière-plan lorsque vous écrivez des requêtes à l'aide de la syntaxe de requête. De plus, étant donné qu'une variable de requête ne stocke pas les résultats de la requête, vous pouvez la modifier ou l'utiliser à tout moment comme base d'une nouvelle requête, même après son exécution.

Les exemples suivants montrent des requêtes LINQ simples en utilisant chaque approche répertoriée précédemment. En général, la règle consiste à utiliser (1) dès que possible et à utiliser (2) et (3) dès que cela est nécessaire.

> ⓘ **Notes**
>
> Ces requêtes fonctionnent sur les collections simples en mémoire ; toutefois, la syntaxe de base est identique à celle utilisée dans LINQ to Entities et LINQ to XML.

Exemple – Syntaxe de requête

La méthode recommandée pour écrire la plupart des requêtes consiste à utiliser la *syntaxe de requête* pour créer des *expressions de requête*. L'exemple suivant présente trois expressions de requête. La première expression de requête montre comment filtrer ou restreindre des résultats en appliquant des conditions avec une clause `where`. Tous les éléments de la séquence source dont la valeur est supérieure à 7 ou inférieure à 3 sont retournés. La deuxième expression montre comment classer les résultats retournés.

La troisième expression montre comment regrouper des résultats en fonction d'une clé. Cette requête retourne deux groupes en fonction de la première lettre du mot.

C#

```
List<int> numbers = new() { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

// The query variables can also be implicitly typed by using var

// Query #1.
IEnumerable<int> filteringQuery =
    from num in numbers
    where num < 3 || num > 7
    select num;

// Query #2.
IEnumerable<int> orderingQuery =
    from num in numbers
    where num < 3 || num > 7
    orderby num ascending
    select num;

// Query #3.
string[] groupingQuery = { "carrots", "cabbage", "broccoli", "beans",
"barley" };
IEnumerable<IGrouping<char, string>> queryFoodGroups =
    from item in groupingQuery
    group item by item[0];
```

Notez que le type des requêtes est IEnumerable<T>. Toutes ces requêtes pourraient être écrites à l'aide de `var`, comme illustré dans l'exemple suivant :

```
var query = from num in numbers...
```

Dans chacun des exemples précédents, les requêtes ne s'exécutent pas réellement tant vous n'avez pas itéré la variable de requête dans une instruction `foreach` ou une autre instruction. Pour plus d'informations, consultez Introduction aux requêtes LINQ.

Exemple – Syntaxe de méthode

Certaines opérations de requête doivent être exprimées comme un appel de méthode. Les plus répandues de ces méthodes retournent des valeurs numériques uniques, telles que Sum, Max, Min, Average, et ainsi de suite. Ces méthodes doivent toujours être appelées en dernier dans toutes les requêtes, car elles ne représentent qu'une valeur unique et ne peuvent pas servir de source pour une opération de requête supplémentaire. L'exemple suivant présente un appel de méthode dans une expression de requête :

C#

```
List<int> numbers1 = new() { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
List<int> numbers2 = new() { 15, 14, 11, 13, 19, 18, 16, 17, 12, 10 };

// Query #4.
double average = numbers1.Average();

// Query #5.
IEnumerable<int> concatenationQuery = numbers1.Concat(numbers2);
```

Si la méthode a des paramètres Action ou Func, ceux-ci sont fournis sous la forme d'une expression lambda, comme dans l'exemple suivant :

C#

```
// Query #6.
IEnumerable<int> largeNumbersQuery = numbers2.Where(c => c > 15);
```

Dans les requêtes précédentes, seule la requête 4 s'exécute immédiatement. Cela s'explique par le fait qu'elle retourne une valeur unique et non une collection IEnumerable<T> générique. La méthode elle-même doit utiliser `foreach` pour calculer sa valeur.

Chacune des requêtes précédentes peut être écrite en utilisant des types implicites avec var, comme dans l'exemple suivant :

C#

```
// var is used for convenience in these queries
var average = numbers1.Average();
var concatenationQuery = numbers1.Concat(numbers2);
var largeNumbersQuery = numbers2.Where(c => c > 15);
```

Exemple – Syntaxe de méthode et de requête mixte

Cet exemple montre comment utiliser la syntaxe de méthode sur les résultats d'une clause de requête. Encadrez simplement l'expression de requête entre parenthèses, puis appliquez l'opérateur point et appelez la méthode. Dans l'exemple suivant, la requête 7 retourne les nombres dont la valeur est comprise entre 3 et 7. Toutefois, il est généralement préférable d'utiliser une deuxième variable pour stocker le résultat de

l'appel de méthode. De cette manière, il est moins probable que la requête ne soit confondue avec les résultats de la requête.

C#

```
// Query #7.

// Using a query expression with method syntax
int numCount1 = (
    from num in numbers1
    where num < 3 || num > 7
    select num
).Count();

// Better: Create a new variable to store
// the method call result
IEnumerable<int> numbersQuery =
    from num in numbers1
    where num < 3 || num > 7
    select num;

int numCount2 = numbersQuery.Count();
```

Comme la requête 7 retourne une valeur unique et non une collection, la requête s'exécute immédiatement.

La requête précédente peut être écrite en utilisant des types implicites avec `var`, comme suit :

C#

```
var numCount = (from num in numbers...
```

Elle peut être écrite dans la syntaxe de méthode comme suit :

C#

```
var numCount = numbers.Where(n => n < 3 || n > 7).Count();
```

Elle peut être écrite en utilisant des types explicites, comme suit :

C#

```
int numCount = numbers.Where(n => n < 3 || n > 7).Count();
```

Voir aussi

- Procédure pas à pas : écriture de requêtes en C#
- LINQ (Language-Integrated Query)
- where, clause

Interroger une collection d'objets

Article • 20/07/2023

« LINQ to Objects » fait référence à l'utilisation directe de requêtes LINQ avec n'importe quelle collection IEnumerable ou IEnumerable<T>, sans utiliser de fournisseur LINQ ou d'API intermédiaire comme LINQ to SQL ou LINQ to XML. Vous pouvez utiliser LINQ pour interroger des collections énumérables telles que List<T>, Array ou Dictionary<TKey,TValue>. La collection peut être définie par l'utilisateur ou retournée par une API .NET. Avec l'approche LINQ, vous écrivez du code déclaratif qui décrit ce que vous voulez récupérer.

De plus, les requêtes LINQ offrent trois avantages principaux par rapport aux boucles `foreach` classiques :

- Elles sont plus concises et lisibles, en particulier durant le filtrage de plusieurs conditions.
- Elles fournissent de puissantes fonctionnalités de filtrage, de classement et de regroupement avec un minimum de code d'application.
- Elles peuvent être portées vers d'autres sources de données avec peu ou pas de changements.

En général, plus l'opération que vous voulez effectuer sur les données est complexe, plus vous avez intérêt à utiliser LINQ à la place des techniques d'itération classiques.

Cet exemple montre comment effectuer une requête simple sur une liste d'objets `Student`. Chaque objet `Student` contient des informations de base sur l'étudiant et une liste qui représente les notes de l'étudiant lors de quatre examens.

> **Notes**
>
> De nombreux autres exemples de cette section utilisent la même `Student` classe et `students` la même collection.

C#

```
class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int ID { get; set; }
    public GradeLevel? Year { get; set; }
```

```
    public List<int> ExamScores { get; set; }

    public Student(string FirstName, string LastName, int ID, GradeLevel
Year, List<int> ExamScores)
    {
        this.FirstName = FirstName;
        this.LastName = LastName;
        this.ID = ID;
        this.Year = Year;
        this.ExamScores = ExamScores;
    }

    public Student(string FirstName, string LastName, int StudentID,
List<int>? ExamScores = null)
    {
        this.FirstName = FirstName;
        this.LastName = LastName;
        ID = StudentID;
        this.ExamScores = ExamScores ?? Enumerable.Empty<int>().ToList();
    }

    public static List<Student> students = new()
    {
        new(
            FirstName: "Terry", LastName: "Adams", ID: 120,
            Year: GradeLevel.SecondYear,
            ExamScores: new() { 99, 82, 81, 79 }
        ),
        new(
            "Fadi", "Fakhouri", 116,
            GradeLevel.ThirdYear,
            new() { 99, 86, 90, 94 }
        ),
        new(
            "Hanying", "Feng", 117,
            GradeLevel.FirstYear,
            new() { 93, 92, 80, 87 }
        ),
        new(
            "Cesar", "Garcia", 114,
            GradeLevel.FourthYear,
            new() { 97, 89, 85, 82 }
        ),
        new(
            "Debra", "Garcia", 115,
            GradeLevel.ThirdYear,
            new() { 35, 72, 91, 70 }
        ),
        new(
            "Hugo", "Garcia", 118,
            GradeLevel.SecondYear,
            new() { 92, 90, 83, 78 }
        ),
        new(
            "Sven", "Mortensen", 113,
```

```
            GradeLevel.FirstYear,
            new() { 88, 94, 65, 91 }
        ),
        new(
            "Claire", "O'Donnell", 112,
            GradeLevel.FourthYear,
            new() { 75, 84, 91, 39 }
        ),
        new(
            "Svetlana", "Omelchenko", 111,
            GradeLevel.SecondYear,
            new() { 97, 92, 81, 60 }
        ),
        new(
            "Lance", "Tucker", 119,
            GradeLevel.ThirdYear,
            new() { 68, 79, 88, 92 }
        ),
        new(
            "Michael", "Tucker", 122,
            GradeLevel.FirstYear,
            new() { 94, 92, 91, 91 }
        ),
        new(
            "Eugene", "Zabokritski", 121,
            GradeLevel.FourthYear,
            new() { 96, 85, 91, 60 }
        )
    };
}

enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};
```

Exemple

La requête suivante retourne les étudiants qui ont reçu une note supérieure ou égale à 90 lors de leur premier examen.

C#

```
void QueryHighScores(int exam, int score)
{
    var highScores =
        from student in students
        where student.ExamScores[exam] > score
```

```
        select new
        {
            Name = student.FirstName,
            Score = student.ExamScores[exam]
        };

    foreach (var item in highScores)
    {
        Console.WriteLine($"{item.Name,-15}{item.Score}");
    }
}

QueryHighScores(1, 90);
```

Cette requête est volontairement simple pour vous permettre de faire des essais. Par exemple, vous pouvez essayer plus de conditions dans la clause `where` ou utiliser une clause `orderby` pour trier les résultats.

Classification des opérateurs de requête standard en fonction de leur mode d'exécution

Les implémentations LINQ to Objects des méthodes d'opérateur de requête standard s'exécutent de deux manières principales : immédiate ou différée. En outre, les opérateurs de requête qui utilisent l'exécution différée peuvent être divisés en deux catégories : ceux prenant en charge la diffusion en continu et ceux ne la prenant pas en charge. Le fait de connaître le mode d'exécution des différents opérateurs de requête peut vous aider à comprendre les résultats obtenus à partir d'une requête donnée. Ceci est particulièrement vrai si la source de données change ou si vous générez une requête sur une autre requête. Cette rubrique classe les opérateurs de requête standard selon leur mode d'exécution.

Immédiat

L'exécution immédiate signifie que la source de données est lue et que l'opération est effectuée une seule fois. Tous les opérateurs de requête standard qui retournent un résultat scalaire s'exécutent immédiatement. Vous pouvez forcer l'exécution d'une requête immédiatement à l'aide des méthodes Enumerable.ToList ou Enumerable.ToArray. L'exécution immédiate permet la réutilisation des résultats de la requête, et non la déclaration de requête. Les résultats sont récupérés une fois, puis stockés pour une utilisation ultérieure.

Différé

L'exécution différée signifie que l'opération n'est pas effectuée au point où la requête est déclarée dans le code. L'opération est effectuée uniquement quand la variable de requête est énumérée, par exemple à l'aide d'une instruction `foreach`. Cela signifie que les résultats de l'exécution de la requête dépendent du contenu de la source de données au moment de l'exécution de la requête, et non au moment de sa définition. Si la variable de requête est énumérée plusieurs fois, les résultats peuvent différer chaque fois. Presque tous les opérateurs de requête standard dont le type de retour est IEnumerable<T> ou IOrderedEnumerable<TElement> s'exécutent de manière différée. L'exécution différée permet la réutilisation des requêtes, car la requête extrait les données mises à jour de la source de données chaque fois que les résultats de la requête sont itérés.

Les opérateurs de requête qui utilisent l'exécution différée peuvent également être classés comme prenant en charge la diffusion en continu ou non.

Diffusion en continu

Les opérateurs prenant en charge la diffusion en continu n'ont pas à lire toutes les données sources avant de générer des éléments. Au moment de l'exécution, un opérateur prenant en charge la diffusion en continu effectue son opération sur chaque élément source à mesure qu'il est lu et génère l'élément si nécessaire. Ce type d'opérateur continue à lire des éléments source jusqu'à ce qu'un élément de résultat puisse être produit. Cela signifie que plusieurs éléments source peuvent être lus pour produire un élément de résultat.

Requêtes qui ne sont pas des requêtes de streaming

Les opérateurs ne prenant pas en charge la diffusion en continu doivent lire toutes les données source avant de générer un élément de résultat. Les opérations telles que le tri ou le regroupement font partie de cette catégorie. Au moment de l'exécution, les opérateurs de requête ne prenant pas en charge la diffusion en continu lisent toutes les données source, les placent dans une structure de données, puis effectuent l'opération et génèrent les éléments de résultat.

Tableau de classification

Le tableau suivant classe chaque méthode d'opérateur de requête standard en fonction de son mode d'exécution.

> ⓘ **Notes**

Si un opérateur est présent dans deux colonnes, deux séquences d'entrée sont impliquées dans l'opération et chacune d'elles est évaluée différemment. Dans ces cas, la première séquence dans la liste de paramètres est toujours évaluée de façon différée, avec diffusion en continu.

Opérateur de requête standard	Type de retour	Exécution immédiate	Exécution différée avec diffusion en continu	Exécution différée sans diffusion en continu
Aggregate	`TSource`	X		
All	Boolean	X		
Any	Boolean	X		
AsEnumerable	IEnumerable<T>		X	
Average	Valeur numérique unique	X		
Cast	IEnumerable<T>		X	
Concat	IEnumerable<T>		X	
Contains	Boolean	X		
Count	Int32	X		
DefaultIfEmpty	IEnumerable<T>		X	
Distinct	IEnumerable<T>		X	
ElementAt	`TSource`	X		
ElementAtOrDefault	`TSource?`	X		
Empty	IEnumerable<T>	X		
Except	IEnumerable<T>		X	X
First	`TSource`	X		
FirstOrDefault	`TSource?`	X		
GroupBy	IEnumerable<T>			X
GroupJoin	IEnumerable<T>		X	X
Intersect	IEnumerable<T>		X	X

Opérateur de requête standard	Type de retour	Exécution immédiate	Exécution différée avec diffusion en continu	Exécution différée sans diffusion en continu
Join	IEnumerable<T>		X	X
Last	`TSource`	X		
LastOrDefault	`TSource?`	X		
LongCount	Int64	X		
Max	Valeur numérique unique, `TSource`, ou `TResult?`	X		
Min	Valeur numérique unique, `TSource`, ou `TResult?`	X		
OfType	IEnumerable<T>		X	
OrderBy	IOrderedEnumerable<TElement>			X
OrderByDescending	IOrderedEnumerable<TElement>			X
Range	IEnumerable<T>		X	
Repeat	IEnumerable<T>		X	
Reverse	IEnumerable<T>			X
Select	IEnumerable<T>		X	
SelectMany	IEnumerable<T>		X	
SequenceEqual	Boolean	X		
Single	`TSource`	X		
SingleOrDefault	`TSource?`	X		
Skip	IEnumerable<T>		X	
SkipWhile	IEnumerable<T>		X	
Sum	Valeur numérique unique	X		
Take	IEnumerable<T>		X	
TakeWhile	IEnumerable<T>		X	

Opérateur de requête standard	Type de retour	Exécution immédiate	Exécution différée avec diffusion en continu	Exécution différée sans diffusion en continu
ThenBy	IOrderedEnumerable<TElement>			X
ThenByDescending	IOrderedEnumerable<TElement>			X
ToArray	Tableau `TSource[]`	X		
ToDictionary	Dictionary<TKey,TValue>	X		
ToList	IList<T>	X		
ToLookup	ILookup<TKey,TElement>	X		
Union	IEnumerable<T>		X	
Where	IEnumerable<T>		X	

Voir aussi

- LINQ (Language-Integrated Query)
- Interpolation de chaîne

Comment retourner une requête à partir d'une méthode

Article • 07/04/2023

Cet exemple montre comment retourner une requête à partir d'une méthode en tant que valeur de retour et en tant que paramètre `out`.

Les objets de requête sont composables, ce qui signifie que vous pouvez retourner une requête à partir d'une méthode. Les objets qui représentent des requêtes ne stockent pas la collection résultante, mais plutôt les étapes pour générer les résultats lorsque cela est nécessaire. L'avantage de retourner des objets de requête à partir de méthodes est qu'il est possible de les composer ou modifier encore. Par conséquent, toute valeur de retour ou paramètre `out` d'une méthode qui retourne une requête doit également avoir ce type. Si une méthode matérialise une requête en un type List<T> ou Array concret, elle est considérée comme retournant les résultats de la requête plutôt que la requête elle-même. Une variable de requête retournée à partir d'une méthode peut encore être composée ou modifiée.

Exemple

Dans l'exemple suivant, la première méthode retourne une requête comme valeur de retour et la seconde méthode retourne une requête comme paramètre `out`. Notez que, dans les deux cas, une requête est retournée et non pas les résultats d'une requête.

C#

```
// QueryMethod1 returns a query as its value.
IEnumerable<string> QueryMethod1(int[] ints) =>
    from i in ints
    where i > 4
    select i.ToString();

// QueryMethod2 returns a query as the value of the out parameter returnQ
void QueryMethod2(int[] ints, out IEnumerable<string> returnQ) =>
    returnQ =
        from i in ints
        where i < 4
        select i.ToString();

int[] nums = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// QueryMethod1 returns a query as the value of the method.
var myQuery1 = QueryMethod1(nums);
```

```
// Query myQuery1 is executed in the following foreach loop.
Console.WriteLine("Results of executing myQuery1:");
// Rest the mouse pointer over myQuery1 to see its type.
foreach (var s in myQuery1)
{
    Console.WriteLine(s);
}

// You also can execute the query returned from QueryMethod1
// directly, without using myQuery1.
Console.WriteLine("\nResults of executing myQuery1 directly:");
// Rest the mouse pointer over the call to QueryMethod1 to see its
// return type.
foreach (var s in QueryMethod1(nums))
{
    Console.WriteLine(s);
}

// QueryMethod2 returns a query as the value of its out parameter.
QueryMethod2(nums, out IEnumerable<string> myQuery2);

// Execute the returned query.
Console.WriteLine("\nResults of executing myQuery2:");
foreach (var s in myQuery2)
{
    Console.WriteLine(s);
}

// You can modify a query by using query composition. In this case, the
// previous query object is used to create a new query object; this new
object
// will return different results than the original query object.
myQuery1 =
    from item in myQuery1
    orderby item descending
    select item;

// Execute the modified query.
Console.WriteLine("\nResults of executing modified myQuery1:");
foreach (var s in myQuery1)
{
    Console.WriteLine(s);
}
```

Voir aussi

- LINQ (Language-Integrated Query)

Stocker les résultats d'une requête dans la mémoire

Article • 07/04/2023

Une requête est en fait un ensemble d'instructions permettant de récupérer et d'organiser des données. Les requêtes sont exécutées de manière différée, à mesure que chaque élément dans le résultat est demandé. Lorsque vous utilisez `foreach` pour effectuer une itération sur les résultats, les éléments sont retournés lorsque vous y accédez. Pour évaluer une requête et stocker ses résultats sans exécuter une boucle `foreach`, appelez simplement l'une des méthodes suivantes sur la variable de requête :

- ToList
- ToArray
- ToDictionary
- ToLookup

Lorsque vous stockez les résultats de requête, nous vous recommandons d'assigner l'objet de collection retourné à une nouvelle variable, comme indiqué dans l'exemple suivant :

Exemple

C#

```
List<int> numbers = new() { 1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };

IEnumerable<int> queryFactorsOfFour =
    from num in numbers
    where num % 4 == 0
    select num;

// Store the results in a new variable
// without executing a foreach loop.
List<int> factorsofFourList = queryFactorsOfFour.ToList();

// Read and write from the newly created list to demonstrate that it holds
data.
Console.WriteLine(factorsofFourList[2]);
factorsofFourList[2] = 0;
Console.WriteLine(factorsofFourList[2]);
```

Voir aussi

- LINQ (Language-Integrated Query)

Regrouper les résultats d'une requête

Article • 07/04/2023

Le regroupement est l'une des fonctionnalités les plus puissantes de LINQ. Les exemples suivants montrent comment regrouper des données de différentes manières :

- Selon une propriété
- Selon la première lettre d'une propriété de chaîne
- Selon une plage numérique calculée
- Selon un prédicat booléen ou une autre expression
- Selon une clé composée

En outre, les deux dernières requêtes projettent leurs résultats dans un nouveau type anonyme qui contient uniquement le prénom et le nom de l'étudiant. Pour plus d'informations, consultez group, clause.

> ⓘ **Notes**
>
> Les exemples de cette rubrique utilisent la `Student` classe et `students` la liste de l'exemple de code dans **Interroger une collection d'objets**.

Exemple de regroupement par propriété unique

L'exemple suivant montre comment regrouper des éléments source en utilisant une propriété de l'élément comme clé de groupe. Dans ce cas, la clé est un `string`, qui correspond au nom de l'étudiant. Il est également possible d'utiliser une sous-chaîne pour la clé ; consultez l'exemple suivant. L'opération de regroupement utilise le comparateur d'égalité par défaut pour le type.

C#

```
// Variable groupByLastNamesQuery is an IEnumerable<IGrouping<string,
// DataClass.Student>>.
var groupByLastNamesQuery =
    from student in students
    group student by student.LastName into newGroup
    orderby newGroup.Key
```

```
    select newGroup;

foreach (var nameGroup in groupByLastNamesQuery)
{
    Console.WriteLine($"Key: {nameGroup.Key}");
    foreach (var student in nameGroup)
    {
        Console.WriteLine($"\t{student.LastName}, {student.FirstName}");
    }
}

/* Output:
    Key: Adams
            Adams, Terry
    Key: Fakhouri
            Fakhouri, Fadi
    Key: Feng
            Feng, Hanying
    Key: Garcia
            Garcia, Cesar
            Garcia, Debra
            Garcia, Hugo
    Key: Mortensen
            Mortensen, Sven
    Key: O'Donnell
            O'Donnell, Claire
    Key: Omelchenko
            Omelchenko, Svetlana
    Key: Tucker
            Tucker, Lance
            Tucker, Michael
    Key: Zabokritski
            Zabokritski, Eugene
*/
```

Exemple de groupe par valeur

L'exemple suivant montre comment regrouper des éléments source en utilisant autre chose qu'une propriété de l'objet comme clé de groupe. Dans cet exemple, la clé est la première lettre du nom de l'étudiant.

C#

```
var groupByFirstLetterQuery =
    from student in students
    group student by student.LastName[0];

foreach (var studentGroup in groupByFirstLetterQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key}");
    // Nested foreach is required to access group items.
```

```
    foreach (var student in studentGroup)
    {
        Console.WriteLine($"\t{student.LastName}, {student.FirstName}");
    }
}

/* Output:
    Key: A
            Adams, Terry
    Key: F
            Fakhouri, Fadi
            Feng, Hanying
    Key: G
            Garcia, Cesar
            Garcia, Debra
            Garcia, Hugo
    Key: M
            Mortensen, Sven
    Key: O
            O'Donnell, Claire
            Omelchenko, Svetlana
    Key: T
            Tucker, Lance
            Tucker, Michael
    Key: Z
            Zabokritski, Eugene
*/
```

Exemple de regroupement par plage

L'exemple suivant montre comment regrouper des éléments source en utilisant une plage numérique comme clé de groupe. La requête projette ensuite les résultats dans un type anonyme qui contient uniquement le prénom et le nom de l'étudiant, ainsi que la plage de centiles à laquelle appartient l'étudiant. Un type anonyme est utilisé, car il n'est pas nécessaire d'utiliser l'intégralité de l'objet `Student` pour afficher les résultats. `GetPercentile` est une fonction d'assistance qui calcule un centile à partir de la note moyenne obtenue par l'étudiant. La méthode retourne un entier compris entre 0 et 10.

C#

```
int GetPercentile(Student s)
{
    double avg = s.ExamScores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}

var groupByPercentileQuery =
    from student in students
    let percentile = GetPercentile(student)
```

```
    group new
    {
        student.FirstName,
        student.LastName
    } by percentile into percentGroup
    orderby percentGroup.Key
    select percentGroup;

// Nested foreach required to iterate over groups and group items.
foreach (var studentGroup in groupByPercentileQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key * 10}");
    foreach (var item in studentGroup)
    {
        Console.WriteLine($"\t{item.LastName}, {item.FirstName}");
    }
}

/* Output:
    Key: 60
            Garcia, Debra
    Key: 70
            O'Donnell, Claire
    Key: 80
            Adams, Terry
            Feng, Hanying
            Garcia, Cesar
            Garcia, Hugo
            Mortensen, Sven
            Omelchenko, Svetlana
            Tucker, Lance
            Zabokritski, Eugene
    Key: 90
            Fakhouri, Fadi
            Tucker, Michael
*/
```

Exemple de regroupement par comparaison

L'exemple suivant montre comment regrouper des éléments source en utilisant une expression de comparaison booléenne. Dans cet exemple, l'expression booléenne teste si la note moyenne d'un étudiant est supérieure à 75. Comme dans les exemples précédents, les résultats sont projetés dans un type anonyme, car il n'est pas nécessaire d'avoir l'élément source complet. Notez que les propriétés du type anonyme deviennent des propriétés du membre `Key` et sont accessibles par nom lorsque la requête est exécutée.

```
C#
```

```
var groupByHighAverageQuery =
    from student in students
    group new
    {
        student.FirstName,
        student.LastName
    } by student.ExamScores.Average() > 75 into studentGroup
    select studentGroup;

foreach (var studentGroup in groupByHighAverageQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key}");
    foreach (var student in studentGroup)
    {
        Console.WriteLine($"\t{student.FirstName} {student.LastName}");
    }
}

/* Output:
    Key: True
            Terry Adams
            Fadi Fakhouri
            Hanying Feng
            Cesar Garcia
            Hugo Garcia
            Sven Mortensen
            Svetlana Omelchenko
            Lance Tucker
            Michael Tucker
            Eugene Zabokritski
    Key: False
            Debra Garcia
            Claire O'Donnell
*/
```

Regroupement par type anonyme

L'exemple suivant montre comment utiliser un type anonyme pour encapsuler une clé qui contient plusieurs valeurs. Dans cet exemple, la première valeur de clé est la première lettre du nom de l'étudiant. La deuxième valeur de clé est une valeur booléenne qui spécifie si l'étudiant a obtenu une note supérieure à 85 au premier examen. Vous pouvez organiser les groupes selon n'importe quelle propriété de la clé.

C#

```
var groupByCompoundKey =
    from student in students
    group student by new
    {
```

```
        FirstLetter = student.LastName[0],
        IsScoreOver85 = student.ExamScores[0] > 85
    } into studentGroup
    orderby studentGroup.Key.FirstLetter
    select studentGroup;

foreach (var scoreGroup in groupByCompoundKey)
{
    string s = scoreGroup.Key.IsScoreOver85 == true ? "more than 85" : "less
than 85";
    Console.WriteLine($"Name starts with {scoreGroup.Key.FirstLetter} who
scored {s}");
    foreach (var item in scoreGroup)
    {
        Console.WriteLine($"\t{item.FirstName} {item.LastName}");
    }
}

/* Output:
    Name starts with A who scored more than 85
            Terry Adams
    Name starts with F who scored more than 85
            Fadi Fakhouri
            Hanying Feng
    Name starts with G who scored more than 85
            Cesar Garcia
            Hugo Garcia
    Name starts with G who scored less than 85
            Debra Garcia
    Name starts with M who scored more than 85
            Sven Mortensen
    Name starts with O who scored less than 85
            Claire O'Donnell
    Name starts with O who scored more than 85
            Svetlana Omelchenko
    Name starts with T who scored less than 85
            Lance Tucker
    Name starts with T who scored more than 85
            Michael Tucker
    Name starts with Z who scored more than 85
            Eugene Zabokritski
*/
```

Voir aussi

- GroupBy
- IGrouping<TKey,TElement>
- LINQ (Language-Integrated Query)
- group, clause
- Types anonymes

- Effectuer une sous-requête sur une opération de regroupement
- Créer un groupe imbriqué
- Regroupement de données

Créer un groupe imbriqué

Article • 07/04/2023

L'exemple suivant montre comment créer des groupes imbriqués dans une expression de requête LINQ. Chaque groupe créé en fonction de l'année/du niveau d'étude est ensuite encore subdivisé en groupes en fonction des noms des individus.

Exemple

> **Notes**
>
> L'exemple de cette rubrique utilise la `Student` classe et `students` la liste de l'exemple de code dans **Interroger une collection d'objets**.

C#

```
var nestedGroupsQuery =
    from student in students
    group student by student.Year into newGroup1
    from newGroup2 in (
        from student in newGroup1
        group student by student.LastName
    )
    group newGroup2 by newGroup1.Key;

// Three nested foreach loops are required to iterate
// over all elements of a grouped group. Hover the mouse
// cursor over the iteration variables to see their actual type.
foreach (var outerGroup in nestedGroupsQuery)
{
    Console.WriteLine($"DataClass.Student Level = {outerGroup.Key}");
    foreach (var innerGroup in outerGroup)
    {
        Console.WriteLine($"\tNames that begin with: {innerGroup.Key}");
        foreach (var innerGroupElement in innerGroup)
        {
            Console.WriteLine($"\t\t{innerGroupElement.LastName}
{innerGroupElement.FirstName}");
        }
    }
}

/* Output:
    DataClass.Student Level = SecondYear
            Names that begin with: Adams
                    Adams Terry
            Names that begin with: Garcia
```

```
                    Garcia Hugo
            Names that begin with: Omelchenko
                    Omelchenko Svetlana
    DataClass.Student Level = ThirdYear
            Names that begin with: Fakhouri
                    Fakhouri Fadi
            Names that begin with: Garcia
                    Garcia Debra
            Names that begin with: Tucker
                    Tucker Lance
    DataClass.Student Level = FirstYear
            Names that begin with: Feng
                    Feng Hanying
            Names that begin with: Mortensen
                    Mortensen Sven
            Names that begin with: Tucker
                    Tucker Michael
    DataClass.Student Level = FourthYear
            Names that begin with: Garcia
                    Garcia Cesar
            Names that begin with: O'Donnell
                    O'Donnell Claire
            Names that begin with: Zabokritski
                    Zabokritski Eugene
*/
```

Notez que trois boucles `foreach` imbriquées sont nécessaires pour effectuer une itération sur les éléments internes d'un groupe imbriqué.

Voir aussi

- LINQ (Language-Integrated Query)

Effectuer une sous-requête sur une opération de regroupement

Article • 07/04/2023

Cet article présente deux façons de créer une requête qui organise les données sources en groupes et effectue ensuite une sous-requête sur chacun de ces groupes. Dans chaque exemple, la technique de base consiste à regrouper les éléments sources en utilisant une *continuation* nommée `newGroup`, puis en créant une sous-requête sur `newGroup`. Cette sous-requête est exécutée sur chaque groupe créé par la requête externe. Notez que, dans cet exemple particulier, le résultat final n'est pas un groupe, mais une séquence plate de types anonymes.

Pour plus d'informations sur les regroupements, consultez group, clause.

Pour plus d'informations sur les continuations, consultez into. L'exemple suivant utilise une structure de données en mémoire comme source de données, mais les mêmes principes s'appliquent à tous les types de sources de données LINQ.

Exemple

> ⓘ **Notes**
>
> Les exemples de cette rubrique utilisent la classe `Student` et la liste `students` de l'exemple de code dans **Interroger une collection d'objets**.

C#

```
var queryGroupMax =
    from student in students
    group student by student.Year into studentGroup
    select new
    {
        Level = studentGroup.Key,
        HighestScore = (
            from student2 in studentGroup
            select student2.ExamScores.Average()
        ).Max()
    };

int count = queryGroupMax.Count();
Console.WriteLine($"Number of groups = {count}");

foreach (var item in queryGroupMax)
```

```
{
    Console.WriteLine($"  {item.Level} Highest Score={item.HighestScore}");
}
```

La requête de l'extrait de code ci-dessus peut également s'écrire avec la syntaxe de méthode. L'extrait de code suivant comporte une requête sémantiquement équivalente, écrite avec la syntaxe de méthode.

C#

```
var queryGroupMax =
    students
        .GroupBy(student => student.Year)
        .Select(studentGroup => new
        {
            Level = studentGroup.Key,
            HighestScore = studentGroup.Select(student2 =>
student2.ExamScores.Average()).Max()
        });

int count = queryGroupMax.Count();
Console.WriteLine($"Number of groups = {count}");

foreach (var item in queryGroupMax)
{
    Console.WriteLine($"  {item.Level} Highest Score={item.HighestScore}");
}
```

Voir aussi

- LINQ (Language-Integrated Query)

Regrouper des résultats par clés contiguës

Article • 07/04/2023

L'exemple suivant montre comment regrouper des éléments dans des blocs représentant des sous-séquences de clés contiguës. Par exemple, supposons que vous disposiez de la séquence de paires clé-valeur suivante :

Clé	Valeur
Un	Nous
Un	pensons
A	que
B	Linq
C	is
Un	vraiment
B	chouette
B	!

Les groupes suivants seront créés dans cet ordre :

1. Nous, pensons, que
2. Linq
3. is
4. vraiment
5. chouette, !

La solution est implémentée comme une méthode d'extension thread-safe qui retourne ses résultats en continu. En d'autres termes, elle produit ses groupes à mesure qu'elle se déplace dans la séquence source. Contrairement aux opérateurs `group` ou `orderby`, elle peut commencer à retourner des groupes à l'appelant avant que l'intégralité de la séquence ait été lue.

La cohérence de thread est obtenue en effectuant une copie de chaque groupe ou bloc pendant l'itération de la séquence source, comme expliqué dans les commentaires du code source. Si la séquence source comporte une longue séquence d'éléments contigus, le common language runtime peut lever une OutOfMemoryException.

Exemple

L'exemple suivant montre la méthode d'extension et le code client qui l'utilise :

C#

```
public static class ChunkExtensions
{
    public static IEnumerable<IGrouping<TKey, TSource>> ChunkBy<TSource,
TKey>(
            this IEnumerable<TSource> source,
            Func<TSource, TKey> keySelector
        ) =>
            source.ChunkBy(keySelector, EqualityComparer<TKey>.Default);

    public static IEnumerable<IGrouping<TKey, TSource>> ChunkBy<TSource,
TKey>(
            this IEnumerable<TSource> source,
            Func<TSource, TKey> keySelector,
            IEqualityComparer<TKey> comparer
        )
    {
        // Flag to signal end of source sequence.
        const bool noMoreSourceElements = true;

        // Auto-generated iterator for the source array.
        var enumerator = source.GetEnumerator();

        // Move to the first element in the source sequence.
        if (!enumerator.MoveNext())
        {
            yield break;
        }

        // Iterate through source sequence and create a copy of each Chunk.
        // On each pass, the iterator advances to the first element of the
next "Chunk"
        // in the source sequence. This loop corresponds to the outer
foreach loop that
        // executes the query.
        Chunk<TKey, TSource> current = null;
        while (true)
        {
            // Get the key for the current Chunk. The source iterator will
churn through
            // the source sequence until it finds an element with a key that
doesn't match.
            var key = keySelector(enumerator.Current);

            // Make a new Chunk (group) object that initially has one
GroupItem, which is a copy of the current source element.
            current = new Chunk<TKey, TSource>(key, enumerator, value =>
comparer.Equals(key, keySelector(value)));
```

```
            // Return the Chunk. A Chunk is an IGrouping<TKey,TSource>,
which is the return value of the ChunkBy method.
            // At this point the Chunk only has the first element in its
source sequence. The remaining elements will be
            // returned only when the client code foreach's over this chunk.
See Chunk.GetEnumerator for more info.
            yield return current;

            // Check to see whether (a) the chunk has made a copy of all its
source elements or
            // (b) the iterator has reached the end of the source sequence.
If the caller uses an inner
            // foreach loop to iterate the chunk items, and that loop ran to
completion,
            // then the Chunk.GetEnumerator method will already have made
            // copies of all chunk items before we get here. If the
Chunk.GetEnumerator loop did not
            // enumerate all elements in the chunk, we need to do it here to
avoid corrupting the iterator
            // for clients that may be calling us on a separate thread.
            if (current.CopyAllChunkElements() == noMoreSourceElements)
            {
                yield break;
            }
        }

    }
}

// A Chunk is a contiguous group of one or more source elements that have
the same key. A Chunk
// has a key and a list of ChunkItem objects, which are copies of the
elements in the source sequence.
class Chunk<TKey, TSource> : IGrouping<TKey, TSource>
{
    // INVARIANT: DoneCopyingChunk == true ||
    //   (predicate != null && predicate(enumerator.Current) &&
current.Value == enumerator.Current)

    // A Chunk has a linked list of ChunkItems, which represent the elements
in the current chunk. Each ChunkItem
    // has a reference to the next ChunkItem in the list.
    class ChunkItem
    {
        public ChunkItem(TSource value)
        {
            Value = value;
        }
        public readonly TSource Value;
        public ChunkItem? Next = null;
    }

    // Stores a reference to the enumerator for the source sequence
    private IEnumerator<TSource> enumerator;
```

```
    // A reference to the predicate that is used to compare keys.
    private Func<TSource, bool> predicate;

    // Stores the contents of the first source element that
    // belongs with this chunk.
    private readonly ChunkItem head;

    // End of the list. It is repositioned each time a new
    // ChunkItem is added.
    private ChunkItem tail;

    // Flag to indicate the source iterator has reached the end of the
source sequence.
    internal bool isLastSourceElement = false;

    // Private object for thread syncronization
    private readonly object m_Lock;

    // REQUIRES: enumerator != null && predicate != null
    public Chunk(TKey key, IEnumerator<TSource> enumerator, Func<TSource,
bool> predicate)
    {
        Key = key;
        this.enumerator = enumerator;
        this.predicate = predicate;

        // A Chunk always contains at least one element.
        head = new ChunkItem(enumerator.Current);

        // The end and beginning are the same until the list contains > 1
elements.
        tail = head;

        m_Lock = new object();
    }

    // Indicates that all chunk elements have been copied to the list of
ChunkItems,
    // and the source enumerator is either at the end, or else on an element
with a new key.
    // the tail of the linked list is set to null in the
CopyNextChunkElement method if the
    // key of the next element does not match the current chunk's key, or
there are no more elements in the source.
    private bool DoneCopyingChunk => tail == null;

    // Adds one ChunkItem to the current group
    // REQUIRES: !DoneCopyingChunk && lock(this)
    private void CopyNextChunkElement()
    {
        // Try to advance the iterator on the source sequence.
        // If MoveNext returns false we are at the end, and
isLastSourceElement is set to true
        isLastSourceElement = !enumerator.MoveNext();
```

```
        // If we are (a) at the end of the source, or (b) at the end of the
current chunk
        // then null out the enumerator and predicate for reuse with the
next chunk.
        if (isLastSourceElement || !predicate(enumerator.Current))
        {
            enumerator = null;
            predicate = null;
        }
        else
        {
            tail.Next = new ChunkItem(enumerator.Current);
        }

        // tail will be null if we are at the end of the chunk elements
        // This check is made in DoneCopyingChunk.
        tail = tail.Next!;
    }

    // Called after the end of the last chunk was reached. It first checks
whether
    // there are more elements in the source sequence. If there are, it
    // Returns true if enumerator for this chunk was exhausted.
    internal bool CopyAllChunkElements()
    {
        while (true)
        {
            lock (m_Lock)
            {
                if (DoneCopyingChunk)
                {
                    // If isLastSourceElement is false,
                    // it signals to the outer iterator
                    // to continue iterating.
                    return isLastSourceElement;
                }
                else
                {
                    CopyNextChunkElement();
                }
            }
        }
    }

    public TKey Key { get; }

    // Invoked by the inner foreach loop. This method stays just one step
ahead
    // of the client requests. It adds the next element of the chunk only
after
    // the clients requests the last element in the list so far.
    public IEnumerator<TSource> GetEnumerator()
    {
        //Specify the initial element to enumerate.
```

```
        ChunkItem current = head;

        // There should always be at least one ChunkItem in a Chunk.
        while (current != null)
        {
            // Yield the current item in the list.
            yield return current.Value;

            // Copy the next item from the source sequence,
            // if we are at the end of our local list.
            lock (m_Lock)
            {
                if (current == tail)
                {
                    CopyNextChunkElement();
                }
            }

            // Move to the next ChunkItem in the list.
            current = current.Next;
        }
    }

    System.Collections.IEnumerator
System.Collections.IEnumerable.GetEnumerator() => GetEnumerator();
}

public static class GroupByContiguousKeys
{
    // The source sequence.
    static readonly KeyValuePair<string, string>[] list = {
        new("A","We"),
        new("A","think"),
        new("A","that"),
        new("B","LINQ"),
        new("C","is"),
        new ("A","really"),
        new("B","cool"),
        new("B","!")
    };

    // Query variable declared as class member to be available
    // on different threads.
    static readonly IEnumerable<IGrouping<string, KeyValuePair<string,
string>>> query =
        list.ChunkBy(p => p.Key);

    public static void GroupByContiguousKeys1()
    {
        // ChunkBy returns IGrouping objects, therefore a nested
        // foreach loop is required to access the elements in each "chunk".
        foreach (var item in query)
        {
            Console.WriteLine($"Group key = {item.Key}");
            foreach (var inner in item)
```

```
            {
                Console.WriteLine($"\t{inner.Value}");
            }
        }
    }
}
```

Voir aussi

- LINQ (Language-Integrated Query)

Spécifier dynamiquement des filtres de prédicat au moment de l'exécution

Article • 09/03/2023

Dans certains cas, ce n'est qu'au moment de l'exécution que vous savez combien de prédicats vous devez appliquer aux éléments sources dans la clause `where`. L'une des manières de spécifier plusieurs filtres de prédicat de manière dynamique consiste à utiliser la méthode Contains, comme indiqué dans l'exemple suivant. La requête retourne des résultats différents en fonction de la valeur de `id` lors de l'exécution de la requête.

C#

```
int[] ids = { 111, 114, 112 };

var queryNames =
    from student in students
    where ids.Contains(student.ID)
    select new
    {
        student.LastName,
        student.ID
    };

foreach (var name in queryNames)
{
    Console.WriteLine($"{name.LastName}: {name.ID}");
}

/* Output:
    Garcia: 114
    O'Donnell: 112
    Omelchenko: 111
 */

// Change the ids.
ids = new[] { 122, 117, 120, 115 };

// The query will now return different results
foreach (var name in queryNames)
{
    Console.WriteLine($"{name.LastName}: {name.ID}");
}

/* Output:
    Adams: 120
    Feng: 117
    Garcia: 115
```

```
    Tucker: 122
 */
```

Utilisation de différentes requêtes au moment de l'exécution

Vous pouvez utiliser des instructions de flux de contrôle, telles que `if... else` ou `switch`, pour effectuer une sélection parmi des requêtes alternatives prédéterminées. Dans l'exemple suivant, `studentQuery` utilise une clause `where` différente si la valeur d'exécution de `oddYear` est `true` ou `false`.

C#

```
void FilterByYearType(bool oddYear)
{
    IEnumerable<Student> studentQuery;
    if (oddYear)
    {
        studentQuery =
            from student in students
            where student.Year == GradeLevel.FirstYear || student.Year ==
GradeLevel.ThirdYear
            select student;
    }
    else
    {
        studentQuery =
            from student in students
            where student.Year == GradeLevel.SecondYear || student.Year ==
GradeLevel.FourthYear
            select student;
    }

    string descr = oddYear ? "odd" : "even";
    Console.WriteLine($"The following students are at an {descr} year
level:");
    foreach (Student name in studentQuery)
    {
        Console.WriteLine($"{name.LastName}: {name.ID}");
    }
}

FilterByYearType(true);

/* Output:
    The following students are at an odd year level:
    Fakhouri: 116
    Feng: 117
    Garcia: 115
```

```
    Mortensen: 113
    Tucker: 119
    Tucker: 122
 */

FilterByYearType(false);

/* Output:
    The following students are at an even year level:
    Adams: 120
    Garcia: 114
    Garcia: 118
    O'Donnell: 112
    Omelchenko: 111
    Zabokritski: 121
 */
```

Voir aussi

- LINQ (Language-Integrated Query)
- where, clause
- Requêtes en fonction de l'état du runtime

Effectuer des jointures internes

Article • 28/07/2023

Dans le domaine des bases de données relationnelles, une *jointure interne* produit un jeu de résultats dans lequel chaque élément de la première collection apparaît une fois pour chaque élément correspondant dans la deuxième collection. Si un élément de la première collection n'a pas d'élément correspondant, il n'apparaît pas dans le jeu de résultats. La méthode Join, qui est appelée par la clause `join` en C#, implémente une jointure interne.

Cet article explique comment effectuer quatre variations d'une jointure interne :

- Une jointure interne simple qui met en corrélation des éléments de deux sources de données sur la base d'une clé simple.
- Une jointure interne qui met en corrélation des éléments de deux sources de données sur la base d'une clé *composite*. Une clé composite, qui est une clé composée de plusieurs valeurs, permet de mettre en corrélation des éléments sur la base de plusieurs propriétés.
- Une *jointure multiple* dans laquelle les opérations de jointure consécutives sont ajoutées les unes aux autres.
- Une jointure interne qui est implémentée à l'aide d'une jointure groupée.

ⓘ Notes

Les exemples de cette rubrique utilisent les classes de données suivantes :

C#

```
record Person(string FirstName, string LastName);
record Pet(string Name, Person Owner);
record Employee(string FirstName, string LastName, int EmployeeID);
record Cat(string Name, Person Owner) : Pet(Name, Owner);
record Dog(string Name, Person Owner) : Pet(Name, Owner);
```

ainsi que la classe `Student` à partir de **Interroger une collection d'objets**.

Exemple – Jointure de clé simple

L'exemple suivant crée deux collections qui contiennent des objets de deux types définis par l'utilisateur, `Person` et `Pet`. La requête utilise la clause `join` dans C# pour faire correspondre les objets `Person` avec les objets `Pet` dont la propriété `Owner` est `Person`. La clause `select` dans C# définit l'apparence des objets résultants. Dans cet exemple, les objets résultants sont des types anonymes qui se composent du prénom du propriétaire et du nom de l'animal domestique.

C#

```
Person magnus = new(FirstName: "Magnus", LastName: "Hedlund");
Person terry = new("Terry", "Adams");
Person charlotte = new("Charlotte", "Weiss");
Person arlene = new("Arlene", "Huff");
Person rui = new("Rui", "Raposo");

List<Person> people = new() { magnus, terry, charlotte, arlene, rui };

List<Pet> pets = new()
{
    new(Name: "Barley", Owner: terry),
    new("Boots", terry),
    new("Whiskers", charlotte),
    new("Blue Moon", rui),
    new("Daisy", magnus),
};

// Create a collection of person-pet pairs. Each element in the collection
// is an anonymous type containing both the person's name and their pet's
name.
var query =
    from person in people
    join pet in pets on person equals pet.Owner
    select new
    {
        OwnerName = person.FirstName,
        PetName = pet.Name
    };

string result = "";
foreach (var ownerAndPet in query)
{
    result += $"\"{ownerAndPet.PetName}\" is owned by
{ownerAndPet.OwnerName}\r\n";
}
Console.Write(result);
return result;

/* Output:
     "Daisy" is owned by Magnus
     "Barley" is owned by Terry
     "Boots" is owned by Terry
     "Whiskers" is owned by Charlotte
```

```
     "Blue Moon" is owned by Rui
*/
```

Vous obtenez les mêmes résultats à l'aide de la syntaxe de méthode Join :

C#

```
var query =
    people.Join(pets,
                person => person,
                pet => pet.Owner,
                (person, pet) =>
                    new { OwnerName = person.FirstName, PetName = pet.Name
});
```

Notez que l'objet `Person` dont la propriété `LastName` est « Huff » n'apparaît pas dans le jeu de résultats, car il n'y a pas d'objet `Pet` ayant une propriété `Pet.Owner` égale à `Person`.

Exemple – Jointure de clé composite

Au lieu de mettre en corrélation des éléments sur la base d'une seule propriété, vous pouvez utiliser une clé composite pour comparer des éléments en fonction de plusieurs propriétés. Pour ce faire, spécifiez la fonction de sélection de clé de chaque collection pour retourner un type anonyme qui se compose des propriétés à comparer. Si vous étiquetez les propriétés, elles doivent avoir la même étiquette dans le type anonyme de chaque clé. Les propriétés doivent également apparaître dans le même ordre.

L'exemple suivant utilise une liste d'objets `Employee` et une liste d'objets `Student` pour déterminer quels employés sont également étudiants. Ces deux types ont des propriétés `FirstName` et `LastName` de type String. Les fonctions qui créent les clés de jointure à partir des éléments de chaque liste retournent un type anonyme qui se compose des propriétés `FirstName` et `LastName` de chaque élément. L'opération de jointure effectue une comparaison d'égalité de ces clés composites et retourne les paires d'objets de chaque liste où il y a correspondance entre le prénom et le nom.

C#

```
List<Employee> employees = new()
{
    new(FirstName: "Terry", LastName: "Adams", EmployeeID: 522459),
    new("Charlotte", "Weiss", 204467),
    new("Magnus", "Hedland", 866200),
    new("Vernette", "Price", 437139)
```

```
};

List<Student> students = new()
{
    new(FirstName: "Vernette", LastName: "Price", StudentID: 9562),
    new("Terry", "Earls", 9870),
    new("Terry", "Adams", 9913)
};

// Join the two data sources based on a composite key consisting of first
and last name,
// to determine which employees are also students.
var query =
    from employee in employees
    join student in students on new
    {
        employee.FirstName,
        employee.LastName
    } equals new
    {
        student.FirstName,
        student.LastName
    }
    select employee.FirstName + " " + employee.LastName;

string result = "";
result += "The following people are both employees and students:\r\n";
foreach (string name in query)
{
    result += $"{name}\r\n";
}
Console.Write(result);
return result;

/* Output:
    The following people are both employees and students:
    Terry Adams
    Vernette Price
 */
```

Vous pouvez utiliser la méthode Join, comme illustré dans l'exemple suivant :

C#

```
var query = employees.Join(
     students,
     employee => new {FirstName = employee.FirstName, LastName =
employee.LastName },
     student => new { FirstName = student.FirstName, student.LastName },
     (employee, student) => $"{employee.FirstName} {employee.LastName}"
 );
```

Exemple – Jointure multiple

Vous pouvez effectuer une jointure multiple en ajoutant n'importe quel nombre d'opérations de jointure les unes aux autres. Chaque clause `join` dans C# met en corrélation une source de données spécifiée avec les résultats de la jointure précédente.

L'exemple suivant crée trois collections : une liste d'objets `Person`, une liste d'objets `Cat` et une liste d'objets `Dog`.

La première clause `join` dans C# met en corrélation des personnes et des chats sur la base d'un objet `Person` correspondant à `Cat.Owner`. Elle retourne une séquence de types anonymes qui contiennent l'objet `Person` et la propriété `Cat.Name`.

La deuxième clause `join` dans C# met en corrélation les types anonymes retournés par la première jointure avec les objets `Dog` de la liste de chiens fournie, sur la base d'une clé composite constituée de la propriété `Owner` de type `Person` et de la première lettre du nom de l'animal. Elle retourne une séquence de types anonymes qui contiennent les propriétés `Cat.Name` et `Dog.Name` de chaque paire correspondante. Comme il s'agit d'une jointure interne, seuls les objets de la première source de données qui ont une correspondance dans la deuxième source de données sont retournés.

C#

```
Person magnus = new(FirstName: "Magnus", LastName: "Hedlund");
Person terry = new("Terry", "Adams");
Person charlotte = new("Charlotte", "Weiss");
Person arlene = new("Arlene", "Huff");
Person rui = new("Rui", "Raposo");
Person phyllis = new("Phyllis", "Harris");

List<Person> people = new() { magnus, terry, charlotte, arlene, rui, phyllis
};

List<Cat> cats = new()
{
    new(Name: "Barley", Owner: terry),
    new("Boots", terry),
    new("Whiskers", charlotte),
    new("Blue Moon", rui),
    new("Daisy", magnus),
};

List<Dog> dogs = new()
{
    new(Name: "Four Wheel Drive", Owner: phyllis),
    new("Duke", magnus),
    new("Denim", terry),
    new("Wiley", charlotte),
```

```
    new("Snoopy", rui),
    new("Snickers", arlene),
};

// The first join matches Person and Cat.Owner from the list of people and
// cats, based on a common Person. The second join matches dogs whose names
start
// with the same letter as the cats that have the same owner.
var query =
    from person in people
    join cat in cats on person equals cat.Owner
    join dog in dogs on new
    {
        Owner = person,
        Letter = cat.Name.Substring(0, 1)
    } equals new
    {
        dog.Owner,
        Letter = dog.Name.Substring(0, 1)
    }
    select new
    {
        CatName = cat.Name,
        DogName = dog.Name
    };

string result = "";
foreach (var obj in query)
{
    result += $"The cat \"{obj.CatName}\" shares a house, and the first
letter of their name, with \"{obj.DogName}\".\r\n";
}
Console.Write(result);
return result;

/* Output:
     The cat "Daisy" shares a house, and the first letter of their name,
with "Duke".
     The cat "Whiskers" shares a house, and the first letter of their name,
with "Wiley".
 */
```

L'équivalent utilisant plusieurs Join méthodes utilise la même approche avec le type anonyme (dans l'exemple ci-dessous, il est nommé `commonOwner`) :

C#

```
var query = people.Join(cats,
        person => person,
        cat => cat.Owner,
        (person, cat) => new { person, cat })
    .Join(dogs,
        commonOwner => new { Owner = commonOwner.person, Letter =
```

```
commonOwner.cat.Name.Substring(0, 1) },
        dog => new { dog.Owner, Letter = dog.Name.Substring(0, 1) },
        (commonOwner, dog) => new { CatName = commonOwner.cat.Name, DogName
= dog.Name });
```

Exemple – Jointure interne à l'aide d'une jointure groupée

L'exemple suivant montre comment implémenter une jointure interne en utilisant une jointure groupée.

Dans `query1`, la liste d'objets `Person` est jointe par groupe à la liste d'objets `Pet` sur la base de l'objet `Person` qui correspond à la propriété `Pet.Owner`. La jointure groupée crée une collection de groupes intermédiaires où chaque groupe se compose d'un objet `Person` et d'une séquence d'objets `Pet` correspondants.

En ajoutant une deuxième clause `from` à la requête, cette séquence de séquences est combinée (ou aplatie) dans une séquence plus longue. Le type des éléments de la séquence finale est spécifié par la clause `select`. Dans cet exemple, ce type est un type anonyme qui se compose des propriétés `Person.FirstName` et `Pet.Name` pour chaque paire correspondante.

Le résultat de `query1` est équivalent au jeu de résultats qui aurait été obtenu en utilisant la clause `join` sans la clause `into` pour effectuer une jointure interne. La variable `query2` illustre cette requête équivalente.

C#

```
Person magnus = new(FirstName: "Magnus", LastName: "Hedlund");
Person terry = new("Terry", "Adams");
Person charlotte = new("Charlotte", "Weiss");
Person arlene = new("Arlene", "Huff");

List<Person> people = new() { magnus, terry, charlotte, arlene };

List<Pet> pets = new()
{
    new(Name: "Barley", Owner: terry),
    new("Boots", terry),
    new("Whiskers", charlotte),
    new("Blue Moon", terry),
    new("Daisy", magnus),
};

var query1 =
    from person in people
```

```
    join pet in pets on person equals pet.Owner into gj
    from subpet in gj
    select new
    {
        OwnerName = person.FirstName,
        PetName = subpet.Name
    };

string result = "";
result += "Inner join using GroupJoin():\r\n";
foreach (var v in query1)
{
    result += $"{v.OwnerName} - {v.PetName}\r\n";
}

var query2 =
    from person in people
    join pet in pets on person equals pet.Owner
    select new
    {
        OwnerName = person.FirstName,
        PetName = pet.Name
    };

result += "\r\nThe equivalent operation using Join():\r\n";
foreach (var v in query2)
{
    result += $"{v.OwnerName} - {v.PetName}\r\n";
}
Console.Write(result);
return result;

/* Output:
    Inner join using GroupJoin():
    Magnus - Daisy
    Terry - Barley
    Terry - Boots
    Terry - Blue Moon
    Charlotte - Whiskers

    The equivalent operation using Join():
    Magnus - Daisy
    Terry - Barley
    Terry - Boots
    Terry - Blue Moon
    Charlotte - Whiskers
*/
```

Les mêmes résultats peuvent être obtenus à l'aide de GroupJoin la méthode suivante :

```
C#
```

```
var query1 = people.GroupJoin(pets,
        person => person,
        pet => pet.Owner,
        (person, gj) => new { person, gj })
    .SelectMany(pet => pet.gj,
        (groupJoinPet, subpet) => new { OwnerName =
groupJoinPet.person.FirstName, PetName = subpet.Name });
```

Notez que cette approche nécessite de chaîner les résultats de la requête avec SelectMany pour créer la liste finale de la relation Propriétaire -Animal en fonction des résultats de la jointure de groupe. Pour éviter le chaînage, la méthode unique Join peut être utilisée comme indiqué ci-dessous :

C#

```
var query2 = people.Join(pets,
    person => person,
    pet => pet.Owner,
    (person, pet) => new { OwnerName = person.FirstName, PetName = pet.Name
});
```

Voir aussi

- Join
- GroupJoin
- Effectuer des jointures groupées
- Effectuer des jointures externes gauches
- Types anonymes

Effectuer des jointures groupées

Article • 07/04/2023

La jointure groupée est utile pour produire des structures de données hiérarchiques. Elle associe chaque élément de la première collection à un jeu d'éléments corrélés de la deuxième collection.

Par exemple, une classe ou une table de base de données relationnelle nommée `Student` peut contenir deux champs : `Id` et `Name`. Une deuxième classe ou table de base de données relationnelle nommée `Course` peut contenir deux champs : `StudentId` et `CourseTitle`. Une jointure groupée de ces deux sources de données, basée sur les champs `Student.Id` et `Course.StudentId` correspondants, regroupe chaque `Student` avec une collection d'objets `Course` (qui peut être vide).

> ⓘ **Notes**
>
> Chaque élément de la première collection apparaît dans le jeu de résultats d'une jointure groupée, même si des éléments corrélés sont trouvés dans la deuxième collection. Si aucun élément corrélé n'est trouvé, la séquence d'éléments corrélés pour cet élément est vide. Le sélecteur de résultats a donc accès à chaque élément de la première collection. Cela n'est pas le cas du sélecteur de résultats dans une jointure non groupée, qui ne peut pas accéder à des éléments de la première collection qui n'ont aucune correspondance dans la deuxième collection.

> ⚠ **Avertissement**
>
> **Enumerable.GroupJoin** n'a pas d'équivalent direct dans les termes de base de données relationnelle traditionnels. Toutefois, cette méthode implémente un sur-ensemble de jointures internes et de jointures externes gauches. Ces deux opérations peuvent être écrites en termes de jointure groupée. Pour plus d'informations, consultez **Opérations de jointure** et **Entity Framework Core, GroupJoin**.

Le premier exemple de cet article montre comment effectuer une jointure groupée. Le deuxième exemple montre comment utiliser une jointure groupée pour créer des éléments XML.

> ⓘ **Notes**

Les exemples de cette rubrique utilisent les classes de données `Person` et `Pet` issues de **Effectuer des jointures internes**.

Exemple – Jointure groupée

L'exemple suivant effectue une jointure groupée d'objets de types `Person` et `Pet` où `Person` correspond à la propriété `Pet.Owner`. Contrairement à une jointure non groupée qui produirait une paire d'éléments pour chaque correspondance, la jointure groupée produit un seul objet résultant pour chaque élément de la première collection, qui est un objet `Person` dans cet exemple. Les éléments correspondants de la deuxième collection, qui sont des objets `Pet` dans cet exemple, sont regroupés dans une collection. Enfin, le sélecteur de résultats crée un type anonyme pour chaque correspondance qui se compose de `Person.FirstName` et d'une collection d'objets `Pet`.

C#

```
Person magnus = new(FirstName: "Magnus", LastName: "Hedlund");
Person terry = new("Terry", "Adams");
Person charlotte = new("Charlotte", "Weiss");
Person arlene = new("Arlene", "Huff");

List<Person> people = new() { magnus, terry, charlotte, arlene };

List<Pet> pets = new()
{
    new(Name: "Barley", Owner: terry),
    new("Boots", terry),
    new("Whiskers", charlotte),
    new("Blue Moon", terry),
    new("Daisy", magnus),
};

// Create a list where each element is an anonymous type
// that contains the person's first name and a collection of
// pets that are owned by them.
var query =
    from person in people
    join pet in pets on person equals pet.Owner into gj
    select new
    {
        OwnerName = person.FirstName,
        Pets = gj
    };

foreach (var v in query)
{
    // Output the owner's name.
    Console.WriteLine($"{v.OwnerName}:");
```

```
    // Output each of the owner's pet's names.
    foreach (var pet in v.Pets)
    {
        Console.WriteLine($"  {pet.Name}");
    }
}

/* Output:
     Magnus:
       Daisy
     Terry:
       Barley
       Boots
       Blue Moon
     Charlotte:
       Whiskers
     Arlene:
 */
```

Exemple – Jointure groupée pour créer des éléments XLM

Les jointures groupées sont appropriées pour créer des éléments XML à l'aide de LINQ to XML. L'exemple suivant est similaire à l'exemple précédent, sauf qu'au lieu de créer des types anonymes, le sélecteur de résultats crée des éléments XML qui représentent les objets joints.

C#

```
// using System.Xml.Linq;

Person magnus = new(FirstName: "Magnus", LastName: "Hedlund");
Person terry = new("Terry", "Adams");
Person charlotte = new("Charlotte", "Weiss");
Person arlene = new("Arlene", "Huff");

List<Person> people = new() { magnus, terry, charlotte, arlene };

List<Pet> pets = new()
{
    new(Name: "Barley", Owner: terry),
    new("Boots", terry),
    new("Whiskers", charlotte),
    new("Blue Moon", terry),
    new("Daisy", magnus),
};

XElement ownersAndPets = new("PetOwners",
    from person in people
```

```
    join pet in pets on person equals pet.Owner into gj
    select new XElement("Person",
        new XAttribute("FirstName", person.FirstName),
        new XAttribute("LastName", person.LastName),
        from subpet in gj
        select new XElement("Pet", subpet.Name)
    )
);

Console.WriteLine(ownersAndPets);

/* Output:
     <PetOwners>
       <Person FirstName="Magnus" LastName="Hedlund">
         <Pet>Daisy</Pet>
       </Person>
       <Person FirstName="Terry" LastName="Adams">
         <Pet>Barley</Pet>
         <Pet>Boots</Pet>
         <Pet>Blue Moon</Pet>
       </Person>
       <Person FirstName="Charlotte" LastName="Weiss">
         <Pet>Whiskers</Pet>
       </Person>
       <Person FirstName="Arlene" LastName="Huff" />
     </PetOwners>
*/
```

Voir aussi

- Join
- GroupJoin
- Effectuer des jointures internes
- Effectuer des jointures externes gauches
- Types anonymes

Effectuer des jointures externes gauches

Article • 07/04/2023

Une jointure externe gauche est une jointure dans laquelle chaque élément de la première collection est retourné, qu'elle ait ou non des éléments corrélés dans la deuxième collection. Vous pouvez utiliser LINQ pour effectuer une jointure externe gauche en appelant la méthode DefaultIfEmpty sur les résultats d'une jointure groupée.

> ⓘ Notes
>
> L'exemple de cette rubrique utilise les classes de données `Pet` et `Person` issues de **Effectuer des jointures internes**.

Exemple

L'exemple suivant montre comment utiliser la méthode DefaultIfEmpty sur les résultats d'une jointure groupée pour effectuer une jointure externe gauche.

Pour créer une jointure externe gauche entre deux collections, la première étape consiste à effectuer une jointure interne à l'aide d'une jointure groupée. (Pour une explication de ce processus, consultez Effectuer des jointures internes.) Dans cet exemple, la liste des objets `Person` est jointe en interne à la liste des objets `Pet` en fonction d'un objet `Person` qui correspond à `Pet.Owner`.

La deuxième étape consiste à inclure tous les éléments de la première collection (celle de gauche) dans le jeu de résultats, y compris les éléments sans correspondance dans la collection de droite. Pour cela, appelez DefaultIfEmpty sur chaque séquence d'éléments correspondants de la jointure groupée. Dans cet exemple, la méthode DefaultIfEmpty est appelée sur chaque séquence d'objets `Pet` correspondants. La méthode retourne une collection qui contient uniquement une valeur par défaut si la séquence des objets `Pet` correspondants à un objet `Person` est vide, ce qui garantit que chaque objet `Person` est représenté dans la collection de résultats.

> ⓘ Notes
>
> La valeur par défaut pour un type référence est `null`. L'exemple recherche donc une référence null avant d'accéder à chaque élément de chaque collection `Pet`.

C#

```
Person magnus = new("Magnus", "Hedlund");
Person terry = new("Terry", "Adams");
Person charlotte = new("Charlotte", "Weiss");
Person arlene = new("Arlene", "Huff");

Pet barley = new("Barley", terry);
Pet boots = new("Boots", terry);
Pet whiskers = new("Whiskers", charlotte);
Pet bluemoon = new("Blue Moon", terry);
Pet daisy = new("Daisy", magnus);

// Create two lists.
List<Person> people = new() { magnus, terry, charlotte, arlene };
List<Pet> pets = new() { barley, boots, whiskers, bluemoon, daisy };

var query =
    from person in people
    join pet in pets on person equals pet.Owner into gj
    from subpet in gj.DefaultIfEmpty()
    select new
    {
        person.FirstName,
        PetName = subpet?.Name ?? string.Empty
    };

foreach (var v in query)
{
    Console.WriteLine($"{v.FirstName + ":",-15}{v.PetName}");
}

record class Person(string FirstName, string LastName);
record class Pet(string Name, Person Owner);

// This code produces the following output:
//
// Magnus:        Daisy
// Terry:         Barley
// Terry:         Boots
// Terry:         Blue Moon
// Charlotte:     Whiskers
// Arlene:
```

Voir aussi

- Join
- GroupJoin
- Effectuer des jointures internes
- Effectuer des jointures groupées

- Types anonymes

Classer les résultats d'une clause join

Article • 07/04/2023

Cet exemple montre comment classer les résultats d'une opération de jointure. Notez que le classement est effectué après la jointure. Vous pouvez utiliser une clause `orderby` avec une ou plusieurs séquences sources avant la jointure, mais cette pratique est généralement déconseillée. Certains fournisseurs LINQ ne conservent pas ce classement après la jointure.

> ⓘ **Notes**
>
> L'exemple de cette rubrique utilise les classes de données suivantes :

C#

```
record Product(string Name, int CategoryID);
record Category(string Name, int ID);
```

Exemple

Cette requête crée une jointure groupée, puis trie les groupes en fonction de l'élément de catégorie, qui est encore dans la portée. À l'intérieur de l'initialiseur de type anonyme, une sous-requête classe tous les éléments correspondants de la séquence de produits.

C#

```
List<Category> categories = new()
{
    new(Name: "Beverages", ID: 001),
    new("Condiments", 002),
    new("Vegetables", 003),
    new("Grains", 004),
    new("Fruit", 005)
};

List<Product> products = new()
{
    new(Name: "Cola", CategoryID: 001),
    new("Tea", 001),
    new("Mustard", 002),
    new("Pickles", 002),
    new("Carrots", 003),
    new("Bok Choy", 003),
```

```
    new("Peaches", 005),
    new("Melons", 005),
};

var groupJoinQuery2 =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID into
prodGroup
    orderby category.Name
    select new
    {
        Category = category.Name,
        Products =
            from prod2 in prodGroup
            orderby prod2.Name
            select prod2
    };

foreach (var productGroup in groupJoinQuery2)
{
    Console.WriteLine(productGroup.Category);
    foreach (var prodItem in productGroup.Products)
    {
        Console.WriteLine($"  {prodItem.Name,-10} {prodItem.CategoryID}");
    }
}

/* Output:
    Beverages
      Cola       1
      Tea        1
    Condiments
      Mustard    2
      Pickles    2
    Fruit
      Melons     5
      Peaches    5
    Grains
    Vegetables
      Bok Choy   3
      Carrots    3
 */
```

Voir aussi

- LINQ (Language-Integrated Query)
- orderby, clause
- join, clause

Effectuer des jointures à l'aide de clés composites

Article • 07/04/2023

Cet exemple montre comment effectuer des opérations de jointure où vous voulez utiliser plusieurs clés pour définir une correspondance. Ceci s'effectue à l'aide d'une clé composite. Vous créez une clé composite en tant que type anonyme ou typé nommé avec les valeurs que vous voulez comparer. Si la variable de requête doit être passée au-delà des limites de la méthode, utilisez un type nommé qui substitue Equals et GetHashCode pour la clé. Les noms des propriétés et l'ordre dans lequel elles se trouvent doivent être identiques dans chaque clé.

Exemple

L'exemple suivant montre comment utiliser une clé composite pour joindre les données de trois tables :

C#

```
var query = from o in db.Orders
    from p in db.Products
    join d in db.OrderDetails
        on new {o.OrderID, p.ProductID} equals new {d.OrderID, d.ProductID}
into details
        from d in details
        select new {o.OrderID, p.ProductID, d.UnitPrice};
```

L'inférence du type sur les clés composites dépend des noms des propriétés dans les clés et l'ordre dans lequel elles apparaissent. Si les propriétés dans les séquences sources n'ont pas les mêmes noms, vous devez affecter de nouveaux noms dans les clés. Par exemple, si la table `Orders` et la table `OrderDetails` utilisent chacune des noms différents pour leurs colonnes, vous pouvez créer des clés composites en affectant des noms identiques dans les types anonymes :

C#

```
join...on new {Name = o.CustomerName, ID = o.CustID} equals
    new {Name = d.CustName, ID = d.CustID }
```

Vous pouvez aussi utiliser des clés composites dans une clause `group`.

Voir aussi

- LINQ (Language-Integrated Query)
- join, clause
- group, clause

Effectuer des opérations de jointure personnalisées

Article • 07/04/2023

Cet exemple montre comment effectuer des opérations de jointure qui ne sont pas possibles avec la clause `join`. Dans une expression de requête, la clause `join` est limitée à et optimisée pour les équijointures, qui sont de loin le type le plus courant d'opération de jointure. Quand vous effectuez une équijointure, l'utilisation de la clause `join` offre généralement les meilleures performances.

Toutefois, la clause `join` ne peut pas être utilisée dans les cas suivants :

- Quand la jointure est basée sur une expression de prédicat d'inégalité (une non-équijointure).
- Quand la jointure est basée sur plusieurs expressions de prédicat d'égalité ou d'inégalité.
- Quand vous devez introduire une variable de portée temporaire pour la séquence de droite (interne) avant l'opération de jointure.

Pour effectuer des jointures qui ne sont pas des équijointures, vous pouvez utiliser plusieurs clauses `from` pour introduire chaque source de données indépendamment. Vous appliquez ensuite une expression de prédicat dans une clause `where` à la variable de portée pour chaque source. L'expression peut également prendre la forme d'un appel de méthode.

> ⓘ **Notes**
>
> Ne confondez pas ce type d'opération de jointure personnalisée avec l'utilisation de plusieurs clauses `from` permettant d'accéder aux collections internes. Pour plus d'informations, consultez **join, clause**.

Jointure croisée

> ⓘ **Notes**
>
> Cet exemple et celui-ci après avoir utilisé les définitions `Product` et `Category` de **Trier les résultats d'une clause de jointure**.

Cette requête montre une jointure croisée simple. Les jointures croisées doivent être utilisées avec précaution, car elles peuvent produire des jeux de résultats très volumineux. Toutefois, elles peuvent être utiles dans certains scénarios pour créer des séquences sources sur lesquelles exécuter des requêtes supplémentaires.

C#

```
List<Category> categories = new()
{
    new(Name: "Beverages", ID: 001),
    new("Condiments", 002),
    new("Vegetables", 003)
};

List<Product> products = new()
{
    new(Name: "Tea", CategoryID: 001),
    new("Mustard", 002),
    new("Pickles", 002),
    new("Carrots", 003),
    new("Bok Choy", 003),
    new("Peaches", 005),
    new("Melons", 005),
    new("Ice Cream", 007),
    new("Mackerel", 012)
};

var crossJoinQuery =
    from c in categories
    from p in products
    select new
    {
        c.ID,
        p.Name
    };

Console.WriteLine("Cross Join Query:");
foreach (var v in crossJoinQuery)
{
    Console.WriteLine($"{v.ID,-5}{v.Name}");
}
/* Output:
    Cross Join Query:
    1    Tea
    1    Mustard
    1    Pickles
    1    Carrots
    1    Bok Choy
    1    Peaches
    1    Melons
    1    Ice Cream
    1    Mackerel
```

```
    2    Tea
    2    Mustard
    2    Pickles
    2    Carrots
    2    Bok Choy
    2    Peaches
    2    Melons
    2    Ice Cream
    2    Mackerel
    3    Tea
    3    Mustard
    3    Pickles
    3    Carrots
    3    Bok Choy
    3    Peaches
    3    Melons
    3    Ice Cream
    3    Mackerel
*/
```

Non-equijoin

Cette requête produit une séquence de tous les produits dont l'ID de catégorie est répertorié dans la liste des catégories à gauche. Notez l'utilisation de la clause `let` et de la méthode `Contains` pour créer un tableau temporaire. Il est également possible de créer le tableau avant la requête et de supprimer la première clause `from`.

C#

```
var nonEquijoinQuery =
    from p in products
    let catIds =
        from c in categories
        select c.ID
    where catIds.Contains(p.CategoryID) == true
    select new
    {
        Product = p.Name,
        p.CategoryID
    };

Console.WriteLine("Non-equijoin query:");
foreach (var v in nonEquijoinQuery)
{
    Console.WriteLine($"{v.CategoryID,-5}{v.Product}");
}

/* Output:
    Non-equijoin query:
    1    Tea
```

```
    2    Mustard
    2    Pickles
    3    Carrots
    3    Bok Choy
*/
```

Fusionner des fichiers CSV

Dans l'exemple suivant, la requête doit joindre deux séquences sur la base des clés correspondantes qui, pour la séquence interne (à droite), ne peuvent pas être obtenues avant la clause de jointure elle-même. Si cette jointure a été effectuée avec une clause `join`, la méthode `Split` doit être appelée pour chaque élément. L'utilisation de plusieurs clauses `from` permet d'éviter à la requête la surcharge inhérente à la répétition de l'appel de méthode. Toutefois, comme la clause `join` est optimisée, dans ce cas particulier, son utilisation peut s'avérer plus rapide que d'utiliser plusieurs clauses `from`. Les résultats dépendront principalement de la surcharge liée à l'appel de la méthode.

C#

```
string[] names = File.ReadAllLines(@"csv/names.csv");
string[] scores = File.ReadAllLines(@"csv/scores.csv");

// Merge the data sources using a named type.
// You could use var instead of an explicit type for the query.
IEnumerable<Student> queryNamesScores =
    // Split each line in the data files into an array of strings.
    from name in names
    let x = name.Split(',')
    from score in scores
    let s = score.Split(',')
    // Look for matching IDs from the two data files.
    where x[2] == s[0]
    // If the IDs match, build a Student object.
    select new Student(
        FirstName: x[0],
        LastName: x[1],
        StudentID: int.Parse(x[2]),
        ExamScores: (
            from scoreAsText in s.Skip(1)
            select int.Parse(scoreAsText)
        ).ToList()
    );

// Optional. Store the newly created student objects in memory
// for faster access in future queries
List<Student> students = queryNamesScores.ToList();

foreach (var student in students)
{
```

```
    Console.WriteLine($"The average score of {student.FirstName}
{student.LastName} is {student.ExamScores.Average()}.");
}

/* Output:
    The average score of Omelchenko Svetlana is 82.5.
    The average score of O'Donnell Claire is 72.25.
    The average score of Mortensen Sven is 84.5.
    The average score of Garcia Cesar is 88.25.
    The average score of Garcia Debra is 67.
    The average score of Fakhouri Fadi is 92.25.
    The average score of Feng Hanying is 88.
    The average score of Garcia Hugo is 85.75.
    The average score of Tucker Lance is 81.75.
    The average score of Adams Terry is 85.25.
    The average score of Zabokritski Eugene is 83.
    The average score of Tucker Michael is 92.
 */
```

Voir aussi

- LINQ (Language-Integrated Query)
- join, clause
- Classer les résultats d'une clause join

Gérer des valeurs Null dans des expressions de requête

Article • 07/04/2023

Cet exemple montre comment gérer d'éventuelles valeurs Null dans des collections sources. Une collection d'objets telle qu'un IEnumerable<T> peut contenir des éléments dont la valeur est null. Si une collection source est `null` ou contient un élément dont la valeur est `null` et que votre requête ne gère pas les valeurs `null`, une NullReferenceException est levée quand vous exécutez la requête.

Vous pouvez écrire du code en prévention pour éviter une exception de référence Null, comme indiqué dans l'exemple suivant :

C#

```
var query1 =
    from c in categories
    where c != null
    join p in products on c.ID equals p?.CategoryID
    select new
    {
        Category = c.Name,
        Name = p.Name
    };
```

Dans l'exemple précédent, la clause `where` exclut tous les éléments Null de la séquence de catégories. Cette technique est indépendante de la vérification de valeur Null de la clause join. Dans cet exemple, l'expression conditionnelle avec une valeur Null fonctionne, car `Products.CategoryID` est de type `int?`, qui est le raccourci de `Nullable<int>`.

Dans une clause join, si seulement l'une des clés de comparaison est un type valeur Nullable, vous pouvez caster l'autre clé en type valeur Nullable dans l'expression de requête. Dans l'exemple suivant, supposons que `EmployeeID` soit une colonne qui contienne des valeurs de type `int?` :

C#

```
void TestMethod(Northwind db)
{
    var query =
        from o in db.Orders
        join e in db.Employees
```

```
            on o.EmployeeID equals (int?)e.EmployeeID
        select new { o.OrderID, e.FirstName };
}
```

Dans chacun des exemples, le mot clé de requête `equals` est utilisé. C# 9 ajoute des critères spéciaux, qui inclut des modèles pour `is null` et `is not null`. Ces modèles ne sont pas recommandés dans les requêtes LINQ, car les fournisseurs de requêtes peuvent ne pas interpréter correctement la nouvelle syntaxe C#. Un fournisseur de requêtes est une bibliothèque qui traduit des expressions de requête C# dans un format de données natif, tel qu'Entity Framework Core. Les fournisseurs de requêtes implémentent l'interface System.Linq.IQueryProvider pour créer des sources de données qui implémentent l'interface System.Linq.IQueryable<T>.

Voir aussi

- Nullable<T>
- LINQ (Language-Integrated Query)
- Types valeur Nullable

Gérer des exceptions dans des expressions de requête

Article • 10/05/2023

Dans le contexte d'une expression de requête, vous pouvez appeler n'importe quelle méthode. Toutefois, nous vous recommandons d'éviter d'appeler une méthode dans une expression de requête susceptible de créer un effet secondaire, tel que la modification du contenu de la source de données ou la levée d'une exception. Cet exemple montre comment éviter la levée d'exceptions lorsque vous appelez des méthodes dans une expression de requête, en respectant les directives générales .NET relatives à la gestion des exceptions. Selon ces directives, il est acceptable d'intercepter une exception spécifique si vous comprenez pourquoi elle est levée dans un contexte donné. Pour plus d'informations, consultez les Bonnes pratiques pour les exceptions.

Le dernier exemple montre comment gérer les cas où vous devez lever une exception pendant l'exécution d'une requête.

Exemple 1

L'exemple suivant montre comment déplacer du code de gestion des exceptions en dehors d'une expression de requête. Ceci est possible uniquement lorsque la méthode ne dépend pas des variables locales de la requête.

C#

```
// A data source that is very likely to throw an exception!
IEnumerable<int> GetData() => throw new InvalidOperationException();

// DO THIS with a datasource that might
// throw an exception. It is easier to deal with
// outside of the query expression.
IEnumerable<int>? dataSource = null;
try
{
    dataSource = GetData();
}
catch (InvalidOperationException)
{
    // Handle (or don't handle) the exception
    // in the way that is appropriate for your application.
    Console.WriteLine("Invalid operation");
}

if (dataSource is not null)
```

```
{
    // If we get here, it is safe to proceed.
    var query =
        from i in dataSource
        select i * i;

    foreach (var i in query)
    {
        Console.WriteLine(i.ToString());
    }
}
```

Exemple 2

Dans certains cas, la meilleure réponse à la levée d'une exception à l'intérieur d'une requête consiste à arrêter immédiatement l'exécution de la requête. L'exemple suivant montre comment gérer les exceptions pouvant être levées dans le corps d'une requête. Supposons que `SomeMethodThatMightThrow` puisse provoquer la levée d'une exception qui nécessite l'arrêt de l'exécution de la requête.

Notez que le bloc `try` englobe la boucle `foreach` et non la requête. Ceci s'explique par le fait que c'est au niveau de la boucle `foreach` que la requête est exécutée. Pour plus d'informations, consultez Introduction aux requêtes LINQ.

C#

```
// Not very useful as a general purpose method.
string SomeMethodThatMightThrow(string s) =>
    s[4] == 'C' ?
        throw new InvalidOperationException() :
        @"C:\newFolder\" + s;

// Data source.
string[] files = { "fileA.txt", "fileB.txt", "fileC.txt" };

// Demonstration query that throws.
var exceptionDemoQuery =
    from file in files
    let n = SomeMethodThatMightThrow(file)
    select n;

// The runtime exception will only be thrown when the query is executed.
// Therefore they must be handled in the foreach loop.
try
{
    foreach (var item in exceptionDemoQuery)
    {
        Console.WriteLine($"Processing {item}");
    }
```

```
}

// Catch whatever exception you expect to raise
// and/or do any necessary cleanup in a finally block
catch (InvalidOperationException e)
{
    Console.WriteLine(e.Message);
}

/* Output:
    Processing C:\newFolder\fileA.txt
    Processing C:\newFolder\fileB.txt
    Operation is not valid due to the current state of the object.
 */
```

Voir aussi

- LINQ (Language-Integrated Query)

Reduce memory allocations using new C# features

Article • 04/06/2023

> **ⓘ Important**
>
> The techniques described in this section improve performance when applied to *hot paths* in your code. *Hot paths* are those sections of your codebase that are executed often and repeatedly in normal operations. Applying these techniques to code that isn't often executed will have minimal impact. Before making any changes to improve performance, it's critical to measure a baseline. Then, analyze that baseline to determine where memory bottlenecks occur. You can learn about many cross platform tools to measure your application's performance in the section on **Diagnostics and instrumentation**. You can practice a profiling session in the tutorial to **Measure memory usage** in the Visual Studio documentation.
>
> Once you've measured memory usage and have determined that you can reduce allocations, use the techniques in this section to reduce allocations. After each successive change, measure memory usage again. Make sure each change has a positive impact on the memory usage in your application.

Performance work in .NET often means removing allocations from your code. Every block of memory you allocate must eventually be freed. Fewer allocations reduce time spent in garbage collection. It allows for more predictable execution time by removing garbage collections from specific code paths.

A common tactic to reduce allocations is to change critical data structures from `class` types to `struct` types. This change impacts the semantics of using those types. Parameters and returns are now passed by value instead of by reference. The cost of copying a value is negligible if the types are small, three words or less (considering one word being of natural size of one integer). It's measurable and can have real performance impact for larger types. To combat the effect of copying, developers can pass these types by `ref` to get back the intended semantics.

The C# `ref` features give you the ability to express the desired semantics for `struct` types without negatively impacting their overall usability. Prior to these enhancements, developers needed to resort to `unsafe` constructs with pointers and raw memory to achieve the same performance impact. The compiler generates *verifiably safe code* for the new `ref` related features. *Verifiably safe code* means the compiler detects possible

buffer overruns or accessing unallocated or freed memory. The compiler detects and prevents some errors.

Pass and return by reference

Variables in C# store *values*. In `struct` types, the value is the contents of an instance of the type. In `class` types, the value is a reference to a block of memory that stores an instance of the type. Adding the `ref` modifier means that the variable stores the *reference* to the value. In `struct` types, the reference points to the storage containing the value. In `class` types, the reference points to the storage containing the reference to the block of memory.

In C#, parameters to methods are *passed by value*, and return values are *return by value*. The *value* of the argument is passed to the method. The *value* of the return argument is the return value.

The `ref`, `in`, or `out` modifier indicates that parameter is *passed by reference*. The *reference* to the storage location is passed to the method. Adding `ref` to the method signature means the return value is *returned by reference*. The *reference* to the storage location is the return value.

You can also use *ref assignment* to have a variable refer to another variable. A typical assignment copies the *value* of the right hand side to the variable on the left hand side of the assignment. A *ref assignment* copies the memory location of the variable on the right hand side to the variable on the left hand side. The `ref` now refers to the original variable:

C#

```
int anInteger = 42; // assignment.
ref int location = ref anInteger; // ref assignment.
ref int sameLocation = ref location; // ref assignment

Console.WriteLine(location); // output: 42

sameLocation = 19; // assignment

Console.WriteLine(anInteger); // output: 19
```

When you *assign* a variable, you change its *value*. When you *ref assign* a variable, you change what it refers to.

You can work directly with the storage for values using `ref` variables, pass by reference, and ref assignment. Scope rules enforced by the compiler ensure safety when working directly with storage.

Ref safe to escape scope

C# includes rules for `ref` expressions to ensure that a `ref` expression can't be accessed where the storage it refers to is no longer valid. Consider the following example:

```
C#
```

```
public ref int CantEscape()
{
    int index = 42;
    return ref index; // Error: index's ref safe to escape scope is the body
of CantEscape
}
```

The compiler reports an error because you can't return a reference to a local variable from a method. The caller can't access the storage being referred to. The *ref safe to escape scope* defines the scope in which a `ref` expression is safe to access or modify. The following table lists the *ref safe to escape scopes* for variable types. `ref` fields can't be declared in a `class` or a non-ref `struct`, so those rows aren't in the table:

Declaration	***ref safe to escape scope***
non-ref local	block where local is declared
non-ref parameter	current method
`ref`, `in` parameter	calling method
`out` parameter	current method
`class` field	calling method
non-ref `struct` field	current method
`ref` field of `ref struct`	calling method

A variable can be `ref` returned if its *ref safe to escape scope* is the calling method. If its *ref safe to escape scope* is the current method or a block, `ref` return is disallowed. The following snippet shows two examples. A member field can be accessed from the scope calling a method, so a class or struct field's *ref safe to escape scope* is the calling method.

The *ref safe to escape scope* for a parameter with the `ref`, or `in` modifiers is the entire method. Both can be `ref` returned from a member method:

```
C#
```

```
private int anIndex;

public ref int RetrieveIndexRef()
{
    return ref anIndex;
}

public ref int RefMin(ref int left, ref int right)
{
    if (left < right)
        return ref left;
    else
        return ref right;
}
```

> **Note**
>
> When the `in` modifier is applied to a parameter, that parameter can be returned by `ref readonly`, not `ref`.

The compiler ensures that a reference can't escape its *ref safe to escape scope*. You can use `ref` parameters, `ref return` and `ref` local variables safely because the compiler detects if you've accidentally written code where a `ref` expression could be accessed when its storage isn't valid.

Safe to escape scope and ref structs

`ref struct` types require more rules to ensure they can be used safely. A `ref struct` type may include `ref` fields. That requires the introduction of a *safe to escape scope*. For most types, the *safe to escape scope* is the calling method. In other words, a value that's not a `ref struct` can always be returned from a method.

Informally, the *safe to escape scope* for a `ref struct` is the scope where all of its `ref` fields can be accessed. In other words, it's the intersection of the *ref safe to escape scopes* of all its `ref` fields. The following method returns a `ReadOnlySpan<char>` to a member field, so its *safe to escape scope* is the method:

```
C#
```

```
private string longMessage = "This is a long message";

public ReadOnlySpan<char> Safe()
{
    var span = longMessage.AsSpan();
    return span;
}
```

In contrast, the following code emits an error because the `ref field` member of the `Span<int>` refers to the stack allocated array of integers. It can't escape the method:

C#

```
public Span<int> M()
{
    int length = 3;
    Span<int> numbers = stackalloc int[length];
    for (var i = 0; i < length; i++)
    {
        numbers[i] = i;
    }
    return numbers; // Error! numbers can't escape this method.
}
```

Unify memory types

The introduction of System.Span<T> and System.Memory<T> provide a unified model for working with memory. System.ReadOnlySpan<T> and System.ReadOnlyMemory<T> provide readonly versions for accessing memory. They all provide an abstraction over a block of memory storing an array of similar elements. The difference is that `Span<T>` and `ReadOnlySpan<T>` are `ref struct` types whereas `Memory<T>` and `ReadOnlyMemory<T>` are `struct` types. Spans contain a `ref field`. Therefore instances of a span can't leave its *safe to escape scope*. The *safe to escape* scope of a `ref struct` is the *ref safe to escape scope* of its `ref field`. The implementation of `Memory<T>` and `ReadOnlyMemory<T>` remove this restriction. You use these types to directly access memory buffers.

Improve performance with ref safety

Using these features to improve performance involves these tasks:

- *Avoid allocations*: When you change a type from a `class` to a `struct`, you change how it's stored. Local variables are stored on the stack. Members are stored inline when the container object is allocated. This change means fewer allocations and

that decreases the work the garbage collector does. It may also decrease memory pressure so the garbage collector runs less often.

- *Preserve reference semantics*: Changing a type from a `class` to a `struct` changes the semantics of passing a variable to a method. Code that modified the state of its parameters needs modification. Now that the parameter is a `struct`, the method is modifying a copy of the original object. You can restore the original semantics by passing that parameter as a `ref` parameter. After that change, the method modifies the original `struct` again.
- *Avoid copying data*: Copying larger `struct` types can impact performance in some code paths. You can also add the `ref` modifier to pass larger data structures to methods by reference instead of by value.
- *Restrict modifications*: When a `struct` type is passed by reference, the called method could modify the state of the struct. You can replace the `ref` modifier with the `in` modifier to indicate that the argument can't be modified. You can also create `readonly struct` types or `struct` types with `readonly` members to provide more control over what members of a `struct` can be modified.
- *Directly manipulate memory*: Some algorithms are most efficient when treating data structures as a block of memory containing a sequence of elements. The `Span` and `Memory` types provide safe access to blocks of memory.

None of these techniques require `unsafe` code. Used wisely, you can get performance characteristics from safe code that was previously only possible by using unsafe techniques. You can try the techniques yourself in the tutorial on reducing memory allocations.

Expression Trees

Article • 03/09/2023

Expression trees represent code in a tree-like data structure, where each node is an expression, for example, a method call or a binary operation such as `x < y`.

If you have used LINQ, you have experience with a rich library where the `Func` types are part of the API set. (If you aren't familiar with LINQ, you probably want to read the LINQ tutorial and the article about lambda expressions before this one.) Expression Trees provide richer interaction with the arguments that are functions.

You write function arguments, typically using Lambda Expressions, when you create LINQ queries. In a typical LINQ query, those function arguments are transformed into a delegate the compiler creates.

You've likely already written code that uses Expression trees. Entity Framework's LINQ APIs accept Expression trees as the arguments for the LINQ Query Expression Pattern. That enables Entity Framework to translate the query you wrote in C# into SQL that executes in the database engine. Another example is Moq, which is a popular mocking framework for .NET.

When you want to have a richer interaction, you need to use *Expression Trees*. Expression Trees represent code as a structure that you examine, modify, or execute. These tools give you the power to manipulate code during run time. You write code that examines running algorithms, or injects new capabilities. In more advanced scenarios, you modify running algorithms and even translate C# expressions into another form for execution in another environment.

You compile and run code represented by expression trees. Building and running expression trees enables dynamic modification of executable code, the execution of LINQ queries in various databases, and the creation of dynamic queries. For more information about expression trees in LINQ, see How to use expression trees to build dynamic queries.

Expression trees are also used in the dynamic language runtime (DLR) to provide interoperability between dynamic languages and .NET and to enable compiler writers to emit expression trees instead of Microsoft intermediate language (MSIL). For more information about the DLR, see Dynamic Language Runtime Overview.

You can have the C# or Visual Basic compiler create an expression tree for you based on an anonymous lambda expression, or you can create expression trees manually by using the System.Linq.Expressions namespace.

When a lambda expression is assigned to a variable of type Expression<TDelegate>, the compiler emits code to build an expression tree that represents the lambda expression.

The C# compiler generates expression trees only from expression lambdas (or single-line lambdas). It can't parse statement lambdas (or multi-line lambdas). For more information about lambda expressions in C#, see Lambda Expressions.

The following code examples demonstrate how to have the C# compiler create an expression tree that represents the lambda expression `num => num < 5`.

C#

```
Expression<Func<int, bool>> lambda = num => num < 5;
```

You create expression trees in your code. You build the tree by creating each node and attaching the nodes into a tree structure. You learn how to create expressions in the article on building expression trees.

Expression trees are immutable. If you want to modify an expression tree, you must construct a new expression tree by copying the existing one and replacing nodes in it. You use an expression tree visitor to traverse the existing expression tree. For more information, see the article on translating expression trees.

Once you build an expression tree, you execute the code represented by the expression tree.

Limitations

There are some newer C# language elements that don't translate well into expression trees. Expression trees can't contain `await` expressions, or `async` lambda expressions. Many of the features added in C# 6 and later don't appear exactly as written in expression trees. Instead, newer features are exposed in expression trees in the equivalent, earlier syntax, where possible. Other constructs aren't available. It means that code that interprets expression trees works the same when new language features are introduced. However, even with these limitations, expression trees do enable you to create dynamic algorithms that rely on interpreting and modifying code that is represented as a data structure. It enables rich libraries such as Entity Framework to accomplish what they do.

Expression trees won't support new expression node types. It would be a breaking change for all libraries interpreting expression trees to introduce new node types. The following list includes most C# language elements that can't be used:

- Conditional methods that have been removed
- base access
- Method group expressions, including *address-of* (&) a method group, and anonymous method expressions
- References to local functions
- Statements, including assignment (`=`) and statement bodied expressions
- Partial methods with only a defining declaration
- Unsafe pointer operations
- dynamic operations
- Coalescing operators with null or default literal left side, null coalescing assignment, and the null propagating operator (?.)
- Multi-dimensional array initializers, indexed properties, and dictionary initializers
- throw expressions
- Accessing static virtual or abstract interface members
- Lambda expressions that have attributes
- Interpolated strings
- UTF-8 string conversions or UTF-8 string literals
- Method invocations using variable arguments, named arguments or optional arguments
- Expressions using System.Index or System.Range, index "from end" (^) operator or range expressions (..)
- async lambda expressions or await expressions, including await foreach and await using
- Tuple literals, tuple conversions, tuple == or !=, or with expressions
- Discards (_), deconstructing assignment, pattern matching is operator or the pattern matching switch expression
- COM call with `ref` omitted on the arguments
- ref, in or out parameters, `ref` return values, `out` arguments, or any values of ref struct type

Arborescences d'expression : données qui définissent le code

Article • 13/03/2023

Les arborescences d'expressions sont des structures de données qui définissent du code. Les arborescences d'expression sont basées sur les mêmes structures que celles utilisées par un compilateur pour analyser du code et générer la sortie compilée. Lors de la lecture de cet article, vous remarquez qu'il existe certaines similitudes entre les arborescences d'expressions et les types utilisés dans les API Roslyn pour générer des Analyzers et CodeFixes. (Analyzers et CodeFixes sont des packages NuGet qui effectuent une analyse statique sur le code et suggèrent des correctifs potentiels à un développeur.) Les concepts sont similaires et le résultat final est une structure de données qui permet d'examiner le code source de manière constructive. Toutefois, les arborescences d'expressions sont basées sur un ensemble de classes et d'API différent des API Roslyn. Voici une ligne de code :

```
C#

var sum = 1 + 2;
```

Si vous analysez le code précédent comme une arborescence d'expressions, l'arborescence contient plusieurs nœuds. Le nœud extérieur est une instruction de déclaration de variable avec attribution (`var sum = 1 + 2;`). Ce nœud extérieur contient plusieurs nœuds enfants : une déclaration de variable, un opérateur d'assignation et une expression qui représente la partie à droite du signe égal. Cette expression est sous-divisée en expressions qui représentent l'opération d'addition et les opérandes gauche et droit de l'addition.

Penchons-nous un peu plus sur les expressions qui composent la partie à droite du signe égal. L'expression est `1 + 2`, une expression binaire. Plus spécifiquement, il s'agit d'une expression d'addition binaire. Une expression d'addition binaire a deux enfants, représentant les nœuds gauche et droit de l'expression d'addition. Ici, les deux nœuds sont des expressions constantes : l'opérande gauche est la valeur `1`, et l'opérande droit est la valeur `2`.

Visuellement, l'instruction entière est une arborescence. Vous pouvez partir du nœud racine et accéder à chaque nœud dans l'arborescence pour voir le code qui compose l'instruction :

- Instruction de déclaration de variable avec attribution (`var sum = 1 + 2;`)

- Déclaration de type de variable implicite (`var sum`)
  - Mot clé var implicite (`var`)
  - Déclaration de nom de variable (`sum`)
- Opérateur d'assignation (`=`)
- Expression d'addition binaire (`1 + 2`)
  - Opérande gauche (`1`)
  - Opérateur d'addition (`+`)
  - Opérande droit (`2`)

L'arborescence précédente peut sembler compliquée, mais elle est très puissante. Suivant le même processus, vous décomposez des expressions beaucoup plus complexes. Examinons cette expression :

```
C#

var finalAnswer = this.SecretSauceFunction(
    currentState.createInterimResult(), currentState.createSecondValue(1,
2),
    decisionServer.considerFinalOptions("hello")) +
    MoreSecretSauce('A', DateTime.Now, true);
```

L'expression précédente est aussi une déclaration de variable avec une assignation. La partie droite de l'assignation est une arborescence beaucoup plus complexe. Vous n'allez pas décomposer cette expression, mais réfléchissez à ce que pourraient être les différents nœuds. Il y a des appels de méthode utilisant l'objet actif comme destinataire, un qui a un récepteur `this` explicite, un autre qui n'en a pas. Il y a des appels de méthode utilisant d'autres objets récepteurs, et des arguments constants de différents types. Pour finir, il y a un opérateur d'addition binaire. En fonction du type de retour de `SecretSauceFunction()` ou `MoreSecretSauce()`, cet opérateur d'addition binaire peut être un appel de méthode à un opérateur d'addition substitué, résolu en un appel de méthode statique à l'opérateur d'addition binaire défini pour une classe.

Malgré cette complexité apparente, l'expression précédente crée une arborescence qui peut être parcourue aussi facilement que le premier exemple. Vous traversez les nœuds enfants successifs pour rechercher des nœuds terminaux dans l'expression. Les nœuds parents ont des références à leurs enfants, et chaque nœud a une propriété qui décrit le genre de nœud dont il s'agit.

La structure d'une arborescence d'expressions est très cohérente. Une fois que vous maîtrisez les principes de base, vous comprenez le code le plus complexe quand il est représenté sous forme d'arborescence d'expressions. L'élégance dans la structure de

données explique comment le compilateur C# analyse les programmes C# les plus complexes et crée une sortie appropriée à partir de ce code source complexe.

Une fois que vous vous serez familiarisé avec la structure des arborescences d'expressions, vous constatez que les connaissances acquises vous permettront de travailler rapidement avec de nombreux scénarios avancés. Les arborescences d'expressions offrent une puissance incroyable.

Outre la conversion d'algorithmes à exécuter dans d'autres environnements, utilisez des arborescences d'expressions pour simplifier l'écriture d'algorithmes qui inspectent du code avant de l'exécuter. Vous écrivez une méthode dont les arguments sont des expressions, puis examinez ces expressions avant d'exécuter le code. L'arborescence d'expressions est une représentation complète du code : vous voyez les valeurs de n'importe quelle sous-expression. Vous voyez les noms des méthodes et des propriétés. Vous voyez la valeur de n'importe quelle expression constante. Vous convertissez une arborescence d'expressions en délégué exécutable et exécutez le code.

Les API d'arborescences d'expressions vous permettent de créer des arborescences qui représentent presque n'importe quelle construction de code valide. Toutefois, pour simplifier les choses au maximum, certains idiomes C# ne peuvent pas être créés dans une arborescence d'expressions. C'est le cas des expressions asynchrones (utilisant les mots clés `async` et `await`). Si vous avez besoin d'algorithmes asynchrones, vous devez manipuler directement les objets `Task` plutôt que de vous fier à la prise en charge du compilateur. C'est aussi le cas pour la création des boucles. En général, vous les créez à l'aide de boucles `for`, `foreach`, `while` ou `do`. Comme vous le verrez plus loin dans cette série, les API d'arborescences d'expressions prennent en charge une expression de boucle unique, avec des expressions `break` et `continue` qui contrôlent la répétition de la boucle.

La seule chose que vous ne pouvez pas faire est de modifier une arborescence d'expressions. Les arborescences d'expressions sont des structures de données immuables. Si vous souhaitez muter (changer) une expression de l'arborescence, vous devez créer une nouvelle arborescence qui est une copie de l'original, mais avec les modifications souhaitées.

Prise en charge du runtime .NET pour les arborescences d'expression

Article • 13/03/2023

Il existe une longue liste de classes dans le runtime .NET qui fonctionnent avec des arborescences d'expressions. Vous pouvez voir la liste complète dans System.Linq.Expressions. Au lieu d'énumérer la liste complète, nous allons comprendre comment les classes du runtime ont été conçues.

Dans la conception du langage, une expression est un corps de code qui évalue et retourne une valeur. Les expressions peuvent être simples : l'expression constante `1` retourne la valeur constante 1. Elles peuvent être plus compliquées : l'expression `(-B + Math.Sqrt(B*B - 4 * A * C)) / (2 * A)` retourne une racine pour une équation quadratique (dans le cas où l'équation a une solution).

System.Linq.Expression et types dérivés

L'une des complexités liées à l'utilisation des arborescences d'expressions est que de nombreuses sortes d'expressions sont valides à de nombreux endroits dans les programmes. Prenez par exemple une expression d'assignation. La partie droite d'une assignation peut être une valeur constante, une variable, une expression d'appel de méthode ou autre. Cette flexibilité linguistique signifie que vous pouvez rencontrer de nombreux types d'expressions différents à n'importe quel endroit dans les nœuds d'une arborescence quand vous parcourez une arborescence d'expressions. Ainsi, quand vous travaillez avec le type d'expression de base, c'est le moyen le plus simple de procéder. Cependant, vous devez parfois en savoir plus. La classe Expression de base contient une propriété `NodeType` à cet effet. Elle retourne un `ExpressionType`, qui est une énumération des types d'expressions possibles. Une fois que vous connaissez le type du nœud, vous le castez en ce type et effectuer des actions spécifiques. Vous pouvez rechercher certains types de nœuds, puis utiliser les propriétés spécifiques de ce genre d'expression.

Par exemple, ce code imprime le nom d'une variable pour une expression d'accès à la variable. Le code suivant montre la pratique qui consiste à vérifier le type de nœud, à caster en une expression d'accès à la variable, puis à vérifier les propriétés du type d'expression spécifique :

```
C#
```

```
Expression<Func<int, int>> addFive = (num) => num + 5;

if (addFive is LambdaExpression lambdaExp)
{
    var parameter = lambdaExp.Parameters[0];  -- first

    Console.WriteLine(parameter.Name);
    Console.WriteLine(parameter.Type);
}
```

Créer des arborescences d'expressions

La classe `System.Linq.Expression` contient également de nombreuses méthodes statiques pour créer des expressions. Ces méthodes créent un nœud d'expression en utilisant les arguments fournis pour ses enfants. De cette façon, vous générez une expression à partir de ses nœuds terminaux. Par exemple, ce code génère une expression Add :

C#

```
// Addition is an add expression for "1 + 2"
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var addition = Expression.Add(one, two);
```

Vous pouvez constater d'après cet exemple simple que de nombreux types sont impliqués dans la création et l'utilisation des arborescences d'expressions. Cette complexité est nécessaire pour fournir les fonctionnalités du vocabulaire enrichi fourni par le langage C#.

Parcourir les API

Il existe des types de nœuds Expression qui mappent à presque tous les éléments de syntaxe du langage C#. Chaque type a des méthodes spécifiques pour ce type d'élément de langage. Cela fait beaucoup de choses à mémoriser en même temps. Plutôt que d'essayer de mémoriser tout, voici les techniques que vous appliquez quand vous travaillez avec des arborescences d'expressions :

1. Examinez les membres de l'énumération `ExpressionType` pour déterminer les éventuels nœuds à examiner. Cette liste aide vraiment quand vous souhaitez parcourir et comprendre une arborescence d'expressions.

2. Examinez les membres statiques de la classe `Expression` pour générer une expression. Ces méthodes peuvent générer tout type d'expression à partir d'un ensemble de ses nœuds enfants.
3. Examinez la classe `ExpressionVisitor` pour générer une arborescence d'expressions modifiée.

Vous trouvez davantage de choses à mesure que vous examinez chacune de ces trois zones. Vous trouvez invariablement ce dont vous avez besoin si vous commencez avec l'une de ces trois étapes.

Exécuter des arborescences d'expressions

Article • 13/03/2023

Une *arborescence d'expressions* est une structure de données qui représente du code. Il ne s'agit pas de code compilé et exécutable. Si vous souhaitez exécuter le code .NET représenté par une arborescence d'expressions, vous devez le convertir en instructions de langage intermédiaire exécutables. L'exécution d'une arborescence d'expressions peut retourner une valeur, ou elle peut simplement effectuer une action telle que l'appel d'une méthode.

Seules les arborescences d'expressions qui représentent des expressions lambda peuvent être exécutées. Les arborescences d'expressions qui représentent des expressions lambda peuvent être de type LambdaExpression ou Expression<TDelegate>. Pour exécuter ces arborescences d'expressions, appelez la méthode Compile pour créer un délégué exécutable, puis appelez le délégué.

> ⓘ **Notes**
>
> Si le type du délégué n'est pas connu, autrement dit si l'expression lambda est de type **LambdaExpression** et non **Expression<TDelegate>**, appelez la méthode **DynamicInvoke** sur le délégué au lieu de l'appeler directement.

Si une arborescence d'expressions ne représente pas une expression lambda, vous pouvez créer une expression lambda ayant l'arborescence d'expressions d'origine comme corps, en appelant la méthode Lambda<TDelegate>(Expression, IEnumerable<ParameterExpression>). Ensuite, vous pouvez exécuter l'expression lambda comme décrit plus haut dans cette section.

Des expressions lambda aux fonctions

Vous pouvez convertir n'importe quelle LambdaExpression ou n'importe quel type dérivé de LambdaExpression en langage intermédiaire exécutable. Les autres types d'expressions ne peuvent pas être convertis directement en code. Cette restriction a peu d'effet dans la pratique. Les expressions lambda sont les seuls types d'expressions que vous pourriez souhaiter exécuter par l'intermédiaire d'une conversion en langage intermédiaire exécutable. (Réfléchissez à ce que signifie l'exécution directe d'une System.Linq.Expressions.ConstantExpression. Est-ce que cela veut dire quelque chose d'utile ?) Toute arborescence de l'expression qui est une

System.Linq.Expressions.LambdaExpression ou un type dérivé de `LambdaExpression` peut être convertie en langage intermédiaire (IL). Le type d'expression System.Linq.Expressions.Expression<TDelegate> est le seul exemple concret dans les bibliothèques .NET Core. Il sert à représenter une expression mappée à un type délégué quelconque. Ce type étant mappé à un type délégué, .NET peut examiner l'expression et générer le langage intermédiaire pour un délégué approprié qui correspond à la signature de l'expression lambda. Le type délégué est basé sur le type d'expression. Si vous souhaitez utiliser l'objet délégué d'une manière fortement typée, vous devez connaître le type de retour et la liste d'arguments. La méthode `LambdaExpression.Compile()` retourne le type `Delegate`. Vous devez effectuer un cast vers le type délégué approprié pour que les outils de compilation puissent vérifier la liste d'arguments de type de retour.

Dans la plupart des cas, un mappage simple entre une expression et son délégué correspondant existe. Par exemple, une arborescence d'expressions représentée par `Expression<Func<int>>` serait convertie en un délégué du type `Func<int>`. Pour une expression lambda avec tout type de retour et liste d'arguments, il existe un type délégué qui est le type cible pour le code exécutable représenté par cette expression lambda.

Le type System.Linq.Expressions.LambdaExpression contient des membres LambdaExpression.Compile et LambdaExpression.CompileToMethod que vous utiliseriez pour convertir une arborescence d'expressions en code exécutable. La méthode `Compile` crée un délégué. La méthode `CompileToMethod` met à jour un objet System.Reflection.Emit.MethodBuilder avec le langage intermédiaire qui représente la sortie compilée de l'arborescence d'expressions.

> **ⓘ Important**
>
> `CompileToMethod` est disponible uniquement dans .NET Framework, et non dans .NET Core ou .NET 5 et versions ultérieures.

Si vous le souhaitez, vous pouvez également fournir un System.Runtime.CompilerServices.DebugInfoGenerator qui reçoit les informations de débogage de symbole pour l'objet délégué généré. `DebugInfoGenerator` fournit des informations de débogage complètes sur le délégué généré.

Vous pouvez convertir une expression en délégué à l'aide du code suivant :

```
C#
```

```
Expression<Func<int>> add = () => 1 + 2;
var func = add.Compile(); // Create Delegate
var answer = func(); // Invoke Delegate
Console.WriteLine(answer);
```

L'exemple de code suivant illustre les types concrets utilisés lorsque vous compilez et exécutez une arborescence d'expression.

C#

```
Expression<Func<int, bool>> expr = num => num < 5;

// Compiling the expression tree into a delegate.
Func<int, bool> result = expr.Compile();

// Invoking the delegate and writing the result to the console.
Console.WriteLine(result(4));

// Prints True.

// You can also use simplified syntax
// to compile and run an expression tree.
// The following line can replace two previous statements.
Console.WriteLine(expr.Compile()(4));

// Also prints True.
```

L'exemple de code suivant montre comment exécuter une arborescence d'expressions qui représente l'élévation d'un nombre à une puissance en créant une expression lambda et en l'exécutant. Le résultat, qui représente le nombre élevé à la puissance, est affiché.

C#

```
// The expression tree to execute.
BinaryExpression be = Expression.Power(Expression.Constant(2d),
Expression.Constant(3d));

// Create a lambda expression.
Expression<Func<double>> le = Expression.Lambda<Func<double>>(be);

// Compile the lambda expression.
Func<double> compiledExpression = le.Compile();

// Execute the lambda expression.
double result = compiledExpression();

// Display the result.
Console.WriteLine(result);
```

```
// This code produces the following output:
// 8
```

Exécution et durées de vie

Vous exécutez le code en appelant le délégué créé quand vous avez appelé `LambdaExpression.Compile()`. Le code précédent, `add.Compile()`, retourne un délégué. Vous appelez ce délégué en appelant `func()`, qui exécute le code.

Ce délégué représente le code dans l'arborescence d'expressions. Vous pouvez conserver le handle de ce délégué et l'appeler ultérieurement. Vous n'avez pas besoin de compiler l'arborescence d'expressions chaque fois que vous souhaitez exécuter le code qu'elle représente. (N'oubliez pas que les arborescences d'expressions sont immuables ; compiler la même arborescence d'expressions ultérieurement crée un délégué qui exécute le même code.)

> **⊗ Attention**
>
> Évitez de créer des mécanismes de mise en cache plus complexes pour améliorer les performances en évitant les appels de compilation inutiles. Comparer deux arborescences d'expressions arbitraires pour déterminer si elles représentent le même algorithme est une opération qui prend du temps. Le temps de calcul économisé en évitant tout appel supplémentaire à `LambdaExpression.Compile()` sera probablement largement contrebalancé par le temps passé à exécuter le code qui détermine si deux arborescences d'expressions génèrent le même code exécutable.

Mises en garde

Compiler une expression lambda en un délégué et appeler ce délégué sont l'une des opérations les plus simples que vous puissiez effectuer avec une arborescence d'expressions. Toutefois, même avec cette simple opération, il existe des pièges dont vous devez être conscient.

Les expressions lambda créent des fermetures sur toutes les variables locales qui sont référencées dans l'expression. Vous devez vérifier que toutes les variables qui font partie du délégué sont utilisables à l'emplacement où vous appelez `Compile` et quand vous exécutez le délégué résultant. Le compilateur garantit que les variables sont dans l'étendue. Toutefois, si votre expression accède à une variable qui implémente

`IDisposable`, il est possible que votre code supprime l'objet alors qu'il est toujours détenu par l'arborescence d'expressions.

Par exemple, ce code fonctionne bien, car `int` n'implémente pas `IDisposable` :

C#

```
private static Func<int, int> CreateBoundFunc()
{
    var constant = 5; // constant is captured by the expression tree
    Expression<Func<int, int>> expression = (b) => constant + b;
    var rVal = expression.Compile();
    return rVal;
}
```

Le délégué a capturé une référence à la variable locale `constant`. Cette variable est accessible à tout moment ultérieurement, quand la fonction retournée par `CreateBoundFunc` s'exécute.

Maintenant, examinez la classe qui suit, qui implémente System.IDisposable :

C#

```
public class Resource : IDisposable
{
    private bool _isDisposed = false;
    public int Argument
    {
        get
        {
            if (!_isDisposed)
                return 5;
            else throw new ObjectDisposedException("Resource");
        }
    }

    public void Dispose()
    {
        _isDisposed = true;
    }
}
```

Si vous l'utilisez dans une expression comme illustré dans le code suivant, vous obtenez une System.ObjectDisposedException quand vous exécutez le code référencé par la propriété `Resource.Argument` :

C#

```
private static Func<int, int> CreateBoundResource()
{
    using (var constant = new Resource()) // constant is captured by the
expression tree
    {
        Expression<Func<int, int>> expression = (b) => constant.Argument +
b;
        var rVal = expression.Compile();
        return rVal;
    }
}
```

Le délégué retourné par cette méthode a fermé l'objet `constant`, qui a été supprimé. (Il a été supprimé car il a été déclaré dans une instruction `using`.)

Désormais, quand vous exécuterez le délégué retourné par cette méthode, une `ObjectDisposedException` est levée au moment de l'exécution.

Cela semble étrange d'avoir une erreur d'exécution qui représente une construction de compilation, mais cela fait partie des éventualités quand vous travaillez avec des arborescences d'expressions.

Comme il existe de nombreuses permutations de ce problème, il est difficile de proposer des conseils généraux pour l'éviter. Faites attention si vous accédez à des variables locales quand vous définissez des expressions et faites attention si vous accédez à l'état de l'objet actif (représenté par `this`) quand vous créez une arborescence d'expressions qui peut être retournée par une API publique.

Le code dans votre expression peut référencer des méthodes ou des propriétés dans d'autres assemblys. Ces assemblys doivent être accessibles quand l'expression est définie, quand elle est compilée et quand le délégué résultant est appelé. S'il n'est pas présent, vous recevrez une `ReferencedAssemblyNotFoundException`.

Résumé

Les arborescences d'expressions qui représentent des expressions lambda peuvent être compilées pour créer un délégué exécutable. Les arborescences d'expression fournissent un mécanisme pour exécuter le code représenté par une arborescence d'expressions.

L'arborescence d'expressions représente le code qui s'exécuterait pour toute construction donnée que vous créez. Tant que l'environnement dans lequel vous compilez et exécutez le code correspond à celui dans lequel vous créez une expression, tout fonctionne comme prévu. Si ce n'est pas le cas, les erreurs sont prévisibles et sont

interceptées lors de vos premiers tests de n'importe quel code utilisant les arborescences d'expressions.

Interpréter les expressions

Article • 13/03/2023

L'exemple de code suivant montre comment l'arborescence d'expressions qui représente l'expression lambda `num => num < 5` peut être décomposée selon ses différentes parties.

C#

```
// Add the following using directive to your code file:
// using System.Linq.Expressions;

// Create an expression tree.
Expression<Func<int, bool>> exprTree = num => num < 5;

// Decompose the expression tree.
ParameterExpression param = (ParameterExpression)exprTree.Parameters[0];
BinaryExpression operation = (BinaryExpression)exprTree.Body;
ParameterExpression left = (ParameterExpression)operation.Left;
ConstantExpression right = (ConstantExpression)operation.Right;

Console.WriteLine("Decomposed expression: {0} => {1} {2} {3}",
                  param.Name, left.Name, operation.NodeType, right.Value);

// This code produces the following output:

// Decomposed expression: num => num LessThan 5
```

Nous allons maintenant écrire du code pour analyser la structure d'une *arborescence d'expressions*. Chaque nœud d'une arborescence d'expressions est un objet d'une classe dérivée de `Expression`.

Grâce à cette conception, visiter tous les nœuds d'une arborescence d'expressions est une opération récursive relativement simple. La stratégie générale consiste à débuter au niveau du nœud racine et à déterminer de quel genre de nœud il s'agit.

Si le type de nœud a des enfants, visitez-les de manière récursive. Sur chaque nœud enfant, répétez le processus utilisé au niveau du nœud racine : déterminez le type et, s'il a des enfants, visitez chacun d'eux.

Examen d'une expression sans enfants

Commençons par visiter chaque nœud dans une arborescence d'expressions simple. Voici le code qui crée une expression constante, puis examine ses propriétés :

C#

```
var constant = Expression.Constant(24, typeof(int));

Console.WriteLine($"This is a/an {constant.NodeType} expression type");
Console.WriteLine($"The type of the constant value is {constant.Type}");
Console.WriteLine($"The value of the constant value is {constant.Value}");
```

Le code ci-dessus imprime la sortie suivante :

Sortie

```
This is a/an Constant expression type
The type of the constant value is System.Int32
The value of the constant value is 24
```

Maintenant, nous allons écrire le code qui examinerait cette expression et écrirait certaines propriétés importantes à son sujet.

Expression binaire

Commençons par l'exemple d'addition issu de l'introduction à cette section.

C#

```
Expression<Func<int>> sum = () => 1 + 2;
```

> ⓘ Notes
>
> N'utilisez pas `var` pour déclarer cette arborescence d'expression, car le type naturel du délégué est `Func<int>`, et non `Expression<Func<int>>`.

Le nœud racine est `LambdaExpression`. Pour obtenir le code intéressant du côté droit de l'opérateur `=>`, nous devons trouver l'un des enfants de `LambdaExpression`. Vous allez le faire avec toutes les expressions dans cette section. Le nœud parent nous aide à trouver le type de retour de `LambdaExpression`.

Pour examiner chaque nœud de cette expression, vous devez visiter plusieurs nœuds de manière récursive. Voici une première implémentation simple :

C#

```
Expression<Func<int, int, int>> addition = (a, b) => a + b;

Console.WriteLine($"This expression is a {addition.NodeType} expression
type");
Console.WriteLine($"The name of the lambda is {((addition.Name == null) ? "
<null>" : addition.Name)}");
Console.WriteLine($"The return type is {addition.ReturnType.ToString()}");
Console.WriteLine($"The expression has {addition.Parameters.Count}
arguments. They are:");
foreach (var argumentExpression in addition.Parameters)
{
    Console.WriteLine($"\tParameter Type:
{argumentExpression.Type.ToString()}, Name: {argumentExpression.Name}");
}

var additionBody = (BinaryExpression)addition.Body;
Console.WriteLine($"The body is a {additionBody.NodeType} expression");
Console.WriteLine($"The left side is a {additionBody.Left.NodeType}
expression");
var left = (ParameterExpression)additionBody.Left;
Console.WriteLine($"\tParameter Type: {left.Type.ToString()}, Name:
{left.Name}");
Console.WriteLine($"The right side is a {additionBody.Right.NodeType}
expression");
var right = (ParameterExpression)additionBody.Right;
Console.WriteLine($"\tParameter Type: {right.Type.ToString()}, Name:
{right.Name}");
```

Cet exemple génère la sortie suivante :

Sortie

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 arguments. They are:
        Parameter Type: System.Int32, Name: a
        Parameter Type: System.Int32, Name: b
The body is a/an Add expression
The left side is a Parameter expression
        Parameter Type: System.Int32, Name: a
The right side is a Parameter expression
        Parameter Type: System.Int32, Name: b
```

Vous remarquez beaucoup de répétitions dans l'exemple de code précédent. Nous allons le nettoyer et générer un visiteur de nœud d'expression plus général. Pour cela, nous allons devoir écrire un algorithme récursif. Tout nœud peut être d'un type susceptible d'avoir des enfants. Tout nœud ayant des enfants nous oblige à visiter ces

enfants et à déterminer ce qu'est ce nœud. Voici la version nettoyée qui utilise la récursivité pour visiter les opérations d'addition :

C#

```
using System.Linq.Expressions;

namespace Visitors;
// Base Visitor class:
public abstract class Visitor
{
    private readonly Expression node;

    protected Visitor(Expression node) => this.node = node;

    public abstract void Visit(string prefix);

    public ExpressionType NodeType => node.NodeType;
    public static Visitor CreateFromExpression(Expression node) =>
        node.NodeType switch
        {
            ExpressionType.Constant => new
ConstantVisitor((ConstantExpression)node),
            ExpressionType.Lambda => new
LambdaVisitor((LambdaExpression)node),
            ExpressionType.Parameter => new
ParameterVisitor((ParameterExpression)node),
            ExpressionType.Add => new BinaryVisitor((BinaryExpression)node),
            _ => throw new NotImplementedException($"Node not processed yet:
{node.NodeType}"),
        };
}

// Lambda Visitor
public class LambdaVisitor : Visitor
{
    private readonly LambdaExpression node;
    public LambdaVisitor(LambdaExpression node) : base(node) => this.node =
node;

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType}
expression type");
        Console.WriteLine($"{prefix}The name of the lambda is {((node.Name
== null) ? "<null>" : node.Name)}");
        Console.WriteLine($"{prefix}The return type is {node.ReturnType}");
        Console.WriteLine($"{prefix}The expression has
{node.Parameters.Count} argument(s). They are:");
        // Visit each parameter:
        foreach (var argumentExpression in node.Parameters)
        {
            var argumentVisitor = CreateFromExpression(argumentExpression);
            argumentVisitor.Visit(prefix + "\t");
```

```
        }
        Console.WriteLine($"{prefix}The expression body is:");
        // Visit the body:
        var bodyVisitor = CreateFromExpression(node.Body);
        bodyVisitor.Visit(prefix + "\t");
    }
}

// Binary Expression Visitor:
public class BinaryVisitor : Visitor
{
    private readonly BinaryExpression node;
    public BinaryVisitor(BinaryExpression node) : base(node) => this.node =
node;

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This binary expression is a {NodeType}
expression");
        var left = CreateFromExpression(node.Left);
        Console.WriteLine($"{prefix}The Left argument is:");
        left.Visit(prefix + "\t");
        var right = CreateFromExpression(node.Right);
        Console.WriteLine($"{prefix}The Right argument is:");
        right.Visit(prefix + "\t");
    }
}

// Parameter visitor:
public class ParameterVisitor : Visitor
{
    private readonly ParameterExpression node;
    public ParameterVisitor(ParameterExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This is an {NodeType} expression type");
        Console.WriteLine($"{prefix}Type: {node.Type}, Name: {node.Name},
ByRef: {node.IsByRef}");
    }
}

// Constant visitor:
public class ConstantVisitor : Visitor
{
    private readonly ConstantExpression node;
    public ConstantVisitor(ConstantExpression node) : base(node) =>
this.node = node;

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This is an {NodeType} expression type");
```

```
        Console.WriteLine($"{prefix}The type of the constant value is
{node.Type}");
        Console.WriteLine($"{prefix}The value of the constant value is
{node.Value}");
    }
}
```

Cet algorithme est la base d'un algorithme qui visite toute `LambdaExpression` arbitraire. Le code que vous avez créé recherche uniquement un petit échantillon des ensembles possibles de nœuds d'arborescence d'expression qu'il peut rencontrer. mais ce qu'il génère peut quand même nous en apprendre beaucoup. (Le cas par défaut dans la méthode `Visitor.CreateFromExpression` affiche un message dans la console d'erreurs quand un nouveau type de nœud est détecté. Ainsi, vous savez que vous devez ajouter un nouveau type d'expression.)

Quand nous exécutons ce visiteur sur l'expression d'addition précédente, nous obtenons la sortie suivante :

Sortie

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 argument(s). They are:
        This is an Parameter expression type
        Type: System.Int32, Name: a, ByRef: False
        This is an Parameter expression type
        Type: System.Int32, Name: b, ByRef: False
The expression body is:
        This binary expression is a Add expression
        The Left argument is:
                This is an Parameter expression type
                Type: System.Int32, Name: a, ByRef: False
        The Right argument is:
                This is an Parameter expression type
                Type: System.Int32, Name: b, ByRef: False
```

Maintenant que nous avons créé une implémentation de visiteur plus générale, nous pouvons visiter et traiter de nombreux autres types d'expressions.

Expression d'addition avec d'autres opérandes

Essayons un exemple plus complexe, tout en continuant de limiter les types de nœuds à l'addition uniquement :

C#

```
Expression<Func<int>> sum = () => 1 + 2 + 3 + 4;
```

Avant d'exécuter ces exemples sur l'algorithme visiteur, essayez de réfléchir à ce que pourrait être le résultat. N'oubliez pas que l'opérateur `+` est un *opérateur binaire* : il doit avoir deux enfants, représentant les opérandes gauche et droit. Il existe plusieurs manières correctes de construire une arborescence :

C#

```
Expression<Func<int>> sum1 = () => 1 + (2 + (3 + 4));
Expression<Func<int>> sum2 = () => ((1 + 2) + 3) + 4;

Expression<Func<int>> sum3 = () => (1 + 2) + (3 + 4);
Expression<Func<int>> sum4 = () => 1 + ((2 + 3) + 4);
Expression<Func<int>> sum5 = () => (1 + (2 + 3)) + 4;
```

Vous pouvez voir la séparation en deux réponses possibles, pour mettre en évidence la plus prometteuse. La première représente les expressions *associatives à droite*. La deuxième représente les expressions *associatives à gauche*. L'avantage de ces deux formats est qu'ils peuvent s'adapter à un nombre arbitraire d'expressions d'addition.

Si nous exécutons cette expression dans le visiteur, nous obtenons ce résultat, qui vérifie que l'expression d'addition simple est *associative à gauche*.

Pour exécuter cet exemple et voir l'arborescence d'expressions complète, vous apportez une modification à l'arborescence d'expressions source. Quand l'arborescence d'expressions contient uniquement des constantes, l'arborescence résultante contient simplement la valeur constante `10`. Le compilateur effectue toute l'addition et réduit l'expression à sa forme la plus simple. Le simple ajout d'une variable dans l'expression suffit pour voir l'arborescence d'origine :

C#

```
Expression<Func<int, int>> sum = (a) => 1 + a + 3 + 4;
```

Créez un visiteur pour cette somme et exécutez-le. Vous obtenez cette sortie :

Sortie

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 1 argument(s). They are:
        This is an Parameter expression type
```

```
        Type: System.Int32, Name: a, ByRef: False
The expression body is:
        This binary expression is a Add expression
        The Left argument is:
                This binary expression is a Add expression
                The Left argument is:
                        This binary expression is a Add expression
                        The Left argument is:
                                This is an Constant expression type
                                The type of the constant value is
System.Int32
                                The value of the constant value is 1
                        The Right argument is:
                                This is an Parameter expression type
                                Type: System.Int32, Name: a, ByRef: False
                The Right argument is:
                        This is an Constant expression type
                        The type of the constant value is System.Int32
                        The value of the constant value is 3
        The Right argument is:
                This is an Constant expression type
                The type of the constant value is System.Int32
                The value of the constant value is 4
```

Vous pouvez exécuter l'un des autres exemples dans le code de visiteur et voir quelle arborescence il représente. Voici un exemple de l'expression `sum3` précédente (avec un paramètre supplémentaire pour empêcher le compilateur de calculer la constante) :

C#

```
Expression<Func<int, int, int>> sum3 = (a, b) => (1 + a) + (3 + b);
```

Voici la sortie du visiteur :

Sortie

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 argument(s). They are:
        This is an Parameter expression type
        Type: System.Int32, Name: a, ByRef: False
        This is an Parameter expression type
        Type: System.Int32, Name: b, ByRef: False
The expression body is:
        This binary expression is a Add expression
        The Left argument is:
                This binary expression is a Add expression
                The Left argument is:
                        This is an Constant expression type
                        The type of the constant value is System.Int32
```

```
                        The value of the constant value is 1
                The Right argument is:
                        This is an Parameter expression type
                        Type: System.Int32, Name: a, ByRef: False
        The Right argument is:
                This binary expression is a Add expression
                The Left argument is:
                        This is an Constant expression type
                        The type of the constant value is System.Int32
                        The value of the constant value is 3
                The Right argument is:
                        This is an Parameter expression type
                        Type: System.Int32, Name: b, ByRef: False
```

Notez que les parenthèses ne font pas partie de la sortie. Il n'y a aucun nœud dans l'arborescence d'expressions qui représente les parenthèses dans l'expression d'entrée. La structure de l'arborescence d'expressions contient toutes les informations nécessaires pour communiquer le niveau de priorité.

Extension de cet exemple

L'exemple traite uniquement les arborescences d'expressions les plus rudimentaires. Le code que nous avons vu dans cette section gère seulement des entiers constants et l'opérateur binaire `+`. En guise de dernier exemple, nous allons mettre à jour le visiteur pour gérer une expression plus complexe. Faisons en sorte que cela fonctionne pour l'expression factorielle suivante :

C#

```
Expression<Func<int, int>> factorial = (n) =>
    n == 0 ?
    1 :
    Enumerable.Range(1, n).Aggregate((product, factor) => product * factor);
```

Il représente une implémentation possible de la fonction mathématique *factorial*. La façon dont vous avez écrit ce code met en évidence deux limitations concernant la méthode de création d'arborescences d'expressions qui consiste à assigner des expressions lambda à Expression. Tout d'abord, les lambda-instructions ne sont pas autorisées. Cela signifie que vous ne pouvez pas utiliser de boucles, de blocs, d'instructions if/else et d'autres structures de contrôle courantes en C#. Vous êtes limité à l'utilisation d'expressions. Ensuite, vous ne pouvez pas appeler la même expression de manière récursive. Vous le pourriez s'il s'agissait déjà d'un délégué, mais vous ne pouvez pas l'appeler sous sa forme d'arborescence d'expressions. Dans la section sur la

génération d'arborescences de l'expression, vous découvrez des techniques pour surmonter ces restrictions.

Dans cette expression, vous rencontrez des nœuds de tous ces types :

1. Égal (expression binaire)
2. Multiplier (expression binaire)
3. Expression conditionnelle (expression `? :`)
4. Expression d'appel de méthode (appel de `Range()` et `Aggregate()`)

L'une des manières de modifier l'algorithme visiteur consiste à continuer à l'exécuter, et à écrire le type de nœud chaque fois que vous atteignez votre clause `default`. Après quelques itérations, vous aurez vu chacun des nœuds potentiels. Vous aurez alors tout ce qu'il vous faut. Le résultat ressemblera à ceci :

C#

```
public static Visitor CreateFromExpression(Expression node) =>
    node.NodeType switch
    {
        ExpressionType.Constant    => new
ConstantVisitor((ConstantExpression)node),
        ExpressionType.Lambda      => new
LambdaVisitor((LambdaExpression)node),
        ExpressionType.Parameter   => new
ParameterVisitor((ParameterExpression)node),
        ExpressionType.Add         => new
BinaryVisitor((BinaryExpression)node),
        ExpressionType.Equal       => new
BinaryVisitor((BinaryExpression)node),
        ExpressionType.Multiply    => new BinaryVisitor((BinaryExpression)
node),
        ExpressionType.Conditional => new
ConditionalVisitor((ConditionalExpression) node),
        ExpressionType.Call        => new
MethodCallVisitor((MethodCallExpression) node),
        _ => throw new NotImplementedException($"Node not processed yet:
{node.NodeType}"),
    };
```

`ConditionalVisitor` et `MethodCallVisitor` traitent ces deux nœuds :

C#

```
public class ConditionalVisitor : Visitor
{
    private readonly ConditionalExpression node;
    public ConditionalVisitor(ConditionalExpression node) : base(node)
    {
```

```
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType}
expression");
        var testVisitor = Visitor.CreateFromExpression(node.Test);
        Console.WriteLine($"{prefix}The Test for this expression is:");
        testVisitor.Visit(prefix + "\t");
        var trueVisitor = Visitor.CreateFromExpression(node.IfTrue);
        Console.WriteLine($"{prefix}The True clause for this expression
is:");
        trueVisitor.Visit(prefix + "\t");
        var falseVisitor = Visitor.CreateFromExpression(node.IfFalse);
        Console.WriteLine($"{prefix}The False clause for this expression
is:");
        falseVisitor.Visit(prefix + "\t");
    }
}

public class MethodCallVisitor : Visitor
{
    private readonly MethodCallExpression node;
    public MethodCallVisitor(MethodCallExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType}
expression");
        if (node.Object == null)
            Console.WriteLine($"{prefix}This is a static method call");
        else
        {
            Console.WriteLine($"{prefix}The receiver (this) is:");
            var receiverVisitor = Visitor.CreateFromExpression(node.Object);
            receiverVisitor.Visit(prefix + "\t");
        }

        var methodInfo = node.Method;
        Console.WriteLine($"{prefix}The method name is
{methodInfo.DeclaringType}.{methodInfo.Name}");
        // There is more here, like generic arguments, and so on.
        Console.WriteLine($"{prefix}The Arguments are:");
        foreach (var arg in node.Arguments)
        {
            var argVisitor = Visitor.CreateFromExpression(arg);
            argVisitor.Visit(prefix + "\t");
        }
    }
}
```

Et la sortie de l'arborescence d'expressions sera :

Sortie

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 1 argument(s). They are:
        This is an Parameter expression type
        Type: System.Int32, Name: n, ByRef: False
The expression body is:
        This expression is a Conditional expression
        The Test for this expression is:
                This binary expression is a Equal expression
                The Left argument is:
                        This is an Parameter expression type
                        Type: System.Int32, Name: n, ByRef: False
                The Right argument is:
                        This is an Constant expression type
                        The type of the constant value is System.Int32
                        The value of the constant value is 0
        The True clause for this expression is:
                This is an Constant expression type
                The type of the constant value is System.Int32
                The value of the constant value is 1
        The False clause for this expression is:
                This expression is a Call expression
                This is a static method call
                The method name is System.Linq.Enumerable.Aggregate
                The Arguments are:
                        This expression is a Call expression
                        This is a static method call
                        The method name is System.Linq.Enumerable.Range
                        The Arguments are:
                                This is an Constant expression type
                                The type of the constant value is
System.Int32
                                The value of the constant value is 1
                                This is an Parameter expression type
                                Type: System.Int32, Name: n, ByRef: False
                        This expression is a Lambda expression type
                        The name of the lambda is <null>
                        The return type is System.Int32
                        The expression has 2 arguments. They are:
                                This is an Parameter expression type
                                Type: System.Int32, Name: product, ByRef:
False
                                This is an Parameter expression type
                                Type: System.Int32, Name: factor, ByRef:
False
                        The expression body is:
                                This binary expression is a Multiply
expression
                                The Left argument is:
```

```
                                        This is an Parameter expression type
                                        Type: System.Int32, Name: product,
ByRef: False
                                The Right argument is:
                                        This is an Parameter expression type
                                        Type: System.Int32, Name: factor,
ByRef: False
```

Extension de la bibliothèque d'exemples

Les exemples de cette section illustrent les techniques de base pour visiter et examiner des nœuds dans une arborescence d'expressions. Cela a simplifié les types de nœuds que vous rencontrerez pour vous concentrer sur les tâches fondamentales de visite et d'accès aux nœuds dans une arborescence d'expression.

Tout d'abord, les visiteurs gèrent uniquement les constantes qui sont des entiers. Les valeurs constantes peuvent être n'importe quel autre type numérique, et le langage C# prend en charge les conversions et les promotions entre ces types. Une version plus robuste de ce code mettrait en miroir toutes ces capacités.

Même le dernier exemple reconnaît un sous-ensemble des types de nœuds possibles. Vous pouvez toujours lui fournir de nombreuses expressions qui entraînent son échec. Une implémentation complète est fournie dans .NET Standard sous le nom ExpressionVisitor. Elle peut prendre en charge tous les types de nœud possibles.

Pour finir, la bibliothèque utilisée dans cet article a été créée à des fins de démonstration et de formation. Elle n'est pas optimisée. Cela clarifie les structures utilisées, et met en évidence les techniques employées pour visiter les nœuds et analyser ce qu'ils contiennent.

Même avec ces limitations, vous devriez être sur la bonne voie pour écrire des algorithmes qui lisent et comprennent les arborescences d'expressions.

Créer des arborescences d'expression

Article • 13/03/2023

Le compilateur C# a créé toutes les arborescences d'expressions que vous avez vues jusqu'à présent. Vous avez créé une expression lambda affectée à une variable typée en tant que `Expression<Func<T>>` ou un type similaire. Dans de nombreux scénarios, vous générez une expression en mémoire au moment de l'exécution.

Les arborescences d'expression sont immuables. Cela signifie que vous devez générer l'arborescence à partir des feuilles jusqu'à la racine. Les API que vous utilisez pour générer des arborescences d'expressions reflètent cet état de fait : les méthodes que vous utilisez pour générer un nœud prennent tous ses enfants en tant qu'arguments. Examinons quelques exemples illustrant les techniques à appliquer.

Créer des nœuds

Vous commencez par l'expression d'addition avec laquelle vous avez travaillé tout au long de ces sections :

C#

```
Expression<Func<int>> sum = () => 1 + 2;
```

Pour construire cette arborescence d'expressions, nous construisons d'abord les nœuds terminaux. Les nœuds feuilles sont des constantes. Utilisez la méthode Constant pour créer les nœuds :

C#

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
```

Ensuite, générez l'expression d'addition :

C#

```
var addition = Expression.Add(one, two);
```

Après cela, nous créons l'expression lambda :

C#

```
var lambda = Expression.Lambda(addition);
```

Cette expression lambda ne contient aucun argument. Plus loin dans cette section, nous voyons comment mapper des arguments à des paramètres, et comment générer des expressions plus complexes.

Pour les expressions comme celle-ci, nous pouvons combiner tous les appels en une seule instruction :

C#

```
var lambda2 = Expression.Lambda(
    Expression.Add(
        Expression.Constant(1, typeof(int)),
        Expression.Constant(2, typeof(int))
    )
);
```

Créer une arborescence

La section précédente a montré les principes de base de la création d'une arborescence d'expression en mémoire. Les arborescences plus complexes signifient généralement qu'il y a davantage de types de nœuds et davantage de nœuds dans l'arborescence. Examinons un autre exemple et voyons deux autres types de nœuds que vous générez souvent lors de la création d'arborescences d'expressions : les nœuds d'argument et les nœuds d'appel de méthode. Commençons par générer une arborescence d'expressions pour créer cette expression :

C#

```
Expression<Func<double, double, double>> distanceCalc =
    (x, y) => Math.Sqrt(x * x + y * y);
```

Nous commençons par créer des expressions de paramètre pour `x` et `y` :

C#

```
var xParameter = Expression.Parameter(typeof(double), "x");
var yParameter = Expression.Parameter(typeof(double), "y");
```

La création des expressions de multiplication et d'addition suit le modèle que nous avons déjà vu :

C#

```
var xSquared = Expression.Multiply(xParameter, xParameter);
var ySquared = Expression.Multiply(yParameter, yParameter);
var sum = Expression.Add(xSquared, ySquared);
```

Ensuite, nous devons créer une expression d'appel de méthode pour l'appel à `Math.Sqrt`.

C#

```
var sqrtMethod = typeof(Math).GetMethod("Sqrt", new[] { typeof(double) }) ??
throw new InvalidOperationException("Math.Sqrt not found!");
var distance = Expression.Call(sqrtMethod, sum);
```

L'appel `GetMethod` peut retourner `null` si la méthode est introuvable. Probablement parce que vous avez mal orthographié le nom de la méthode. Sinon, cela peut signifier que l'assembly requis n'est pas chargé. Enfin, placez l'appel de méthode dans une expression lambda et veillez à définir les arguments de l'expression lambda :

C#

```
var distanceLambda = Expression.Lambda(
    distance,
    xParameter,
    yParameter);
```

Dans cet exemple plus complexe, vous pouvez observer quelques techniques supplémentaires dont vous avez souvent besoin pour créer des arborescences d'expressions.

Tout d'abord, vous devez créer les objets qui représentent des variables locales ou des paramètres avant de les utiliser. Une fois que vous avez créé ces objets, vous pouvez les utiliser dans votre arborescence d'expressions partout où vous en avez besoin.

Ensuite, vous devez utiliser un sous-ensemble des API de réflexion pour créer un objet System.Reflection.MethodInfo afin de pouvoir créer une arborescence d'expressions pour accéder à cette méthode. Vous devez vous limiter au sous-ensemble des API de réflexion disponibles sur la plateforme .NET Core. Là encore, ces techniques s'étendent à d'autres arborescences d'expressions.

Générer du code en profondeur

Vous n'êtes pas limité dans ce que vous pouvez générer à l'aide de ces API. Toutefois, plus l'arborescence d'expressions que vous souhaitez générer est compliquée, plus le code est difficile à gérer et à lire.

Nous allons générer une arborescence d'expressions qui est l'équivalent de ce code :

C#

```
Func<int, int> factorialFunc = (n) =>
{
    var res = 1;
    while (n > 1)
    {
        res = res * n;
        n--;
    }
    return res;
};
```

Le code précédent n'a pas généré l'arborescence d'expression, mais simplement le délégué. À l'aide de la classe `Expression`, vous ne pouvez pas créer de lambda-instructions. Voici le code nécessaire pour générer la même fonctionnalité. Il n'existe pas d'API pour générer une boucle `while`. Au lieu de cela, vous devez générer une boucle qui contient un test conditionnel, et une cible d'étiquette pour quitter la boucle.

C#

```
var nArgument = Expression.Parameter(typeof(int), "n");
var result = Expression.Variable(typeof(int), "result");

// Creating a label that represents the return value
LabelTarget label = Expression.Label(typeof(int));

var initializeResult = Expression.Assign(result, Expression.Constant(1));

// This is the inner block that performs the multiplication,
// and decrements the value of 'n'
var block = Expression.Block(
    Expression.Assign(result,
        Expression.Multiply(result, nArgument)),
    Expression.PostDecrementAssign(nArgument)
);

// Creating a method body.
BlockExpression body = Expression.Block(
    new[] { result },
    initializeResult,
    Expression.Loop(
        Expression.IfThenElse(
            Expression.GreaterThan(nArgument, Expression.Constant(1)),
```

```
            block,
            Expression.Break(label, result)
        ),
        label
    )
);
```

Le code de génération de l'arborescence d'expressions pour la fonction factorielle est un peu plus long, plus compliqué et truffé d'étiquettes, d'instructions break et d'autres éléments que vous souhaitez éviter lors de vos tâches quotidiennes de codage.

Dans cette section, vous avez écrit du code pour visiter tous les nœuds de cette arborescence d'expressions et écrire des informations sur les nœuds créés dans cet exemple. Vous pouvez afficher ou télécharger l'exemple de code à partir du dépôt GitHub dotnet/docs. Essayez par vous-même en créant et en exécutant les exemples.

Mapper des constructions de code à des expressions

L'exemple de code suivant montre comment créer une arborescence d'expressions qui représente l'expression lambda `num => num < 5` à l'aide de l'API.

C#

```
// Manually build the expression tree for
// the lambda expression num => num < 5.
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
ConstantExpression five = Expression.Constant(5, typeof(int));
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
Expression<Func<int, bool>> lambda1 =
    Expression.Lambda<Func<int, bool>>(
        numLessThanFive,
        new ParameterExpression[] { numParam });
```

L'API de l'arborescence d'expressions prend également en charge les affectations et les expressions de flux de contrôle, comme les boucles, les blocs conditionnels et les blocs `try-catch`. À l'aide de l'API, vous pouvez créer des arborescences d'expressions qui sont plus complexes que celles qui peuvent être créées à partir d'expressions lambda par le compilateur C#. L'exemple suivant montre comment créer une arborescence d'expressions qui calcule la factorielle d'un nombre.

C#

```
// Creating a parameter expression.
ParameterExpression value = Expression.Parameter(typeof(int), "value");
```

```
// Creating an expression to hold a local variable.
ParameterExpression result = Expression.Parameter(typeof(int), "result");

// Creating a label to jump to from a loop.
LabelTarget label = Expression.Label(typeof(int));

// Creating a method body.
BlockExpression block = Expression.Block(
    // Adding a local variable.
    new[] { result },
    // Assigning a constant to a local variable: result = 1
    Expression.Assign(result, Expression.Constant(1)),
        // Adding a loop.
        Expression.Loop(
           // Adding a conditional block into the loop.
           Expression.IfThenElse(
               // Condition: value > 1
               Expression.GreaterThan(value, Expression.Constant(1)),
               // If true: result *= value --
               Expression.MultiplyAssign(result,
                   Expression.PostDecrementAssign(value)),
               // If false, exit the loop and go to the label.
               Expression.Break(label, result)
           ),
       // Label to jump to.
       label
    )
);

// Compile and execute an expression tree.
int factorial = Expression.Lambda<Func<int, int>>(block, value).Compile()
(5);

Console.WriteLine(factorial);
// Prints 120.
```

Pour plus d'informations, consultez Génération de méthodes dynamiques avec des arborescences d'expressions dans Visual Studio 2010 (ou version ultérieure).

Traduire des arborescences d'expressions

Article • 13/03/2023

Dans cet article, découvrez comment visiter chaque nœud dans une arborescence d'expressions lors de la génération d'une copie modifiée de cette arborescence d'expressions. Vous traduisez les arborescences d'expressions pour comprendre les algorithmes exprimés pour pouvoir traduire dans un autre environnement. Vous modifiez l'algorithme qui a été créé. Vous pouvez ajouter la journalisation, intercepter des appels de méthode et les suivre, entre autres.

Le code que vous générez pour traduire une arborescence d'expressions est une extension de ce que vous avez déjà vu pour visiter tous les nœuds dans une arborescence. Quand vous traduisez une arborescence d'expressions, vous visitez tous les nœuds et, ce faisant, vous générez la nouvelle arborescence. Celle-ci peut contenir des références aux nœuds d'origine ou de nouveaux nœuds que vous avez placés dans l'arborescence.

Nous allons visiter une arborescence d'expressions et en créant une nouvelle arborescence avec certains nœuds de substitution. Dans cet exemple, nous allons remplacer une constante par une autre 10 fois plus grande. Pour le reste, vous laisserez l'arborescence d'expressions intacte. Au lieu de lire la valeur de la constante et de la remplacer par une nouvelle constante, vous allez remplacer le nœud de constante par un nouveau nœud qui effectue la multiplication.

Ici, une fois que vous avez trouvé un nœud de constante, vous créez un nœud de multiplication dont les enfants sont la constante d'origine, et la constante `10` :

C#

```
private static Expression ReplaceNodes(Expression original)
{
    if (original.NodeType == ExpressionType.Constant)
    {
        return Expression.Multiply(original, Expression.Constant(10));
    }
    else if (original.NodeType == ExpressionType.Add)
    {
        var binaryExpression = (BinaryExpression)original;
        return Expression.Add(
            ReplaceNodes(binaryExpression.Left),
            ReplaceNodes(binaryExpression.Right));
    }
```

```
    return original;
}
```

Créez une arborescence en remplaçant le nœud d'origine par le substitut. Vous pouvez vérifier les changements en compilant et en exécutant l'arborescence remplacée.

C#

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var addition = Expression.Add(one, two);
var sum = ReplaceNodes(addition);
var executableFunc = Expression.Lambda(sum);

var func = (Func<int>)executableFunc.Compile();
var answer = func();
Console.WriteLine(answer);
```

Pour générer une nouvelle arborescence, il faut visiter les nœuds dans l'arborescence existante, créer de nouveaux nœuds et les insérer dans l'arborescence. L'exemple précédent montre l'importance du caractère immuable des arborescences d'expressions. Notez que la nouvelle arborescence créée dans le code précédent contient une combinaison de nœuds nouvellement créés et de nœuds de l'arborescence existante. Les nœuds peuvent être utilisés dans les deux arborescences, car les nœuds de l'arborescence existante ne peuvent pas être modifiés. La réutilisation des nœuds entraîne une amélioration significative de l'efficacité de la mémoire. Les mêmes nœuds sont utilisables dans toute une arborescence, ou dans plusieurs arborescences d'expressions. Les nœuds ne pouvant pas être modifiés, le même nœud peut être réutilisé chaque fois qu'il est nécessaire.

Parcours et exécution d'une addition

Nous allons vérifier cette nouvelle arborescence en générant un deuxième visiteur qui parcourt l'arborescence de nœuds d'addition et calcule le résultat. Apportez quelques modifications au visiteur que nous avons vu jusqu'ici. Dans cette nouvelle version, le visiteur retourne la somme partielle de l'opération d'addition jusqu'à ce point. Pour une expression constante, il s'agit simplement de la valeur de l'expression constante. Pour une expression d'addition, le résultat est la somme des opérandes gauche et droit, une fois ces arborescences parcourues.

C#

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
```

```
var three = Expression.Constant(3, typeof(int));
var four = Expression.Constant(4, typeof(int));
var addition = Expression.Add(one, two);
var add2 = Expression.Add(three, four);
var sum = Expression.Add(addition, add2);

// Declare the delegate, so you can call it
// from itself recursively:
Func<Expression, int> aggregate = null!;
// Aggregate, return constants, or the sum of the left and right operand.
// Major simplification: Assume every binary expression is an addition.
aggregate = (exp) =>
    exp.NodeType == ExpressionType.Constant ?
    (int)((ConstantExpression)exp).Value :
    aggregate(((BinaryExpression)exp).Left) +
aggregate(((BinaryExpression)exp).Right);

var theSum = aggregate(sum);
Console.WriteLine(theSum);
```

Il y a pas mal de code ici, mais les concepts sont abordables. Ce code visite les enfants avec une recherche en profondeur. Quand il rencontre un nœud de constante, le visiteur retourne la valeur de la constante. Une fois que le visiteur a visité les deux enfants, ceux-ci ont calculé la somme calculée pour cette sous-arborescence. Le nœud d'addition peut maintenant calculer sa somme. Une fois que tous les nœuds dans l'arborescence d'expressions ont été visités, la somme a été calculée. Vous pouvez suivre l'exécution en exécutant l'exemple dans le débogueur.

Nous allons simplifier le suivi de l'analyse des nœuds et du calcul de la somme en parcourant l'arborescence. Voici une version mise à jour de la méthode Aggregate qui comprend des informations de suivi :

C#

```
private static int Aggregate(Expression exp)
{
    if (exp.NodeType == ExpressionType.Constant)
    {
        var constantExp = (ConstantExpression)exp;
        Console.Error.WriteLine($"Found Constant: {constantExp.Value}");
        if (constantExp.Value is int value)
        {
            return value;
        }
        else
        {
            return 0;
        }
    }
    else if (exp.NodeType == ExpressionType.Add)
```

```
    {
        var addExp = (BinaryExpression)exp;
        Console.Error.WriteLine("Found Addition Expression");
        Console.Error.WriteLine("Computing Left node");
        var leftOperand = Aggregate(addExp.Left);
        Console.Error.WriteLine($"Left is: {leftOperand}");
        Console.Error.WriteLine("Computing Right node");
        var rightOperand = Aggregate(addExp.Right);
        Console.Error.WriteLine($"Right is: {rightOperand}");
        var sum = leftOperand + rightOperand;
        Console.Error.WriteLine($"Computed sum: {sum}");
        return sum;
    }
    else throw new NotSupportedException("Haven't written this yet");
}
```

Son exécution sur l'expression `sum` génère la sortie suivante :

Sortie

```
10
Found Addition Expression
Computing Left node
Found Addition Expression
Computing Left node
Found Constant: 1
Left is: 1
Computing Right node
Found Constant: 2
Right is: 2
Computed sum: 3
Left is: 3
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 3
Left is: 3
Computing Right node
Found Constant: 4
Right is: 4
Computed sum: 7
Right is: 7
Computed sum: 10
10
```

Suivez la sortie et suivez le code précédent. Vous devriez pouvoir déterminer comment le code visite chaque nœud et calcule la somme à mesure qu'il parcourt l'arborescence et trouve la somme.

À présent, jetons un œil à une autre exécution, avec l'expression donnée par `sum1` :

C#

```
Expression<Func<int>> sum1 = () => 1 + (2 + (3 + 4));
```

Voici la sortie de l'examen de cette expression :

Sortie

```
Found Addition Expression
Computing Left node
Found Constant: 1
Left is: 1
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 2
Left is: 2
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 3
Left is: 3
Computing Right node
Found Constant: 4
Right is: 4
Computed sum: 7
Right is: 7
Computed sum: 9
Right is: 9
Computed sum: 10
10
```

Bien que la réponse finale soit la même, le parcours d'arborescence est différent. Les nœuds sont parcourus dans un ordre différent, car l'arborescence a été construite avec différentes opérations survenant en premier.

Créer une copie modifiée

Créez un projet **Application console**. Ajoutez une directive `using` au fichier pour l'espace de noms `System.Linq.Expressions`. Ajoutez la classe `AndAlsoModifier` à votre projet.

C#

```
public class AndAlsoModifier : ExpressionVisitor
{
    public Expression Modify(Expression expression)
    {
```

```
        return Visit(expression);
    }

    protected override Expression VisitBinary(BinaryExpression b)
    {
        if (b.NodeType == ExpressionType.AndAlso)
        {
            Expression left = this.Visit(b.Left);
            Expression right = this.Visit(b.Right);

            // Make this binary expression an OrElse operation instead of an
AndAlso operation.
            return Expression.MakeBinary(ExpressionType.OrElse, left, right,
b.IsLiftedToNull, b.Method);
        }

        return base.VisitBinary(b);
    }
}
```

Cette classe hérite de la classe `AND` et est spécialisée pour modifier les expressions qui représentent des opérations ExpressionVisitor conditionnelles. Elle convertit ces opérations d'un `AND` conditionnel en un `OR` conditionnel. La classe substitue la méthode VisitBinary du type de base, car les expressions `AND` conditionnelles sont représentées sous forme d'expressions binaires. Dans la méthode `VisitBinary`, si l'expression passée représente une opération `AND` conditionnelle, le code construit une nouvelle expression qui contient l'opérateur `OR` conditionnel au lieu de l'opérateur `AND` conditionnel. Si l'expression passée à `VisitBinary` ne représente pas une opération `AND` conditionnelle, la méthode emploie l'implémentation de classe de base. Les méthodes de classe de base construisent des nœuds qui sont comme les arborescences d'expressions passées, mais les nœuds voient leurs sous-arborescences remplacées par les arborescences d'expressions produites de manière récursive par le visiteur.

Ajoutez une directive `using` au fichier pour l'espace de noms `System.Linq.Expressions`. Ajoutez le code à la méthode `Main` dans le fichier Program.cs pour créer une arborescence d'expressions, et passez-le à la méthode qui le modifie.

C#

```
Expression<Func<string, bool>> expr = name => name.Length > 10 &&
name.StartsWith("G");
Console.WriteLine(expr);

AndAlsoModifier treeModifier = new AndAlsoModifier();
Expression modifiedExpr = treeModifier.Modify((Expression)expr);

Console.WriteLine(modifiedExpr);
```

```
/*  This code produces the following output:

    name => ((name.Length > 10) && name.StartsWith("G"))
    name => ((name.Length > 10) || name.StartsWith("G"))
*/
```

Le code crée une expression qui contient une opération `AND` conditionnelle. Il crée ensuite une instance de la classe `AndAlsoModifier` et passe l'expression à la méthode `Modify` de cette classe. Les arborescences d'expressions d'origine et modifiée sont toutes deux générées pour montrer la modification. Compilez et exécutez l'application.

En savoir plus

Cet exemple montre un petit sous-ensemble du code que vous créeriez pour parcourir et interpréter les algorithmes représentés par une arborescence d'expressions. Pour des informations sur la génération d'une bibliothèque à usage général qui traduit des arborescences d'expressions dans un autre langage, consultez cette série de Matt Warren. Elle décrit en détail comment traduire le code que vous pourriez rencontrer dans une arborescence d'expressions.

Cet article aura mis en évidence toute la puissance des arborescences d'expressions. Vous examinez un ensemble de code, y apportez les modifications souhaitées et exécutez la version modifiée. Les arborescences d'expressions étant immuables, vous créez des arborescences à l'aide de composants d'arborescences existantes. La réutilisation des nœuds réduit la quantité de mémoire nécessaire pour créer des arborescences d'expressions.

Expression Trees

Article • 03/09/2023

Expression trees represent code in a tree-like data structure, where each node is an expression, for example, a method call or a binary operation such as `x < y`.

If you have used LINQ, you have experience with a rich library where the `Func` types are part of the API set. (If you aren't familiar with LINQ, you probably want to read the LINQ tutorial and the article about lambda expressions before this one.) Expression Trees provide richer interaction with the arguments that are functions.

You write function arguments, typically using Lambda Expressions, when you create LINQ queries. In a typical LINQ query, those function arguments are transformed into a delegate the compiler creates.

You've likely already written code that uses Expression trees. Entity Framework's LINQ APIs accept Expression trees as the arguments for the LINQ Query Expression Pattern. That enables Entity Framework to translate the query you wrote in C# into SQL that executes in the database engine. Another example is Moq, which is a popular mocking framework for .NET.

When you want to have a richer interaction, you need to use *Expression Trees*. Expression Trees represent code as a structure that you examine, modify, or execute. These tools give you the power to manipulate code during run time. You write code that examines running algorithms, or injects new capabilities. In more advanced scenarios, you modify running algorithms and even translate C# expressions into another form for execution in another environment.

You compile and run code represented by expression trees. Building and running expression trees enables dynamic modification of executable code, the execution of LINQ queries in various databases, and the creation of dynamic queries. For more information about expression trees in LINQ, see How to use expression trees to build dynamic queries.

Expression trees are also used in the dynamic language runtime (DLR) to provide interoperability between dynamic languages and .NET and to enable compiler writers to emit expression trees instead of Microsoft intermediate language (MSIL). For more information about the DLR, see Dynamic Language Runtime Overview.

You can have the C# or Visual Basic compiler create an expression tree for you based on an anonymous lambda expression, or you can create expression trees manually by using the System.Linq.Expressions namespace.

When a lambda expression is assigned to a variable of type Expression<TDelegate>, the compiler emits code to build an expression tree that represents the lambda expression.

The C# compiler generates expression trees only from expression lambdas (or single-line lambdas). It can't parse statement lambdas (or multi-line lambdas). For more information about lambda expressions in C#, see Lambda Expressions.

The following code examples demonstrate how to have the C# compiler create an expression tree that represents the lambda expression `num => num < 5`.

C#

```
Expression<Func<int, bool>> lambda = num => num < 5;
```

You create expression trees in your code. You build the tree by creating each node and attaching the nodes into a tree structure. You learn how to create expressions in the article on building expression trees.

Expression trees are immutable. If you want to modify an expression tree, you must construct a new expression tree by copying the existing one and replacing nodes in it. You use an expression tree visitor to traverse the existing expression tree. For more information, see the article on translating expression trees.

Once you build an expression tree, you execute the code represented by the expression tree.

Limitations

There are some newer C# language elements that don't translate well into expression trees. Expression trees can't contain `await` expressions, or `async` lambda expressions. Many of the features added in C# 6 and later don't appear exactly as written in expression trees. Instead, newer features are exposed in expression trees in the equivalent, earlier syntax, where possible. Other constructs aren't available. It means that code that interprets expression trees works the same when new language features are introduced. However, even with these limitations, expression trees do enable you to create dynamic algorithms that rely on interpreting and modifying code that is represented as a data structure. It enables rich libraries such as Entity Framework to accomplish what they do.

Expression trees won't support new expression node types. It would be a breaking change for all libraries interpreting expression trees to introduce new node types. The following list includes most C# language elements that can't be used:

- Conditional methods that have been removed
- base access
- Method group expressions, including *address-of* (&) a method group, and anonymous method expressions
- References to local functions
- Statements, including assignment (`=`) and statement bodied expressions
- Partial methods with only a defining declaration
- Unsafe pointer operations
- dynamic operations
- Coalescing operators with null or default literal left side, null coalescing assignment, and the null propagating operator (?.)
- Multi-dimensional array initializers, indexed properties, and dictionary initializers
- throw expressions
- Accessing static virtual or abstract interface members
- Lambda expressions that have attributes
- Interpolated strings
- UTF-8 string conversions or UTF-8 string literals
- Method invocations using variable arguments, named arguments or optional arguments
- Expressions using System.Index or System.Range, index "from end" (^) operator or range expressions (..)
- async lambda expressions or await expressions, including await foreach and await using
- Tuple literals, tuple conversions, tuple == or !=, or with expressions
- Discards (_), deconstructing assignment, pattern matching is operator or the pattern matching switch expression
- COM call with `ref` omitted on the arguments
- ref, in or out parameters, `ref` return values, `out` arguments, or any values of ref struct type

Interoperability Overview

Article • 02/25/2023

Interoperability enables you to preserve and take advantage of existing investments in unmanaged code. Code that runs under the control of the common language runtime (CLR) is *managed code*, and code that runs outside the CLR is *unmanaged code*. COM, COM+, C++ components, ActiveX components, and Microsoft Windows API are examples of unmanaged code.

.NET enables interoperability with unmanaged code through platform invoke services, the System.Runtime.InteropServices namespace, C++ interoperability, and COM interoperability (COM interop).

Platform Invoke

Platform invoke is a service that enables managed code to call unmanaged functions implemented in dynamic link libraries (DLLs), such as the Microsoft Windows API. It locates and invokes an exported function and marshals its arguments (integers, strings, arrays, structures, and so on) across the interoperation boundary as needed.

For more information, see Consuming Unmanaged DLL Functions and How to use platform invoke to play a WAV file.

> **Note**
>
> The **Common Language Runtime** (CLR) manages access to system resources. Calling unmanaged code that is outside the CLR bypasses this security mechanism, and therefore presents a security risk. For example, unmanaged code might call resources in unmanaged code directly, bypassing CLR security mechanisms. For more information, see **Security in .NET**.

C++ Interop

You can use C++ interop, also known as It Just Works (IJW), to wrap a native C++ class. C++ interop enables code authored in C# or another .NET language to access it. You write C++ code to wrap a native DLL or COM component. Unlike other .NET languages, Visual C++ has interoperability support that enables managed and unmanaged code in the same application and even in the same file. You then build the C++ code by using the **/clr** compiler switch to produce a managed assembly. Finally, you add a reference to

the assembly in your C# project and use the wrapped objects just as you would use other managed classes.

Exposing COM Components to C#

You can consume a COM component from a C# project. The general steps are as follows:

1. Locate a COM component to use and register it. Use regsvr32.exe to register or un–register a COM DLL.
2. Add to the project a reference to the COM component or type library. When you add the reference, Visual Studio uses the Tlbimp.exe (Type Library Importer), which takes a type library as input, to output a .NET interop assembly. The assembly, also named a runtime callable wrapper (RCW), contains managed classes and interfaces that wrap the COM classes and interfaces that are in the type library. Visual Studio adds to the project a reference to the generated assembly.
3. Create an instance of a class defined in the RCW. Creating an instance of that class creates an instance of the COM object.
4. Use the object just as you use other managed objects. When the object is reclaimed by garbage collection, the instance of the COM object is also released from memory.

For more information, see Exposing COM Components to the .NET Framework.

Exposing C# to COM

COM clients can consume C# types that have been correctly exposed. The basic steps to expose C# types are as follows:

1. Add interop attributes in the C# project. You can make an assembly COM visible by modifying C# project properties. For more information, see Assembly Information Dialog Box.
2. Generate a COM type library and register it for COM usage. You can modify C# project properties to automatically register the C# assembly for COM interop. Visual Studio uses the Regasm.exe (Assembly Registration Tool), using the `/tlb` command-line switch, which takes a managed assembly as input, to generate a type library. This type library describes the `public` types in the assembly and adds registry entries so that COM clients can create managed classes.

For more information, see Exposing .NET Framework Components to COM and Example COM Class.

See also

- Improving Interop Performance
- Introduction to Interoperability between COM and .NET
- Introduction to COM Interop in Visual Basic
- Marshaling between Managed and Unmanaged Code
- Interoperating with Unmanaged Code

Gestion de version en C#

Article • 09/05/2023

Dans ce didacticiel, vous allez apprendre la signification de la gestion de version .NET. Vous découvrirez également les facteurs à prendre en compte dans le cadre du contrôle de version de votre bibliothèque et de sa mise à niveau vers une nouvelle version.

Création de bibliothèques

En tant que développeur ayant créé des bibliothèques .NET pour une utilisation publique, vous avez probablement été confronté à des situations où vous deviez déployer de nouvelles mises à jour. Votre manière de procéder est très importante, car vous devez assurer une transition fluide du code existant vers la nouvelle version de votre bibliothèque. Voici quelques points à prendre en compte lors de la création d'une version :

Gestion sémantique des versions

La gestion sémantique de version (SemVer en abrégé) est une convention d'affectation de noms appliquée à des versions de votre bibliothèque pour indiquer des événements jalons spécifiques. Dans l'idéal, les informations de version ajoutées à votre bibliothèque doivent aider les développeurs à déterminer la compatibilité avec les projets qui utilisent des versions antérieures de cette même bibliothèque.

L'approche la plus simple de SemVer est le format à 3 composants `MAJOR.MINOR.PATCH`, où :

- `MAJOR` est incrémenté quand vous apportez des changements d'API incompatibles
- `MINOR` est incrémenté quand vous ajoutez des fonctionnalités à compatibilité descendante
- `PATCH` est incrémenté quand vous apportez des correctifs de bogues à compatibilité descendante

Il existe également des moyens de spécifier d'autres scénarios, comme des préversions, lors de l'application des informations de version à votre bibliothèque .NET.

Compatibilité descendante

Quand vous publiez de nouvelles versions de votre bibliothèque, l'une de vos principales préoccupations concerne la compatibilité descendante avec les versions

précédentes. Une nouvelle version de la bibliothèque est compatible au format source avec une version précédente si le code qui dépend de la version précédente est utilisable avec la nouvelle version une fois qu'il est recompilé. Une nouvelle version de la bibliothèque est compatible au format binaire si une application qui dépendait de l'ancienne version est utilisable avec la nouvelle version sans recompilation.

Voici quelques éléments à prendre en compte quand vous tentez de maintenir la compatibilité descendante avec d'anciennes versions de votre bibliothèque :

- Méthodes virtuelles : quand vous rendez une méthode virtuelle non virtuelle dans la nouvelle version, cela signifie que les projets qui substituent cette méthode doivent être mis à jour. Il s'agit là d'une modification avec rupture importante qui est vivement déconseillée.
- Signatures de méthode : quand la mise à jour d'un comportement de méthode vous oblige à modifier également sa signature, vous devez créer à la place une surcharge afin que le code appelant cette méthode fonctionne toujours. Vous pouvez toujours manipuler l'ancienne signature de méthode pour appeler la nouvelle signature de méthode afin que l'implémentation reste cohérente.
- Attribut obsolète : vous pouvez utiliser cet attribut dans votre code pour spécifier des classes ou membres de classe dépréciés et susceptibles d'être supprimés dans les prochaines versions. Cela garantit que les développeurs utilisant votre bibliothèque sont mieux préparés pour les modifications avec rupture.
- Arguments de méthode facultatifs : quand vous rendez obligatoires des arguments de méthode auparavant facultatifs ou modifiez leur valeur par défaut, tout le code qui ne fournit pas ces arguments doit être mis à jour.

> ⓘ **Notes**
>
> Le fait de rendre obligatoires des arguments facultatifs doit avoir très peu d'effet en particulier si le comportement de la méthode ne change pas.

Plus il est facile pour vos utilisateurs d'effectuer la mise à niveau vers la nouvelle version de votre bibliothèque, plus il est probable qu'ils le feront rapidement.

Fichier de configuration de l'application

En tant que développeur .NET, il est très probable que vous ayez rencontré le fichier app.config, présent dans la plupart des types de projets. Ce simple fichier de configuration peut s'avérer très utile pour améliorer le déploiement de nouvelles mises à jour. Vous devez généralement concevoir vos bibliothèques de sorte que les informations qui sont susceptibles de changer régulièrement soient stockées dans le

fichier `app.config` ; ainsi, quand ces informations sont mises à jour, le fichier de configuration des anciennes versions doit uniquement être remplacé par le nouveau sans que la recompilation de la bibliothèque ne soit nécessaire.

Utilisation des bibliothèques

En tant que développeur qui utilise les bibliothèques .NET générées par d'autres développeurs, vous savez probablement qu'une nouvelle version d'une bibliothèque peut ne pas être totalement compatible avec votre projet et vous pouvez souvent être obligé de mettre à jour votre code pour vous adapter à ces modifications.

Heureusement pour vous, C# et l'écosystème .NET sont fournis avec des fonctionnalités et techniques qui nous permettent de mettre facilement à jour notre application pour utiliser les nouvelles versions de bibliothèques qui peuvent présenter des modifications avec rupture.

Redirection des liaisons d'assembly

Vous pouvez utiliser le fichier *app.config* pour mettre à jour la version d'une bibliothèque que votre application emploie. En ajoutant un élément nommé *redirection de liaison*, vous pouvez utiliser la nouvelle version de bibliothèque sans avoir à recompiler votre application. L'exemple suivant montre comment vous devez mettre à jour le fichier *app.config* de votre application pour utiliser la version corrective `1.0.1` de `ReferencedLibrary` au lieu de la version `1.0.0` avec laquelle il a été compilé à l'origine.

XML

```xml
<dependentAssembly>
    <assemblyIdentity name="ReferencedLibrary"
publicKeyToken="32ab4ba45e0a69a1" culture="en-us" />
    <bindingRedirect oldVersion="1.0.0" newVersion="1.0.1" />
</dependentAssembly>
```

> ⓘ **Notes**
>
> Cette approche fonctionne uniquement si la nouvelle version de `ReferencedLibrary` est compatible au format binaire avec votre application. Consultez la section **Compatibilité descendante** ci-dessus pour connaître les changements à rechercher lors de la détermination de la compatibilité.

new

Vous utilisez le modificateur `new` pour masquer les membres hérités d'une classe de base. Il s'agit d'une façon pour les classes dérivées de pouvoir répondre aux mises à jour dans les classes de base.

Prenons l'exemple suivant :

C#

```
public class BaseClass
{
    public void MyMethod()
    {
        Console.WriteLine("A base method");
    }
}

public class DerivedClass : BaseClass
{
    public new void MyMethod()
    {
        Console.WriteLine("A derived method");
    }
}

public static void Main()
{
    BaseClass b = new BaseClass();
    DerivedClass d = new DerivedClass();

    b.MyMethod();
    d.MyMethod();
}
```

Sortie

Console

```
A base method
A derived method
```

Dans l'exemple ci-dessus, vous pouvez voir comment `DerivedClass` masque la méthode `MyMethod` présente dans `BaseClass`. Cela signifie que, quand une classe de base dans la nouvelle version d'une bibliothèque ajoute un membre qui existe déjà dans votre classe dérivée, vous pouvez simplement utiliser le modificateur `new` sur le membre de la classe dérivée pour masquer le membre de la classe de base.

Quand aucun modificateur `new` n'est spécifié, une classe dérivée masque par défaut des membres en conflit dans une classe de base ; même si un avertissement du compilateur est généré, le code est toujours compilé. Cela signifie que le fait d'ajouter simplement de nouveaux membres à une classe existante rend cette nouvelle version de votre bibliothèque compatible au format source et binaire avec le code qui en dépend.

override

Le modificateur `override` indique qu'une implémentation dérivée étend l'implémentation d'un membre de classe de base au lieu de la masquer. Le modificateur `virtual` doit être appliqué au membre de classe de base.

C#

```
public class MyBaseClass
{
    public virtual string MethodOne()
    {
        return "Method One";
    }
}

public class MyDerivedClass : MyBaseClass
{
    public override string MethodOne()
    {
        return "Derived Method One";
    }
}

public static void Main()
{
    MyBaseClass b = new MyBaseClass();
    MyDerivedClass d = new MyDerivedClass();

    Console.WriteLine("Base Method One: {0}", b.MethodOne());
    Console.WriteLine("Derived Method One: {0}", d.MethodOne());
}
```

Sortie

Console

```
Base Method One: Method One
Derived Method One: Derived Method One
```

Le modificateur `override` est évalué au moment de la compilation et génère une erreur s'il ne trouve pas un membre virtuel à substituer.

Votre connaissance des techniques présentées et votre compréhension des situations dans lesquelles les utiliser sera très utile pour passer en douceur d'une version de bibliothèque à une autre.

Guides pratiques (C#)

Article • 23/02/2023

Dans la section Procédures du guide C#, vous trouverez rapidement des réponses à des questions courantes. Dans certains cas, les articles peuvent figurer dans plusieurs sections. Nous avons souhaité les rendre plus faciles à trouver via plusieurs voies de recherche.

Concepts C# généraux

Plusieurs conseils et astuces sont des pratiques courantes des développeurs en C# :

- Initialiser des objets à l'aide d'un initialiseur d'objet.
- Découvrir les différences entre le passage d'un struct et d'une classe à une méthode.
- Utiliser la surcharge d'opérateur.
- Implémenter et appeler une méthode d'extension personnalisée.
- Créer une méthode pour un type enum à l'aide des méthodes d'extension.

Membres de classe, d'enregistrement et de struct

Vous créez des classes, des enregistrements et des structs pour implémenter votre programme. Ces techniques sont couramment utilisées lors de l'écriture de classes, d'enregistrements ou de structs.

- Déclarer des propriétés implémentées automatiquement.
- Déclarer et utiliser des propriétés en lecture-écriture.
- Définir des constantes.
- Remplacer la méthode ToString pour fournir la sortie de chaîne.
- Définir des propriétés abstraites.
- Utiliser les fonctionnalités de la documentation XML pour documenter votre code.
- Implémenter de manière explicite des membres d'interface pour conserver votre interface publique concise.
- Implémenter de manière explicite des membres de deux interfaces.

Utilisation des collections

Ces articles vous permettent d'utiliser des collections de données.

- Initialiser un dictionnaire avec un initialiseur de collection.

Utilisation des chaînes

Les chaînes constituent le type de données fondamental utilisé pour afficher ou manipuler du texte. Ces articles présentent les pratiques courantes avec des chaînes.

- Comparer des chaînes.
- Modifier le contenu d'une chaîne.
- Déterminer si une chaîne représente un nombre.
- Utiliser String.Split pour séparer les chaînes.
- Combiner plusieurs chaînes en une seule.
- Rechercher du texte dans une chaîne.

Conversion entre types

Vous aurez peut-être besoin de convertir un objet en un autre type.

- Déterminer si une chaîne représente un nombre.
- Convertir entre des chaînes qui représentent des nombres hexadécimaux et le nombre.
- Convertir une chaîne en DateTime.
- Convertir un tableau d'octets en int.
- Convertir une chaîne en nombre .
- Utiliser les critères spéciaux et les opérateurs as et is pour caster sans risque en un autre type.
- Définir des conversions de type personnalisé.
- Déterminer si un type est un type valeur nullable.
- Convertir entre des types valeur nullable et non nullable.

Comparaisons d'égalité et de classement

Vous pouvez créer des types qui définissent leurs propres règles pour l'égalité ou qui définissent un classement naturel entre les objets de ce type.

- Tester l'égalité basée sur les références.
- Définir une égalité basée sur les valeurs pour un type.

Gestion des exceptions

Les programmes .NET signalent que des méthodes n'ont pas réussi à effectuer leur travail en levant des exceptions. Dans ces articles, vous allez apprendre à travailler avec les exceptions.

- Gérer les exceptions avec try et catch.
- Nettoyer les ressources avec les clauses finally.
- Récupérer suite à des exceptions non-CLS (Common Language Specification).

Délégués et événements

Les délégués et les événements fournissent une fonctionnalité pour les stratégies qui implique des blocs de code faiblement couplés.

- Déclarer, instancier et utiliser des délégués.
- Combiner des délégués multicast.

Les événements offrent un mécanisme permettant de publier des notifications ou de s'y abonner.

- S'abonner à des événements et s'en désabonner.
- Implémenter des événements déclarés dans des interfaces.
- Se conformer aux instructions .NET quand votre code publie des événements.
- Déclencher des événements définis dans les classes de base de classes dérivées.
- Implémenter des accesseurs d'événement personnalisés.

Pratiques LINQ

LINQ vous permet d'écrire du code pour interroger une source de données qui prend en charge le modèle d'expression de requête LINQ. Ces articles vous permettent de comprendre le modèle et d'utiliser différentes sources de données.

- Interroger une collection.
- Utiliser var dans des expressions de requête.
- Retourner des sous-ensembles de propriétés d'éléments d'une requête.
- Écrire des requêtes avec un filtrage complexe.
- Trier les éléments d'une source de données.
- Trier les éléments sur plusieurs clés.
- Contrôler le type d'une projection.
- Compter les occurrences d'une valeur dans une séquence source.
- Calculer des valeurs intermédiaires.
- Fusionner des données de plusieurs sources.
- Trouver la différence définie entre deux séquences.
- Déboguer des résultats de requête vides.
- Ajouter des méthodes personnalisées dans des requêtes LINQ.

Traitement de plusieurs threads et traitement asynchrone

Les programmes récents utilisent souvent des opérations asynchrones. Les articles suivants vous apprennent à utiliser ces techniques.

- Améliorer les performances asynchrones avec System.Threading.Tasks.Task.WhenAll.
- Effectuer plusieurs requêtes web en parallèle en utilisant async et await.
- Utiliser un pool de threads.

Arguments de ligne de commande dans votre programme

En règle générale, les programmes en C# ont des arguments de ligne de commande. Les articles suivants vous apprennent à accéder et à traiter ces arguments de ligne de commande.

- Récupérer tous les arguments de ligne de commande avec for.

Comment séparer des chaînes à l'aide de String.Split en C

Article • 10/05/2023

La méthode String.Split crée un tableau de sous-chaînes en fractionnant la chaîne d'entrée en fonction d'un ou plusieurs délimiteurs. Cette méthode est souvent le moyen le plus simple de séparer une chaîne sur les limites des mots. Elle sert également à fractionner des chaînes sur d'autres caractères ou chaînes spécifiques.

> **Notes**
>
> Les exemples C# de cet article s'exécutent dans l'exécuteur et le terrain de jeu du code inline Try.NET. Sélectionnez le bouton **Exécuter** pour exécuter un exemple dans une fenêtre interactive. Une fois que vous avez exécuté le code, vous pouvez le modifier et exécuter le code modifié en resélectionnant **Exécuter**. La code modifié s'exécute dans la fenêtre interactive ou, si la compilation échoue, la fenêtre interactive affiche tous les messages d'erreur du compilateur C#.

Le code suivant fractionne une expression commune en un tableau de chaînes pour chaque mot.

C#

```
string phrase = "The quick brown fox jumps over the lazy dog.";
string[] words = phrase.Split(' ');

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

Chaque instance d'un caractère de séparation génère une valeur dans le tableau retourné. Les caractères de séparation consécutifs produisent une chaîne vide comme valeur dans le tableau retourné. Vous pouvez voir comment une chaîne vide est créée dans l'exemple suivant, qui utilise le caractère espace comme séparateur.

C#

```
string phrase = "The quick brown    fox     jumps over the lazy dog.";
string[] words = phrase.Split(' ');

foreach (var word in words)
```

```
{
    System.Console.WriteLine($"<{word}>");
}
```

Ce comportement facilite l'utilisation de formats tels que les fichiers CSV (valeurs séparées par des virgules) représentant des données tabulaires. Les virgules consécutives représentent une colonne vide.

Vous pouvez passer un paramètre StringSplitOptions.RemoveEmptyEntries facultatif pour exclure toutes les chaînes vides dans le tableau retourné. Pour un traitement plus complexe de la collection retournée, vous pouvez utiliser LINQ pour manipuler la séquence de résultat.

String.Split peut utiliser plusieurs caractères de séparation. L'exemple suivant utilise des espaces, des virgules, des points, des deux-points et des onglets comme caractères de séparation, qui sont transmis à Split dans un tableau . La boucle en bas du code affiche chacun des mots dans le tableau retourné.

C#

```
char[] delimiterChars = { ' ', ',', '.', ':', '\t' };

string text = "one\ttwo three:four,five six seven";
System.Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(delimiterChars);
System.Console.WriteLine($"{words.Length} words in text:");

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

Les instances consécutives d'un séparateur produisent une chaîne vide dans le tableau de sortie :

C#

```
char[] delimiterChars = { ' ', ',', '.', ':', '\t' };

string text = "one\ttwo :,five six seven";
System.Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(delimiterChars);
System.Console.WriteLine($"{words.Length} words in text:");

foreach (var word in words)
{
```

```
    System.Console.WriteLine($"<{word}>");
}
```

String.Split peut accepter un tableau de chaînes (séquences de caractères, à la place de caractères uniques, qui agissent comme séparateurs lors de l'analyse de la chaîne cible).

C#

```
string[] separatingStrings = { "<<", "..." };

string text = "one<<two......three<four";
System.Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(separatingStrings,
System.StringSplitOptions.RemoveEmptyEntries);
System.Console.WriteLine($"{words.Length} substrings in text:");

foreach (var word in words)
{
    System.Console.WriteLine(word);
}
```

Voir aussi

- Extraire des éléments d'une chaîne
- Guide de programmation C#
- Chaînes
- Expressions régulières .NET

Guide pratique pour concaténer plusieurs chaînes (Guide C#)

Article • 10/05/2023

La *concaténation* consiste à ajouter une chaîne à la fin d'une autre chaîne. Vous concaténez les chaînes à l'aide de l'opérateur `+`. Pour les littéraux de chaîne et les constantes de chaîne, la concaténation se produit au moment de la compilation ; aucune concaténation ne se produit au moment de l'exécution. Pour les variables de chaîne, la concaténation se produit uniquement au moment de l'exécution.

> **Notes**
>
> Les exemples C# de cet article s'exécutent dans l'exécuteur et le terrain de jeu du code inline **Try.NET**. Sélectionnez le bouton **Exécuter** pour exécuter un exemple dans une fenêtre interactive. Une fois que vous avez exécuté le code, vous pouvez le modifier et exécuter le code modifié en resélectionnant **Exécuter**. La code modifié s'exécute dans la fenêtre interactive ou, si la compilation échoue, la fenêtre interactive affiche tous les messages d'erreur du compilateur C#.

Littéraux de chaîne

L'exemple suivant fractionne un littéral de chaîne long en chaînes plus petites pour améliorer la lisibilité dans le code source. Le code concatène les chaînes plus petites pour créer le littéral de chaîne long. Les parties sont concaténées en une seule chaîne au moment de la compilation. Il n'y a aucune incidence sur les performances d'exécution, quel que soit le nombre de chaînes impliquées.

C#

```
// Concatenation of literals is performed at compile time, not run time.
string text = "Historically, the world of data and the world of objects " +
"have not been well integrated. Programmers work in C# or Visual Basic " +
"and also in SQL or XQuery. On the one side are concepts such as classes, "
+
"objects, fields, inheritance, and .NET Framework APIs. On the other side "
+
"are tables, columns, rows, nodes, and separate languages for dealing with "
+
"them. Data types often require translation between the two worlds; there
are " +
"different standard functions. Because the object world has no notion of
query, a " +
```

```
"query can only be represented as a string without compile-time type
checking or " +
"IntelliSense support in the IDE. Transferring data from SQL tables or XML
trees to " +
"objects in memory is often tedious and error-prone.";

System.Console.WriteLine(text);
```

Opérateurs `+` et `+=`

Pour concaténer des variables de chaîne, vous pouvez utiliser les opérateurs `+` ou `+=`, l'interpolation de chaînes, ou encore les méthodes String.Format, String.Concat, String.Join ou StringBuilder.Append. L'opérateur `+` est facile à utiliser et convient au code intuitif. Même si vous utilisez plusieurs opérateurs `+` dans une instruction, le contenu de la chaîne est copié une seule fois. Le code suivant montre des exemples d'utilisation des opérateurs `+` et `+=` pour concaténer des chaînes :

C#

```
string userName = "<Type your name here>";
string dateString = DateTime.Today.ToShortDateString();

// Use the + and += operators for one-time concatenations.
string str = "Hello " + userName + ". Today is " + dateString + ".";
System.Console.WriteLine(str);

str += " How are you today?";
System.Console.WriteLine(str);
```

Interpolation de chaîne

Dans certaines expressions, il est plus facile de concaténer des chaînes à l'aide de l'interpolation de chaînes, comme le montre le code suivant :

C#

```
string userName = "<Type your name here>";
string date = DateTime.Today.ToShortDateString();

// Use string interpolation to concatenate strings.
string str = $"Hello {userName}. Today is {date}.";
System.Console.WriteLine(str);

str = $"{str} How are you today?";
System.Console.WriteLine(str);
```

> ⓘ Notes
>
> Dans les opérations de concaténation de chaîne, le compilateur C# traite une chaîne null de la même manière qu'une chaîne vide.

À partir de C# 10, vous pouvez utiliser l'interpolation de chaîne pour initialiser une chaîne constante lorsque toutes les expressions utilisées pour les espaces réservés sont également des chaînes constantes.

`String.Format`

Une autre méthode permettant de concaténer des chaînes est String.Format. Cette méthode fonctionne bien lorsque vous créez une chaîne à partir d'un petit nombre de chaînes de composants.

`StringBuilder`

Dans d'autres cas, vous pouvez combiner des chaînes dans une boucle quand vous ne connaissez pas le nombre de chaînes sources que vous combinez et que le nombre réel de chaînes sources peut se révéler très grand. La classe StringBuilder a été conçue pour ces scénarios. Le code suivant utilise la méthode Append de la classe StringBuilder pour concaténer des chaînes.

C#

```
// Use StringBuilder for concatenation in tight loops.
var sb = new System.Text.StringBuilder();
for (int i = 0; i < 20; i++)
{
    sb.AppendLine(i.ToString());
}
System.Console.WriteLine(sb.ToString());
```

Vous pouvez en savoir plus sur les raisons de choisir la concaténation de chaînes ou la classe StringBuilder.

`String.Concat` ou `String.Join`

Une autre option permettant de joindre les chaînes d'une collection consiste à utiliser la méthode String.Concat. Utilisez la méthode String.Join si les chaînes sources doivent

être séparées par un délimiteur. Le code suivant combine un tableau de mots en utilisant ces deux méthodes :

```C#
string[] words = { "The", "quick", "brown", "fox", "jumps", "over", "the",
"lazy", "dog." };

var unreadablePhrase = string.Concat(words);
System.Console.WriteLine(unreadablePhrase);

var readablePhrase = string.Join(" ", words);
System.Console.WriteLine(readablePhrase);
```

LINQ et `Enumerable.Aggregate`

Enfin, vous pouvez utiliser LINQ et la méthode Enumerable.Aggregate pour joindre les chaînes d'une collection. Cette méthode combine les chaînes sources en utilisant une expression lambda. L'expression lambda effectue le travail d'ajouter chaque chaîne à l'accumulation existante. L'exemple suivant combine un tableau de mots, en ajoutant un espace entre chaque mot du tableau :

```C#
string[] words = { "The", "quick", "brown", "fox", "jumps", "over", "the",
"lazy", "dog." };

var phrase = words.Aggregate((partialPhrase, word) =>$"{partialPhrase}
{word}");
System.Console.WriteLine(phrase);
```

Cette option peut créer plus d'allocations que d'autres méthodes pour concaténer des collections, car elle crée une chaîne intermédiaire pour chaque itération. Si l'optimisation des performances est essentielle, considérez la classe StringBuilder ou la méthode String.Concat ou String.Join pour concaténer une collection, au lieu de `Enumerable.Aggregate`.

Voir aussi

- String
- StringBuilder
- Guide de programmation C#
- Chaînes

Guide pratique pour faire des recherches dans des chaînes

Article • 10/05/2023

Vous pouvez utiliser deux stratégies principales pour rechercher du texte dans des chaînes. Les méthodes de la classe String recherchent un texte spécifique. Les expressions régulières recherchent des modèles dans du texte.

> **Notes**
>
> Les exemples C# de cet article s'exécutent dans l'exécuteur et le terrain de jeu du code inline **Try.NET**. Sélectionnez le bouton **Exécuter** pour exécuter un exemple dans une fenêtre interactive. Une fois que vous avez exécuté le code, vous pouvez le modifier et exécuter le code modifié en resélectionnant **Exécuter**. La code modifié s'exécute dans la fenêtre interactive ou, si la compilation échoue, la fenêtre interactive affiche tous les messages d'erreur du compilateur C#.

Le type string, qui est un alias de la classe System.String, propose un certain nombre de méthodes utiles pour rechercher le contenu d'une chaîne. En voici quelques-unes : Contains, StartsWith, EndsWith, IndexOf, LastIndexOf. La classe System.Text.RegularExpressions.Regex fournit un vocabulaire étendu pour rechercher des modèles dans du texte. Dans cet article, vous découvrez ces techniques et comment choisir la meilleure méthode pour vos besoins.

Une chaîne contient-elle un certain texte ?

Les méthodes String.Contains, String.StartsWith, et String.EndsWith recherchent un texte spécifique dans une chaîne. L'exemple suivant montre chacune de ces méthodes et une variante qui utilise une recherche sans respecter la casse :

C#

```
string factMessage = "Extension methods have all the capabilities of regular
static methods.";

// Write the string and include the quotation marks.
Console.WriteLine($"\"{factMessage}\"");

// Simple comparisons are always case sensitive!
bool containsSearchResult = factMessage.Contains("extension");
Console.WriteLine($"Contains \"extension\"? {containsSearchResult}");
```

```
// For user input and strings that will be displayed to the end user,
// use the StringComparison parameter on methods that have it to specify how
to match strings.
bool ignoreCaseSearchResult = factMessage.StartsWith("extension",
System.StringComparison.CurrentCultureIgnoreCase);
Console.WriteLine($"Starts with \"extension\"? {ignoreCaseSearchResult}
(ignoring case)");

bool endsWithSearchResult = factMessage.EndsWith(".",
System.StringComparison.CurrentCultureIgnoreCase);
Console.WriteLine($"Ends with '.'? {endsWithSearchResult}");
```

L'exemple précédent montre un point important de l'utilisation de ces méthodes. Par défaut, les recherches **respectent la casse**. Vous utilisez la valeur d'énumération StringComparison.CurrentCultureIgnoreCase pour spécifier une recherche qui ne respecte pas la casse.

Où le texte recherché se trouve-t-il dans une chaîne ?

Les méthodes IndexOf et LastIndexOf recherchent aussi du texte dans des chaînes. Ces méthodes retournent l'emplacement du texte recherché. Si le texte n'est pas trouvé, elles retournent `-1`. L'exemple suivant montre une recherche de la première et de la dernière occurrence du mot « methods » et affiche le texte qui se trouve entre les deux.

C#

```
string factMessage = "Extension methods have all the capabilities of regular
static methods.";

// Write the string and include the quotation marks.
Console.WriteLine($"\"{factMessage}\"");

// This search returns the substring between two strings, so
// the first index is moved to the character just after the first string.
int first = factMessage.IndexOf("methods") + "methods".Length;
int last = factMessage.LastIndexOf("methods");
string str2 = factMessage.Substring(first, last - first);
Console.WriteLine($"Substring between \"methods\" and \"methods\":
'{str2}'");
```

Recherche de texte spécifique en utilisant des expressions régulières

La classe System.Text.RegularExpressions.Regex permet d'effectuer des recherches de chaînes. Ces recherches peuvent varier en complexité et aller de modèles simples à des modèles plus compliqués.

L'exemple de code suivant recherche le mot « the » ou « their » dans une phrase, indépendamment de la casse. La méthode statique Regex.IsMatch effectue la recherche. Vous lui passez la chaîne à rechercher et un modèle de recherche. Dans ce cas, un troisième argument spécifie que la recherche est insensible à la casse. Pour plus d'informations, consultez System.Text.RegularExpressions.RegexOptions.

Le modèle de recherche décrit le texte que vous recherchez. Le tableau suivant décrit chaque élément du modèle de recherche. (Le tableau ci-dessous utilise une barre oblique inversée [`\`] qui, dans une chaîne C#, doit être précédée d'un caractère d'échappement sous la forme `\\`).

Modèle	Signification
`the`	correspond au texte « the »
`(eir)?`	correspond à 0 ou 1 occurrence de « eir »
`\s`	correspond à un espace blanc

C#

```
string[] sentences =
{
    "Put the water over there.",
    "They're quite thirsty.",
    "Their water bottles broke."
};

string sPattern = "the(ir)?\\s";

foreach (string s in sentences)
{
    Console.Write($"{s,24}");

    if (System.Text.RegularExpressions.Regex.IsMatch(s, sPattern,
System.Text.RegularExpressions.RegexOptions.IgnoreCase))
    {
        Console.WriteLine($"  (match for '{sPattern}' found)");
    }
    else
    {
        Console.WriteLine();
    }
}
```

> **Conseil**
>
> Les méthodes `string` sont généralement de meilleurs choix quand vous recherchez une chaîne précise. Les expressions régulières sont mieux adaptées quand vous recherchez un certain modèle dans une chaîne source.

Une chaîne suit-elle un modèle ?

Le code suivant utilise des expressions régulières pour valider le format de chaque chaîne d'un tableau. La validation doit vérifier que chaque chaîne a la forme d'un numéro de téléphone constitué de trois groupes de chiffres séparés par des tirets, les deux premiers groupes contenant trois chiffres et le troisième en contenant quatre. Le modèle de recherche utilise l'expression régulière `^\\d{3}-\\d{3}-\\d{4}$`. Pour plus d'informations, consultez Langage des expressions régulières - Aide-mémoire.

Modèle	Signification
`^`	correspond au début de la chaîne
`\d{3}`	correspond à exactement 3 caractères numériques
`-`	correspond au caractère « - ».
`\d{4}`	correspond à exactement 4 caractères numériques
`$`	correspond à la fin de la chaîne

C#

```
string[] numbers =
{
    "123-555-0190",
    "444-234-22450",
    "690-555-0178",
    "146-893-232",
    "146-555-0122",
    "4007-555-0111",
    "407-555-0111",
    "407-2-5555",
    "407-555-8974",
    "407-2ab-5555",
    "690-555-8148",
    "146-893-232-"
};

string sPattern = "^\\d{3}-\\d{3}-\\d{4}$";
```

```
foreach (string s in numbers)
{
    Console.Write($"{s,14}");

    if (System.Text.RegularExpressions.Regex.IsMatch(s, sPattern))
    {
        Console.WriteLine(" - valid");
    }
    else
    {
        Console.WriteLine(" - invalid");
    }
}
```

Ce seul modèle de recherche correspond à de nombreuses chaînes valides. Les expressions régulières sont mieux adaptées pour rechercher un modèle ou pour effectuer une validation par rapport à un modèle, et le sont moins pour une seule chaîne de texte.

Voir aussi

- Guide de programmation C#
- Chaînes
- LINQ et chaînes
- System.Text.RegularExpressions.Regex
- Expressions régulières .NET
- Langage des expressions régulières - Aide-mémoire
- Bonnes pratiques pour l'utilisation de chaînes dans .NET

Comment modifier du contenu de chaîne en C#

Article • 09/05/2023

Cet article présente plusieurs techniques pour produire un `string` en modifiant un `string` existant. Toutes les techniques présentées retournent le résultat des modifications sous la forme d'un nouvel objet `string`. Pour démontrer que les chaînes d'origine et modifiées sont des instances distinctes, les exemples stockent le résultat dans une nouvelle variable. Vous pouvez examiner la `string` d'origine et la nouvelle `string` modifiée lorsque vous exécutez chaque exemple.

> **Notes**
>
> Les exemples C# de cet article s'exécutent dans l'exécuteur et le terrain de jeu du code inline **Try.NET**. Sélectionnez le bouton **Exécuter** pour exécuter un exemple dans une fenêtre interactive. Une fois que vous avez exécuté le code, vous pouvez le modifier et exécuter le code modifié en resélectionnant **Exécuter**. La code modifié s'exécute dans la fenêtre interactive ou, si la compilation échoue, la fenêtre interactive affiche tous les messages d'erreur du compilateur C#.

Plusieurs techniques sont présentées dans cet article. Vous pouvez remplacer du texte existant. Vous pouvez rechercher des modèles et remplacer le texte correspondant par un autre texte. Vous pouvez traiter une chaîne en tant que séquence de caractères. Vous pouvez également utiliser des méthodes pratiques qui suppriment les espaces blancs. Choisissez les techniques les mieux adaptées à votre scénario.

Remplacer du texte

Le code suivant crée une chaîne en remplaçant le texte existant par un substitut.

C#

```
string source = "The mountains are behind the clouds today.";

// Replace one substring with another with String.Replace.
// Only exact matches are supported.
var replacement = source.Replace("mountains", "peaks");
Console.WriteLine($"The source string is <{source}>");
Console.WriteLine($"The updated string is <{replacement}>");
```

Le code précédent illustre la propriété *immuable* des chaînes. Vous pouvez voir dans l'exemple précédent que la chaîne d'origine, `source`, n'est pas modifiée. La méthode String.Replace crée un `string` contenant les modifications.

La méthode Replace peut remplacer des chaînes ou des caractères uniques. Dans les deux cas, toutes les occurrences du texte recherché sont remplacées. L'exemple suivant remplace tous les caractères ' ' par '_' :

C#

```
string source = "The mountains are behind the clouds today.";

// Replace all occurrences of one char with another.
var replacement = source.Replace(' ', '_');
Console.WriteLine(source);
Console.WriteLine(replacement);
```

La chaîne source ne change pas, et une nouvelle chaîne est retournée avec le remplacement.

Supprimer les espaces blancs

Vous pouvez utiliser les méthodes String.Trim, String.TrimStart et String.TrimEnd pour supprimer les espaces blancs de début ou de fin. Le code suivant montre un exemple de chaque. La chaîne source ne change pas. Ces méthodes retournent une nouvelle chaîne avec le contenu modifié.

C#

```
// Remove trailing and leading white space.
string source = "    I'm wider than I need to be.      ";
// Store the results in a new string variable.
var trimmedResult = source.Trim();
var trimLeading = source.TrimStart();
var trimTrailing = source.TrimEnd();
Console.WriteLine($"<{source}>");
Console.WriteLine($"<{trimmedResult}>");
Console.WriteLine($"<{trimLeading}>");
Console.WriteLine($"<{trimTrailing}>");
```

Supprimer du texte

Vous pouvez supprimer du texte d'une chaîne à l'aide de la méthode String.Remove. Cette méthode supprime un certain nombre de caractères en commençant à un index

spécifique. L'exemple suivant montre comment utiliser String.IndexOf suivi de Remove pour supprimer le texte d'une chaîne :

C#

```
string source = "Many mountains are behind many clouds today.";
// Remove a substring from the middle of the string.
string toRemove = "many ";
string result = string.Empty;
int i = source.IndexOf(toRemove);
if (i >= 0)
{
    result= source.Remove(i, toRemove.Length);
}
Console.WriteLine(source);
Console.WriteLine(result);
```

Remplacer du texte correspondant à un modèle

Vous pouvez utiliser des expressions régulières pour remplacer du texte correspondant à un modèle par un autre texte, éventuellement défini par un modèle. L'exemple suivant utilise la classe System.Text.RegularExpressions.Regex pour trouver un modèle dans une chaîne source et le remplacer par du texte dont la casse est correcte. La méthode Regex.Replace(String, String, MatchEvaluator, RegexOptions) accepte dans ses arguments une fonction qui fournit la logique du remplacement. Dans cet exemple, la fonction `LocalReplaceMatchCase` est une **fonction locale** déclarée à l'intérieur de l'exemple de méthode. `LocalReplaceMatchCase` utilise la classe System.Text.StringBuilder pour générer la chaîne de remplacement avec la casse correcte.

Les expressions régulières sont plus utiles pour rechercher et remplacer du texte qui suit un modèle que du texte connu. Pour plus d'informations, consultez Comment rechercher des chaînes. Le modèle de recherche, « the\s », recherche le mot « the » suivi d'un espace blanc. Cette partie du modèle permet d'exclure le mot « there » qui figure dans la chaîne source. Pour plus d'informations sur les éléments du langage des expressions régulières, consultez Langage des expressions régulières - Aide-mémoire.

C#

```
string source = "The mountains are still there behind the clouds today.";

// Use Regex.Replace for more flexibility.
// Replace "the" or "The" with "many" or "Many".
// using System.Text.RegularExpressions
string replaceWith = "many ";
source = System.Text.RegularExpressions.Regex.Replace(source, "the\\s",
```

```
LocalReplaceMatchCase,
    System.Text.RegularExpressions.RegexOptions.IgnoreCase);
Console.WriteLine(source);

string LocalReplaceMatchCase(System.Text.RegularExpressions.Match
matchExpression)
{
    // Test whether the match is capitalized
    if (Char.IsUpper(matchExpression.Value[0]))
    {
        // Capitalize the replacement string
        System.Text.StringBuilder replacementBuilder = new
System.Text.StringBuilder(replaceWith);
        replacementBuilder[0] = Char.ToUpper(replacementBuilder[0]);
        return replacementBuilder.ToString();
    }
    else
    {
        return replaceWith;
    }
}
```

La méthode StringBuilder.ToString retourne une chaîne immuable avec le contenu dans l'objet StringBuilder.

Modification de caractères individuels

Vous pouvez générer un tableau de caractères à partir d'une chaîne, modifier le contenu du tableau, puis créer une autre chaîne à partir du contenu modifié du tableau.

L'exemple suivant montre comment remplacer un jeu de caractères dans une chaîne. Tout d'abord, il utilise la méthode String.ToCharArray() pour créer un tableau de caractères. Il utilise la méthode IndexOf pour rechercher l'index de départ du mot « fox ». Les trois caractères suivants sont remplacés par un autre mot. Enfin, une nouvelle chaîne est construite à partir du tableau de caractères mis à jour.

C#

```
string phrase = "The quick brown fox jumps over the fence";
Console.WriteLine(phrase);

char[] phraseAsChars = phrase.ToCharArray();
int animalIndex = phrase.IndexOf("fox");
if (animalIndex != -1)
{
    phraseAsChars[animalIndex++] = 'c';
    phraseAsChars[animalIndex++] = 'a';
    phraseAsChars[animalIndex] = 't';
}
```

```
string updatedPhrase = new string(phraseAsChars);
Console.WriteLine(updatedPhrase);
```

Créer du contenu de chaîne par programmation

Étant donné que les chaînes sont immuables, les exemples précédents créent tous des chaînes temporaires ou des tableaux de caractères. Dans les scénarios hautes performances, il peut être souhaitable d'éviter ces allocations de tas. .NET Core fournit une méthode String.Create qui vous permet de remplir par programmation le contenu des caractères d'une chaîne via un rappel tout en évitant les allocations de chaînes temporaires intermédiaires.

C#

```
// constructing a string from a char array, prefix it with some additional
characters
char[] chars = { 'a', 'b', 'c', 'd', '\0' };
int length = chars.Length + 2;
string result = string.Create(length, chars, (Span<char> strContent, char[]
charArray) =>
{
    strContent[0] = '0';
    strContent[1] = '1';
    for (int i = 0; i < charArray.Length; i++)
    {
        strContent[i + 2] = charArray[i];
    }
});

Console.WriteLine(result);
```

Vous pouvez modifier une chaîne dans un bloc fixe avec du code non sécurisé, mais il est **fortement** déconseillé de modifier le contenu de la chaîne après sa création. Cela brisera les choses de manière imprévisible. Par exemple, si quelqu'un interne une chaîne dont le contenu est identique au vôtre, il obtiendra votre copie et ne s'attendra pas à ce que vous modifiiez sa chaîne.

Voir aussi

- Expressions régulières .NET
- Langage des expressions régulières - Aide-mémoire

Comment comparer des chaînes en C#

Article • 03/06/2023

Vous comparez les chaînes pour répondre à l'une des deux questions suivantes : « Ces deux chaînes sont-elles égales ? » ou « Dans quel ordre ces chaînes doivent-elles être placées lors du tri ? »

Ces deux questions sont compliquées en raison des facteurs qui affectent les comparaisons de chaînes :

- Vous pouvez choisir une comparaison ordinale ou linguistique.
- Vous pouvez choisir si la casse a une incidence.
- Vous pouvez choisir des comparaisons propres à la culture.
- Les comparaisons linguistiques dépendent de la culture et de la plateforme.

> **Notes**
>
> Les exemples C# de cet article s'exécutent dans l'exécuteur et le terrain de jeu du code inline **Try.NET**. Sélectionnez le bouton **Exécuter** pour exécuter un exemple dans une fenêtre interactive. Une fois que vous avez exécuté le code, vous pouvez le modifier et exécuter le code modifié en resélectionnant **Exécuter**. La code modifié s'exécute dans la fenêtre interactive ou, si la compilation échoue, la fenêtre interactive affiche tous les messages d'erreur du compilateur C#.

Quand vous comparez des chaînes, vous les ordonnez les unes par rapport aux autres. Les comparaisons permettent de trier une séquence de chaînes. Quand la séquence est dans un ordre connu, elle est plus facile à observer, que ce soit pour les logiciels et pour les humains. Les autres comparaisons peuvent vérifier si les chaînes sont identiques. Ces vérifications de similitude se rapprochent de celles d'égalité, sauf que certaines différences, comme celles qui concernent la casse, peuvent être ignorées.

Comparaisons ordinales par défaut

Par défaut, les opérations les plus courantes :

- String.Equals
- String.Equality et String.Inequality, c'est-à-dire, les opérateurs d'égalité == et !=, respectivement

effectuent une comparaison ordinale respectant la casse. Dans le cas de String.Equals, un argument StringComparison peut être fourni pour modifier ses règles de tri. L'exemple suivant illustre ce qui suit :

C#

```
string root = @"C:\users";
string root2 = @"C:\Users";

bool result = root.Equals(root2);
Console.WriteLine($"Ordinal comparison: <{root}> and <{root2}> are {(result
? "equal." : "not equal.")}");

result = root.Equals(root2, StringComparison.Ordinal);
Console.WriteLine($"Ordinal comparison: <{root}> and <{root2}> are {(result
? "equal." : "not equal.")}");

Console.WriteLine($"Using == says that <{root}> and <{root2}> are {(root ==
root2 ? "equal" : "not equal")}");
```

La comparaison ordinale par défaut ne prend pas en compte les règles linguistiques lors de la comparaison de chaînes. Elle compare la valeur binaire de chaque objet Char dans les deux chaînes. La comparaison ordinale par défaut respecte ainsi également la casse.

Le test d'égalité avec String.Equals et les opérateurs `==` et `!=` diffère de la comparaison de chaîne à l'aide des méthodes String.CompareTo et Compare(String, String). Ils effectuent tous une comparaison respectant la casse. Toutefois, alors que les tests d'égalité effectuent une comparaison ordinale, les méthodes `CompareTo` et `Compare` effectuent une comparaison linguistique tenant compte de la culture à l'aide de la culture actuelle. Étant donné que ces méthodes de comparaison par défaut diffèrent dans leur façon de comparer les chaînes, nous vous recommandons de toujours indiquer clairement l'intention de votre code en appelant une surcharge qui spécifie explicitement le type de comparaison à effectuer.

Comparaisons ordinales ne respectant pas la casse

La méthode String.Equals(String, StringComparison) vous permet de spécifier une valeur StringComparison de StringComparison.OrdinalIgnoreCase pour une comparaison ordinale ne respectant pas la casse. Il existe également une méthode String.Compare(String, String, StringComparison) statique qui effectue une comparaison ordinale ne respectant pas la casse si vous spécifiez la valeur

StringComparison.OrdinalIgnoreCase pour l'argument StringComparison. Elles sont représentées dans le code suivant :

C#

```
string root = @"C:\users";
string root2 = @"C:\Users";

bool result = root.Equals(root2, StringComparison.OrdinalIgnoreCase);
bool areEqual = String.Equals(root, root2,
StringComparison.OrdinalIgnoreCase);
int comparison = String.Compare(root, root2, comparisonType:
StringComparison.OrdinalIgnoreCase);

Console.WriteLine($"Ordinal ignore case: <{root}> and <{root2}> are {(result
? "equal." : "not equal.")}");
Console.WriteLine($"Ordinal static ignore case: <{root}> and <{root2}> are
{(areEqual ? "equal." : "not equal.")}");
if (comparison < 0)
    Console.WriteLine($"<{root}> is less than <{root2}>");
else if (comparison > 0)
    Console.WriteLine($"<{root}> is greater than <{root2}>");
else
    Console.WriteLine($"<{root}> and <{root2}> are equivalent in order");
```

Lorsque vous effectuez une comparaison ordinale ne respectant pas la casse, ces méthodes utilisent les conventions de casse de la culture invariante.

Comparaisons linguistiques

De nombreuses méthodes de comparaison de chaînes (par exemple String.StartsWith) utilisent par défaut des règles linguistiques liées à la *culture active* pour classer leurs entrées. Cela est parfois appelé « ordre de tri des mots ». Lorsque vous effectuez une comparaison linguistique, des pondérations spéciales peuvent être attribuées à certains caractères Unicode non alphabétiques. Par exemple, vous pouvez affecter une faible pondération au trait d'union "-". Ainsi, "co-op" et "coop" s'affichent côte à côte dans l'ordre de tri, tandis que certains caractères de contrôle non imprimables peuvent être complètement ignorés. Par ailleurs, certains caractères Unicode peuvent être équivalents à une séquence d'instances Char. L'exemple suivant utilise l'expression « Ils dansent dans la rue ». en allemand avec « ss » (U+0073 U+0073) dans une chaîne et « ß » (U+00DF) dans une autre. Du point de vue linguistique (dans Windows), « ss » équivaut à l'Esszet allemand : le caractère « ß » dans les cultures « en-US » et « de-DE ».

C#

```
string first = "Sie tanzen auf der Straße.";
string second = "Sie tanzen auf der Strasse.";

Console.WriteLine($"First sentence is <{first}>");
Console.WriteLine($"Second sentence is <{second}>");

bool equal = String.Equals(first, second,
StringComparison.InvariantCulture);
Console.WriteLine($"The two strings {(equal == true ? "are" : "are not")}
equal.");
showComparison(first, second);

string word = "coop";
string words = "co-op";
string other = "cop";

showComparison(word, words);
showComparison(word, other);
showComparison(words, other);
void showComparison(string one, string two)
{
    int compareLinguistic = String.Compare(one, two,
StringComparison.InvariantCulture);
    int compareOrdinal = String.Compare(one, two, StringComparison.Ordinal);
    if (compareLinguistic < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using invariant
culture");
    else if (compareLinguistic > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using invariant
culture");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order
using invariant culture");
    if (compareOrdinal < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using ordinal
comparison");
    else if (compareOrdinal > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using ordinal
comparison");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order
using ordinal comparison");
}
```

Sur Windows, avant .NET 5 l'ordre de tri pour « cop », « coop » et « co-op » change quand vous passez d'une comparaison linguistique à une comparaison ordinale. De même, les deux phrases en allemand ne sont pas comparées de la même façon selon le type de comparaison. En effet, avant .NET 5, les API de globalisation .NET utilisaient des bibliothèques NLS (National Language Support). Dans .NET 5 et les versions ultérieures, les API de globalisation .NET utilisent des bibliothèques de composants internationaux

pour Unicode (ICU), qui unifient le comportement de globalisation .NET sur tous les systèmes d'exploitation pris en charge.

Comparaisons utilisant des cultures spécifiques

Cet exemple stocke des objets CultureInfo pour les cultures en-US et de-DE. Les comparaisons sont effectuées à l'aide de l'un objet CultureInfo pour garantir une comparaison propre à la culture.

La culture utilisée affecte les comparaisons linguistiques. L'exemple suivant montre les résultats de la comparaison des deux phrases en allemand avec la culture « en-US » et la culture « de-DE » :

C#

```
string first = "Sie tanzen auf der Straße.";
string second = "Sie tanzen auf der Strasse.";

Console.WriteLine($"First sentence is <{first}>");
Console.WriteLine($"Second sentence is <{second}>");

var en = new System.Globalization.CultureInfo("en-US");

// For culture-sensitive comparisons, use the String.Compare
// overload that takes a StringComparison value.
int i = String.Compare(first, second, en,
System.Globalization.CompareOptions.None);
Console.WriteLine($"Comparing in {en.Name} returns {i}.");

var de = new System.Globalization.CultureInfo("de-DE");
i = String.Compare(first, second, de,
System.Globalization.CompareOptions.None);
Console.WriteLine($"Comparing in {de.Name} returns {i}.");

bool b = String.Equals(first, second, StringComparison.CurrentCulture);
Console.WriteLine($"The two strings {(b ? "are" : "are not")} equal.");

string word = "coop";
string words = "co-op";
string other = "cop";

showComparison(word, words, en);
showComparison(word, other, en);
showComparison(words, other, en);
void showComparison(string one, string two, System.Globalization.CultureInfo
culture)
{
    int compareLinguistic = String.Compare(one, two, en,
System.Globalization.CompareOptions.None);
    int compareOrdinal = String.Compare(one, two, StringComparison.Ordinal);
```

```
    if (compareLinguistic < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using en-US
culture");
    else if (compareLinguistic > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using en-US
culture");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order
using en-US culture");
    if (compareOrdinal < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using ordinal
comparison");
    else if (compareOrdinal > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using ordinal
comparison");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order
using ordinal comparison");
}
```

Les comparaisons dépendantes de la culture sont généralement utilisées pour comparer et trier des chaînes entrées par plusieurs utilisateurs. Les caractères et les conventions de tri de ces chaînes peuvent varier selon les paramètres régionaux de l'ordinateur de l'utilisateur. Même les chaînes qui contiennent des caractères identiques peuvent être triées différemment selon la culture du thread actuel.

Tri linguistique et recherche de chaînes dans des tableaux

Les exemples suivants montrent comment trier et rechercher des chaînes dans un tableau à l'aide d'une comparaison linguistique dépendante de la culture actuelle. Vous utilisez les méthodes statiques Array qui prennent un paramètre System.StringComparer.

Cet exemple montre comment trier un tableau de chaînes à l'aide de la culture actuelle :

C#

```
string[] lines = new string[]
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};

Console.WriteLine("Non-sorted order:");
foreach (string s in lines)
```

```
{
    Console.WriteLine($"   {s}");
}

Console.WriteLine("\n\rSorted order:");

// Specify Ordinal to demonstrate the different behavior.
Array.Sort(lines, StringComparer.CurrentCulture);

foreach (string s in lines)
{
    Console.WriteLine($"   {s}");
}
```

Une fois que le tableau est trié, vous pouvez rechercher des entrées à l'aide d'une recherche binaire. Une recherche binaire commence au milieu de la collection pour déterminer la moitié de la collection qui doit contenir la chaîne recherchée. Chaque comparaison ultérieure divise à son tour en deux la partie restante de la collection. Le tableau est trié à l'aide de StringComparer.CurrentCulture. La fonction locale `ShowWhere` affiche des informations sur l'emplacement de la chaîne. Si la chaîne est introuvable, la valeur retournée indique où elle serait si elle a été trouvée.

C#

```
string[] lines = new string[]
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};
Array.Sort(lines, StringComparer.CurrentCulture);

string searchString = @"c:\public\TEXTFILE.TXT";
Console.WriteLine($"Binary search for <{searchString}>");
int result = Array.BinarySearch(lines, searchString,
StringComparer.CurrentCulture);
ShowWhere<string>(lines, result);

Console.WriteLine($"{(result > 0 ? "Found" : "Did not find")}
{searchString}");

void ShowWhere<T>(T[] array, int index)
{
    if (index < 0)
    {
        index = ~index;

        Console.Write("Not found. Sorts between: ");

        if (index == 0)
```

```
            Console.Write("beginning of sequence and ");
        else
            Console.Write($"{array[index - 1]} and ");

        if (index == array.Length)
            Console.WriteLine("end of sequence.");
        else
            Console.WriteLine($"{array[index]}.");
    }
    else
    {
        Console.WriteLine($"Found at index {index}.");
    }
}
```

Tri ordinal et recherche dans des collections

Le code suivant utilise la classe de collection System.Collections.Generic.List<T> pour stocker les chaînes. Les chaînes sont triées à l'aide de la méthode List<T>.Sort. Cette méthode a besoin d'un délégué qui compare et trie les deux chaînes. La méthode String.CompareTo fournit cette fonction de comparaison. Exécutez l'exemple et observez l'ordre. Cette opération de tri utilise un tri ordinal respectant la casse. Vous devez utiliser les méthodes statiques String.Compare pour spécifier des règles de comparaison différentes.

C#

```
List<string> lines = new List<string>
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};

Console.WriteLine("Non-sorted order:");
foreach (string s in lines)
{
    Console.WriteLine($"   {s}");
}

Console.WriteLine("\n\rSorted order:");

lines.Sort((left, right) => left.CompareTo(right));
foreach (string s in lines)
{
    Console.WriteLine($"   {s}");
}
```

Une fois triée, la liste de chaînes peut être parcourue à l'aide d'une recherche binaire. L'exemple suivant montre comment effectuer une recherche dans la liste triée à l'aide de la même fonction de comparaison. La fonction locale `ShowWhere` montre où se trouve le texte recherché ou là où il devrait se trouver :

C#

```
List<string> lines = new List<string>
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};
lines.Sort((left, right) => left.CompareTo(right));

string searchString = @"c:\public\TEXTFILE.TXT";
Console.WriteLine($"Binary search for <{searchString}>");
int result = lines.BinarySearch(searchString);
ShowWhere<string>(lines, result);

Console.WriteLine($"{(result > 0 ? "Found" : "Did not find")}
{searchString}");

void ShowWhere<T>(IList<T> collection, int index)
{
    if (index < 0)
    {
        index = ~index;

        Console.Write("Not found. Sorts between: ");

        if (index == 0)
            Console.Write("beginning of sequence and ");
        else
            Console.Write($"{collection[index - 1]} and ");

        if (index == collection.Count)
            Console.WriteLine("end of sequence.");
        else
            Console.WriteLine($"{collection[index]}.");
    }
    else
    {
        Console.WriteLine($"Found at index {index}.");
    }
}
```

Vérifiez toujours que vous utilisez le même type de comparaison pour le tri et la recherche. L'utilisation de différents types de comparaison pour le tri et la recherche produit des résultats inattendus.

Les classes de collection telles que System.Collections.Hashtable, System.Collections.Generic.Dictionary<TKey,TValue> et System.Collections.Generic.List<T> ont des constructeurs qui prennent un paramètre System.StringComparer quand le type des éléments ou des clés est `string`. En général, vous devez utiliser ces constructeurs chaque fois que cela vous est possible, et spécifier StringComparer.Ordinal ou StringComparer.OrdinalIgnoreCase.

Voir aussi

- System.Globalization.CultureInfo
- System.StringComparer
- Chaînes
- Comparaison de chaînes
- Globalisation et localisation d'applications

Comment intercepter une exception non-CLS

Article • 10/05/2023

Certains langages .NET, dont C++/CLI, permettent aux objets de lever des exceptions qui ne dérivent pas d'Exception. De telles exceptions sont appelées *exceptions non-CLS* ou *non exceptions*. Dans C#, vous ne pouvez pas lever d'exceptions non-CLS, mais vous pouvez les intercepter de deux façons :

- Dans un bloc `catch (RuntimeWrappedException e)`.

  Par défaut, un assembly Visual C# intercepte les exceptions non-CLS comme des exceptions encapsulées. Utilisez cette méthode si vous devez accéder à l'exception d'origine, qui est accessible via la propriété RuntimeWrappedException.WrappedException. La procédure située plus loin dans cette rubrique explique comment intercepter les exceptions de cette manière.

- Dans un bloc catch général (bloc catch sans type d'exception spécifié) qui est placé après tous les autres blocs `catch`.

  Utilisez cette méthode lorsque vous souhaitez effectuer une action (comme écrire dans un fichier journal) en réponse à des exceptions non-CLS, et que vous n'avez pas besoin d'accéder aux informations de l'exception. Par défaut, le common language runtime inclut dans un wrapper toutes les exceptions. Pour désactiver ce comportement, ajoutez cet attribut d'assembly à votre code, généralement dans le fichier AssemblyInfo.cs : `[assembly: RuntimeCompatibilityAttribute(WrapNonExceptionThrows = false)]`.

Comment intercepter une exception non-CLS

Dans un bloc `catch(RuntimeWrappedException e)`, accédez à l'exception d'origine via la propriété RuntimeWrappedException.WrappedException.

Exemple

L'exemple suivant montre comment intercepter une exception non-CLS levée à partir d'une bibliothèque de classes écrite en C++/CLI. Notez que dans cet exemple, le code client C# sait par avance que le type d'exception levé est un System.String. Vous pouvez

caster la propriété RuntimeWrappedException.WrappedException en son type d'origine, tant que celui-ci est accessible à partir de votre code.

C#

```
// Class library written in C++/CLI.
var myClass = new ThrowNonCLS.Class1();

try
{
    // throws gcnew System::String(
    // "I do not derive from System.Exception!");
    myClass.TestThrow();
}
catch (RuntimeWrappedException e)
{
    String s = e.WrappedException as String;
    if (s != null)
    {
        Console.WriteLine(s);
    }
}
```

Voir aussi

- RuntimeWrappedException
- Exceptions et gestion des exceptions

Kit SDK .NET Compiler Platform

Article • 25/02/2023

Les compilateurs génèrent un modèle détaillé du code applicatif lorsqu'ils en valident la syntaxe et la sémantique. Ce modèle permet de générer la sortie exécutable à partir du code source. Le Kit SDK .NET Compiler Platform donne accès à ce modèle. Nous nous appuyons de plus en plus sur des fonctionnalités d'environnements de développement intégré (IDE) telles qu'IntelliSense, la refactorisation, le renommage intelligent, la recherche de l'ensemble des références et l'accès direct aux définitions pour augmenter notre productivité. Nous utilisons des outils d'analyse du code pour améliorer la qualité de notre code et des générateurs de code pour faciliter la construction des applications. Plus ces outils gagnent en intelligence, plus ils ont besoin d'accéder à une grande part du modèle que seuls les compilateurs créent lorsqu'ils traitent le code applicatif. Il s'agit de la mission centrale des API Roslyn : ouvrir les boîtes opaques et laisser les outils et les utilisateurs finaux partager toute la richesse des informations que possèdent les compilateurs sur notre code. Au lieu de rester d'opaques traducteurs de code source en code objet, les compilateurs deviennent des plateformes avec Roslyn : des API utilisables pour les tâches portant sur le code dans les outils et les applications souhaités.

Concepts du Kit SDK .NET Compiler Platform

Le Kit SDK .NET Compiler Platform réduit considérablement la barrière à l'entrée pour la création d'applications et d'outils axés sur le code. Il offre de nombreuses possibilités d'innovation dans différents champs, notamment la métaprogrammation, la génération et la transformation de code, l'utilisation interactive des langages C# et Visual Basic et l'incorporation de C# et de Visual Basic dans des langages propres au domaine.

Le SDK .NET Compiler Platform permet de générer des ***analyseurs*** et des ***correctifs de code*** qui recherchent et corrigent les erreurs de codage. Les ***analyseurs*** comprennent la syntaxe (structure de code) et la sémantique pour détecter les pratiques à corriger. Les ***correctifs de code*** représentent des propositions de correction des erreurs de codage trouvées par les analyseurs ou les diagnostics du compilateur. En règle générale, un analyseur et les correctifs de code associés sont regroupés dans un seul projet.

Les analyseurs et les correctifs de code utilisent l'analyse statique pour comprendre le code. Ils n'exécutent pas le code, et n'offrent aucun autre avantage en matière de tests. Ils peuvent, toutefois, signaler des pratiques qui aboutissent souvent à des bogues, à du code empêchant toute mise à jour ou à la violation des recommandations standard.

Outre les analyseurs et les correctifs de code, le SDK .NET Compiler Platform vous permet également de générer des ***refactorisations de code***. Il se compose aussi d'un unique ensemble d'API permettant d'examiner et de comprendre un codebase C# ou Visual Basic. Grâce à ce codebase unique, il est plus facile d'écrire des analyseurs et des correctifs de code en utilisant les API d'analyse syntaxique et sémantique fournies par le Kit SDK .NET Compiler Platform. Une fois libéré de la tâche chronophage qui consiste à répliquer l'analyse effectuée par le compilateur, vous pouvez vous concentrer sur la tâche, plus ciblée, de recherche et de résolution des erreurs de codage courantes de votre projet ou de votre bibliothèque.

Autre avantage, plus modeste, vos analyseurs et vos correctifs de code sont plus petits et utilisent beaucoup moins de mémoire une fois chargés dans Visual Studio que si vous aviez écrit votre propre codebase pour comprendre le code d'un projet. En exploitant les mêmes classes que le compilateur et Visual Studio, vous pouvez créer vos propres outils d'analyse statique. Votre équipe a donc la possibilité d'utiliser des analyseurs et des correctifs de code sans impact perceptible sur les performances de l'IDE.

Il existe trois grands scénarios d'écriture d'analyseurs et de correctifs de code :

1. *Appliquer des normes de codage pour l'équipe*
2. *Offrir de l'aide sur les packages de bibliothèque*
3. *Proposer des conseils généraux*

Appliquer des normes de codage pour l'équipe

De nombreuses équipes ont des normes de codage, qui sont appliquées par le biais des révisions du code effectuées avec d'autres membres de l'équipe. Les analyseurs et les correctifs de code peuvent rendre ce processus beaucoup plus efficace. Les revues de code se produisent quand un développeur partage son travail avec d'autres membres de l'équipe. Le développeur aura investi tout le temps nécessaire pour terminer une nouvelle fonctionnalité avant de recevoir le moindre commentaire. Plusieurs semaines peuvent s'écouler, au cours desquelles le développeur renforce des habitudes qui ne correspondent pas aux pratiques de l'équipe.

Les analyseurs s'exécutent au moment même où il écrit du code. Le développeur obtient immédiatement des commentaires qui l'encouragent à suivre les instructions sans attendre. Le développeur prend l'habitude d'écrire du code conforme dès qu'il commence le prototypage. Lorsque la fonctionnalité est prête à être révisée par des humains, toutes les instructions standard ont été appliquées.

Les équipes peuvent créer des analyseurs et des correctifs de code permettant de rechercher les pratiques les plus courantes qui enfreignent les pratiques de codage de

l'équipe. Ils peuvent être installés sur l'ordinateur de chaque développeur pour appliquer les normes.

> **Conseil**
>
> Avant de créer votre propre analyseur, consultez les analyseurs intégrés. Pour plus d'informations, consultez **Règles de style de code**.

Offrir de l'aide sur les packages de bibliothèque

De nombreuses bibliothèques sont disponibles pour les développeurs .NET sur NuGet. Certaines proviennent de Microsoft ou de sociétés tierces, d'autres sont proposées par des membres de la communauté et des bénévoles. Ces bibliothèques sont plus souvent utilisées et sont mieux notées lorsque les développeurs réussissent à les exploiter.

En plus de la documentation, vous pouvez proposer des analyseurs et des correctifs de code qui recherchent et corrigent les mauvais emplois courants de votre bibliothèque. Ces corrections immédiates aident les développeurs à en tirer parti plus rapidement.

Vous pouvez ajouter les analyseurs et les correctifs de code au package de votre bibliothèque sur NuGet. Dans ce scénario, un développeur qui installe votre package NuGet installera également le package de l'analyseur. Tous les développeurs qui utilisent votre bibliothèque obtiennent immédiatement de l'aide de la part de votre équipe sous la forme de commentaires immédiats sur les erreurs et de suggestions de corrections.

Proposer des conseils généraux

La communauté de développeurs .NET a découvert, par l'expérience, des modèles qui fonctionnent bien et d'autres qu'il vaut mieux éviter. Plusieurs membres de la communauté ont créé des analyseurs qui appliquent ces modèles recommandés. Plus nous gagnons en expérience, plus il y a de place pour de nouvelles idées.

Ces analyseurs peuvent être chargés sur Visual Studio Marketplace et téléchargés par les développeurs avec Visual Studio. Ceux qui débutent avec le langage et la plateforme apprennent rapidement les pratiques acceptées et deviennent plus vite productifs, dès le début de leur parcours d'apprentissage de .NET. Comme ils sont de plus en plus utilisés, la communauté adopte ces pratiques.

Étapes suivantes

Le Kit SDK .NET Compiler Platform inclut les derniers modèles objet du langage pour la génération, l'analyse et la refactorisation de code. Cette section offre une vue d'ensemble conceptuelle du Kit SDK .NET Compiler Platform. Vous trouverez plus de détails dans les sections des didacticiels, des exemples et des démarrages rapides.

Pour en savoir plus sur les concepts du Kit SDK .NET Compiler Platform, consultez ces cinq rubriques :

- Explorer le code avec le visualiseur de syntaxe
- Comprendre le modèle de l'API du compilateur
- Utiliser la syntaxe
- Utiliser la sémantique
- Utiliser un espace de travail

Pour commencer, vous devez installer le kit **.NET Compiler Platform SDK** :

Instructions d'installation - Visual Studio Installer

Vous pouvez rechercher **SDK .NET Compiler Platform** dans **Visual Studio Installer** de deux façons :

Vue Installation avec Visual Studio Installer - Charges de travail

SDK .NET Compiler Platform n'est pas sélectionné automatiquement dans le cadre de la charge de travail Développement d'extensions Visual Studio. Vous devez le sélectionner comme composant facultatif.

1. Exécutez **Visual Studio Installer**.
2. Sélectionnez **Modifier**
3. Choisissez la charge de travail **Développement d'extensions Visual Studio**.
4. Ouvrez le nœud **Développement d'extensions Visual Studio** dans l'arborescence résumée.
5. Cochez la case pour **SDK .NET Compiler Platform**. Il se trouve en dernier sous les composants facultatifs.

Si vous le souhaitez, vous pouvez ajouter **l'éditeur DGML** pour afficher des graphes dans le visualiseur :

1. Ouvrez le nœud **Composants individuels** dans l'arborescence résumée.
2. Cochez la case pour **Éditeur DGML**.

Onglet Installation avec Visual Studio Installer - Composants individuels

1. Exécutez **Visual Studio Installer**.
2. Sélectionnez **Modifier**
3. Sélectionnez l'onglet **Composants individuels**.
4. Cochez la case pour **SDK .NET Compiler Platform**. Il se trouve en haut de la liste, sous la section **Compilateurs, outils de génération et runtimes**.

Si vous le souhaitez, vous pouvez ajouter **l'éditeur DGML** pour afficher des graphes dans le visualiseur :

1. Cochez la case pour **Éditeur DGML**. Il se trouve sous la section **Outils de code**.

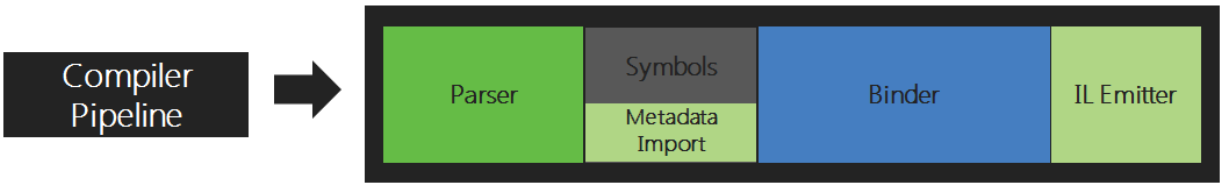
Comprendre le modèle du SDK .NET Compiler Platform

Article • 10/05/2023

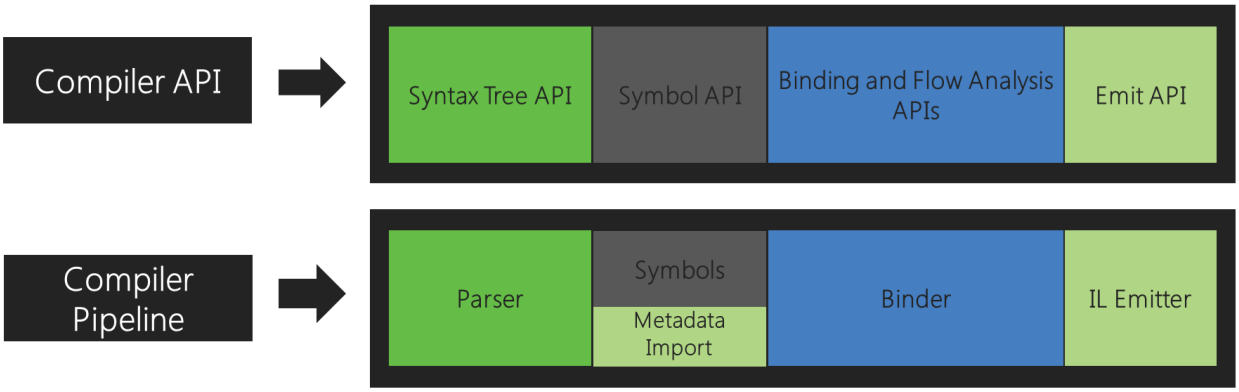
Les compilateurs traitent le code que vous écrivez en suivant des règles structurées qui diffèrent souvent de la façon dont vous pouvez vous-même lire et comprendre le code. Pour bien comprendre le fonctionnement des API dont vous avez besoin pour créer des outils Roslyn, vous devez connaître les concepts de base du modèle utilisé par les compilateurs.

Zones fonctionnelles du pipeline de compilateur

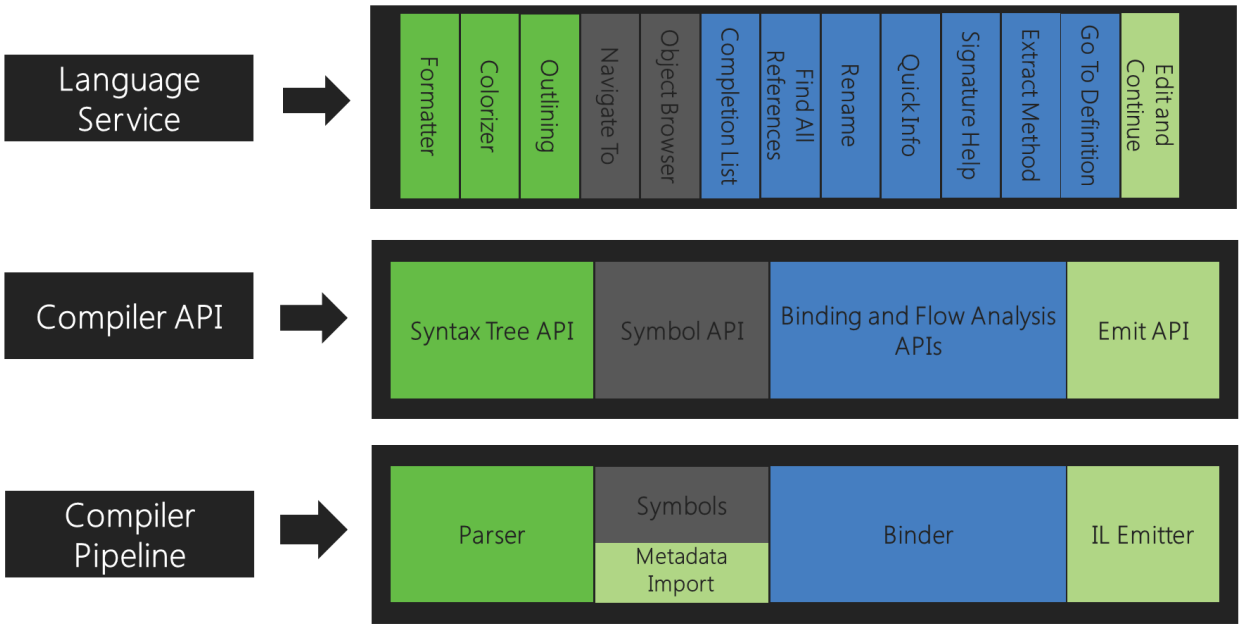
Le SDK .NET Compiler Platform expose les analyses de code des compilateurs C# et Visual Basic en vous fournissant, en tant que consommateur, une couche d'API qui reflète un pipeline de compilateur standard.



Chaque phase de ce pipeline correspond à un composant distinct. Tout d'abord, la phase d'analyse segmente et analyse le texte source dans la syntaxe grammaticale du langage utilisé. Ensuite, la phase de déclaration analyse les métadonnées sources et importées pour former des symboles nommés. Après cela, la phase de liaison mappe les identificateurs dans le code à des symboles. Enfin, la phase d'émission émet un assembly qui contient toutes les informations générées par le compilateur.



Pour chacune de ces phases, le SDK .NET Compiler Platform expose un modèle objet qui permet d'accéder aux informations disponibles à la phase en question. La phase d'analyse expose une arborescence de syntaxe, la phase de déclaration expose une table de symboles hiérarchique, la phase de liaison expose le résultat de l'analyse sémantique du compilateur et la phase d'émission est une API qui produit des codes d'octets IL.



Chaque compilateur combine ces composants en une solution de bout en bout unique.

Ces API sont les mêmes que celles utilisées par Visual Studio. Par exemple, les fonctionnalités de mise en forme et en plan du code utilisent les arborescences de syntaxe, les fonctionnalités de l'**Explorateur d'objets** et de navigation utilisent la table de symboles, les fonctionnalités de refactorisation et **Atteindre la définition** utilisent le modèle sémantique, et la fonctionnalité **Modifier et Continuer** utilise tous ces éléments, y compris l'API d'émission.

Couches d'API

Le SDK du compilateur .NET est constitué de plusieurs couches d'API : les API du compilateur, les API de diagnostic, les API de script et les API d'espaces de travail.

API du compilateur

La couche du compilateur contient les modèles objet qui correspondent aux informations syntaxiques et sémantiques exposées à chaque phase du pipeline du compilateur. La couche du compilateur contient également un instantané immuable d'un appel unique d'un compilateur, notamment les références d'assembly, les options du compilateur et les fichiers de code source. Le langage C# et le langage Visual Basic sont représentés par deux API distinctes. Les deux API sont similaires dans la forme,

mais elles sont adaptées à chaque langage pour garantir une haute fidélité. Cette couche n'a pas de dépendances sur les composants Visual Studio.

API de diagnostic

Dans le cadre de son analyse, le compilateur peut produire un ensemble de diagnostics complets, qui couvrent les erreurs de syntaxe, de sémantique et d'assignation définie, ainsi que divers avertissements et informations. La couche API du compilateur expose les diagnostics par le biais d'une API extensible qui permet d'intégrer des analyseurs définis par l'utilisateur au processus de compilation. Des diagnostics définis par l'utilisateur, tels que ceux générés par des outils comme StyleCop, peuvent ainsi être produits en même temps que les diagnostics définis par le compilateur. Les diagnostics générés de cette manière offrent l'avantage de s'intégrer naturellement à des outils comme MSBuild et Visual Studio qui se basent sur les diagnostics pour effectuer certaines actions (par exemple, arrêter une build basée sur une stratégie, afficher en temps réel des lignes ondulées dans l'éditeur et suggérer des corrections du code).

API de script

Les API d'hébergement et de script font partie de la couche du compilateur. Vous pouvez utiliser ces API pour l'exécution d'extraits de code et l'accumulation d'un contexte d'exécution du runtime. Le REPL (Read-Evaluate-Print Loop) interactif C# utilise ces API. Le REPL vous permet d'utiliser C# comme langage de script, en exécutant le code que vous écrivez de manière interactive.

API des espaces de travail

La couche des espaces de travail contient les API des espaces de travail, à partir desquelles sont effectuées les analyses de code et la refactorisation dans les solutions entières. Elle vous aide à organiser toutes les informations sur les projets d'une solution dans un modèle objet unique. Ce modèle vous permet d'accéder directement aux modèles objet de la couche du compilateur sans avoir à analyser des fichiers, configurer des options ou gérer des dépendances entre projets.

De plus, la couche des espaces de travail expose un ensemble d'API utilisées pour l'implémentation d'outils d'analyse du code et de refactorisation qui fonctionnent dans un environnement hôte, tel que l'IDE Visual Studio. Les API Rechercher toutes les références, Mise en forme et Génération de code en sont des exemples.

Cette couche n'a pas de dépendances sur les composants Visual Studio.

Utiliser la syntaxe

Article • 10/05/2023

Une *arborescence de syntaxe* est une structure de données immuable de base qui est exposée par les API du compilateur. Ces arborescences représentent la structure lexicale et syntaxique du code source. Elles ont deux utilités principales :

- Permettre aux outils (comme un IDE, des compléments, des outils d'analyse de code et des refactorisations) d'afficher et de traiter la structure syntaxique du code source dans le projet d'un utilisateur.
- Activer des outils, tels que des refactorisations et un IDE, pour créer, modifier et réorganiser le code source de manière naturelle sans avoir à utiliser des modifications de texte directes. Par la création et la manipulation des arborescences, les outils peuvent facilement créer et réorganiser le code source.

Arborescences de syntaxe

Les arborescences de syntaxe sont la structure principale utilisée pour la compilation, l'analyse de code, la liaison, la refactorisation, les fonctionnalités de l'IDE et la génération de code. Aucune partie du code source ne peut être comprise sans avoir au préalable été identifiée et classée dans l'un des nombreux éléments de langage structurel connus.

Les arborescences de syntaxe ont trois caractéristiques principales.

- Elles contiennent toutes les informations sources et les représentent en toute fidélité. La fidélité totale signifie que l'arborescence de syntaxe contient toutes les informations trouvées dans le texte source, chaque construction grammaticale, chaque jeton lexical et tout le contenu entre ces éléments, notamment les directives d'espace blanc, de commentaires et de préprocesseur. Par exemple, chaque littéral mentionné dans la source est représenté exactement comme il a été tapé. Les arborescences de syntaxe capturent également les erreurs dans le code source quand le programme est incomplet ou incorrect en affichant les jetons ignorés ou manquants.
- Elles peuvent produire le texte exact à partir duquel ils ont été analysés. À partir de n'importe quel nœud de syntaxe, il est possible d'obtenir la représentation texte de la sous-arborescence ayant ce nœud pour racine. Cette fonctionnalité signifie que vous pouvez utiliser les arborescences de syntaxe pour créer et modifier le texte source. La création d'une arborescence implique que vous avez créé le texte

équivalent, et la création d'une autre arborescence à partir des modifications apportées à une arborescence existante, implique que vous avez modifié le texte.
- Ils sont immuables et thread-safe. Une fois qu'une arborescence a été obtenue, elle est un instantané de l'état actuel du code et elle ne peut plus être changée. Cela permet à plusieurs utilisateurs d'interagir simultanément avec la même arborescence de syntaxe dans différents threads sans verrouillage ou duplication. Étant donné que les arborescences sont immuables et qu'elles ne peuvent pas être modifiées directement, les méthodes de fabrique facilitent la création et la modification des arborescences de syntaxe en créant des instantanés supplémentaires de l'arborescence. Les arborescences réutilisent avantageusement les nœuds sous-jacents pour accélérer la génération d'une nouvelle version, sans consommer beaucoup plus de mémoire.

Une arborescence de syntaxe est littéralement une structure de données arborescente, où des éléments structurels non terminaux sont parents d'autres éléments. Chaque arborescence de syntaxe est constituée de nœuds, de jetons et de trivia.

Nœuds de syntaxe

Les nœuds de syntaxe figurent parmi les principaux éléments des arborescences de syntaxe. Ces nœuds représentent des constructions syntaxiques, telles que des déclarations, des instructions, des clauses et des expressions. Chaque catégorie de nœuds de syntaxe est représentée par une classe distincte dérivée de Microsoft.CodeAnalysis.SyntaxNode. L'ensemble de classes de nœud n'est pas extensible.

Tous les nœuds de syntaxe sont des nœuds non terminaux dans l'arborescence de syntaxe ; ils ont donc toujours d'autres nœuds et jetons comme enfants. Chaque nœud enfant d'un autre nœud a un nœud parent qui est accessible via la propriété SyntaxNode.Parent. Comme les nœuds et les arborescences sont immuables, le parent d'un nœud ne change jamais. La racine de l'arborescence a un parent Null.

Chaque nœud a une méthode SyntaxNode.ChildNodes(), qui retourne une liste des nœuds enfants dans l'ordre séquentiel déterminé par leur position dans le texte source. Cette liste ne contient pas de jetons. Chaque nœud a également des méthodes qui retournent les descendants. Ainsi, les méthodes DescendantNodes, DescendantTokens ou DescendantTrivia représentent une liste de tous les nœuds, jetons ou trivia qui existent dans la sous-arborescence ayant ce nœud comme racine.

En outre, chaque sous-classe de nœud de syntaxe expose les mêmes enfants par le biais de propriétés fortement typées. Par exemple, une classe de nœud BinaryExpressionSyntax a trois propriétés supplémentaires propres aux opérateurs

binaires : Left, OperatorToken et Right. Le type de Left et Right est ExpressionSyntax, et le type de OperatorToken est SyntaxToken.

Certains nœuds de syntaxe ont des enfants facultatifs. Par exemple, IfStatementSyntax a un ElseClauseSyntax facultatif. Si l'enfant n'est pas trouvé, la propriété retourne la valeur Null.

Jetons de syntaxe

Les jetons de syntaxe sont les éléments terminaux dans la grammaire d'un langage ; ils représentent les plus petits fragments syntaxiques du code. Ils ne sont jamais parents d'autres nœuds ou jetons. Les jetons de syntaxe se composent de mots clés, d'identificateurs, de littéraux et de caractères de ponctuation.

Par souci d'efficacité, le type SyntaxToken est un type valeur CLR. Par conséquent, contrairement aux nœuds de syntaxe, il n'y a qu'une seule structure pour tous les genres de jetons, avec une combinaison de propriétés dont le sens dépend du genre du jeton représenté.

Par exemple, un jeton de littéral d'entier représente une valeur numérique. En plus du texte source brut sur lequel s'étend le jeton, le jeton de littéral a une propriété Value qui indique précisément la valeur entière décodée. Cette propriété est typée en Object, car elle peut être l'un des nombreux types primitifs.

La propriété ValueText fournit les mêmes informations que la propriété Value, mais elle est toujours typée en String. Dans du texte source C#, un identificateur peut inclure des caractères d'échappement Unicode, mais la syntaxe de la séquence d'échappement n'est pas considérée comme faisant partie du nom de l'identificateur. Même si le texte brut sur lequel s'étend le jeton inclut la séquence d'échappement, la propriété ValueText ne l'inclut pas. À la place, elle inclut les caractères Unicode identifiés par la séquence d'échappement. Par exemple, si le texte source contient un identificateur écrit sous la forme `\u03C0`, la propriété ValueText pour ce jeton retourne `π`.

Trivia de syntaxe

Les trivia de syntaxe représentent les parties du texte source qui n'ont pas une grande utilité pour la compréhension générale du code. Il s'agit notamment des espaces blancs, des commentaires et des directives de préprocesseur. Comme les jetons de syntaxe, les trivia sont des types valeur. Le type Microsoft.CodeAnalysis.SyntaxTrivia unique est utilisé pour décrire tous les genres de trivia.

Du fait que les trivia ne font pas partie de la syntaxe du langage à proprement parler et qu'ils peuvent se trouver n'importe où entre deux jetons, ils ne sont pas représentés dans l'arborescence de syntaxe comme enfants d'un nœud. Les trivia sont toutefois inclus dans l'arborescence de syntaxe, car ils sont importants lors de l'implémentation d'une fonctionnalité telle que la refactorisation, mais aussi pour garantir une haute fidélité avec le texte source.

Vous pouvez accéder aux trivia en inspectant les collections SyntaxToken.LeadingTrivia ou SyntaxToken.TrailingTrivia d'un jeton. Quand le texte source est analysé, les séquences de trivia sont associées aux jetons. En règle générale, un jeton est propriétaire de tous les trivia qui se trouvent après lui sur la même ligne. Les trivia situés après cette ligne dépendent du jeton suivant. Dans le fichier source, le premier jeton obtient tous les trivia initiaux, et la dernière séquence de trivia est associée au jeton de fin de fichier, qui a sinon une largeur égale à zéro.

Contrairement aux nœuds et jetons de syntaxe, les trivia de syntaxe n'ont pas de parents. Toutefois, comme les trivia font partie de l'arborescence et que chacun d'eux est associé à un jeton unique, vous pouvez accéder au jeton correspondant à l'aide de la propriété SyntaxTrivia.Token.

Étendues

Chaque nœud, jeton ou trivia connaît sa position dans le texte source et le nombre de caractères qui le composent. La position du texte est représentée par un entier 32 bits, qui est un index `char` de base zéro. L'objet TextSpan représente la position de début et le nombre de caractères sous forme de deux entiers. Si TextSpan a une longueur nulle, il fait référence à une position entre deux caractères.

Chaque nœud a deux propriétés TextSpan : Span et FullSpan.

La propriété Span définit l'étendue de texte comprise entre le début du premier jeton dans la sous-arborescence du nœud et la fin du dernier jeton. Cette étendue n'inclut pas les trivia de début ou de fin.

La propriété FullSpan définit l'étendue de texte qui inclut l'étendue du nœud, plus l'étendue des trivia de début ou de fin.

Par exemple :

C#

```
      if (x > 3)
      {
||        // this is bad
```

```
            |throw new Exception("Not right.");|  // better exception?||
        }
```

L'étendue du nœud de l'instruction dans le bloc est délimitée par deux barres verticales simples (|). Elle inclut les caractères `throw new Exception("Not right.");`. L'étendue complète est délimitée par les deux barres verticales doubles (||). Elle inclut les mêmes caractères que l'étendue du nœud, plus les caractères associés aux trivia de début et de fin.

Genres

Chaque nœud, jeton ou trivia a une propriété SyntaxNode.RawKind de type System.Int32, qui identifie précisément l'élément de syntaxe représenté. Cette valeur peut être convertie en énumération spécifique au langage. Chaque langage, C# ou Visual Basic, a une seule énumération `SyntaxKind` (Microsoft.CodeAnalysis.CSharp.SyntaxKind et Microsoft.CodeAnalysis.VisualBasic.SyntaxKind, respectivement) qui répertorie tous les nœuds, jetons et éléments trivia possibles au niveau syntaxique. Cette conversion peut être effectuée automatiquement en accédant aux méthodes d'extension CSharpExtensions.Kind ou VisualBasicExtensions.Kind.

La propriété RawKind permet de lever facilement toute ambiguïté sur les types de nœud de syntaxe qui utilisent la même classe de nœud. Pour les jetons et les trivia, cette propriété est le seul moyen de différencier les types d'élément entre eux.

Prenons l'exemple d'une classe BinaryExpressionSyntax, qui a les enfants Left, OperatorToken et Right. La propriété Kind détermine si le nœud de syntaxe est du genre AddExpression, SubtractExpression ou MultiplyExpression.

> **Conseil**
>
> Il est recommandé de vérifier les types à l'aide de méthodes d'extension **IsKind** (pour C#) ou **IsKind** (pour VB).

Erreurs

Même si le texte source contient des erreurs de syntaxe, une arborescence de syntaxe complète avec aller-retour au code source est exposée. Quand l'analyseur rencontre du code qui n'est pas conforme à la syntaxe définie pour le langage, il crée une arborescence de syntaxe à l'aide d'une des deux techniques suivantes :

- Quand l'analyseur ne trouve pas le genre de jeton attendu, il insère un jeton manquant dans l'arborescence de syntaxe, à l'emplacement où le jeton était attendu. Un jeton manquant représente le jeton qui était attendu, mais du fait qu'il a une étendue vide, sa propriété SyntaxNode.IsMissing retourne `true`.

- L'analyseur ignore les jetons jusqu'à ce qu'il en trouve un à partir duquel il peut poursuivre l'analyse. Dans ce cas, les jetons ignorés sont attachés en tant que nœud de trivia avec le genre SkippedTokensTrivia.

Utiliser la sémantique

Article • 10/05/2023

Les arborescences de syntaxe représentent la structure lexicale et syntaxique du code source. Cette information est suffisante pour décrire toutes les déclarations et la logique utilisées dans le code source, mais elle ne l'est pas pour identifier l'élément référencé. Un nom peut représenter les éléments suivants :

- Un type
- Un champ
- Une méthode
- Une variable locale

Ces éléments se distinguent par des caractéristiques uniques. Toutefois, pour déterminer à quel élément un identificateur fait réellement référence, avoir une connaissance approfondie des règles de langage s'avère souvent indispensable.

Certains éléments de programme sont représentés dans le code source, mais les programmes peuvent aussi faire référence à des bibliothèques précédemment compilées, empaquetées dans des fichiers d'assembly. Il n'y a pas de code source et donc pas d'arborescences ni nœuds de syntaxe disponibles pour les assemblys, mais les programmes peuvent faire référence à des éléments qu'ils contiennent.

Pour ces tâches, vous devez utiliser le **modèle sémantique**.

Par rapport à un modèle syntaxique du code source, un modèle sémantique encapsule en plus les règles de langage, ce qui vous aide à associer correctement les identificateurs avec l'élément de programme référencé.

Compilation

Une compilation est une représentation de tous les éléments nécessaires à la compilation d'un programme C# ou Visual Basic, à savoir l'ensemble des références d'assembly, des options du compilateur et des fichiers sources.

Toutes ces informations étant stockées au même emplacement, les éléments contenus dans le code source peuvent être décrits plus en détail. La compilation représente chaque élément déclaré (type, membre ou variable) sous forme de symbole. La compilation contient diverses méthodes qui vous aident à trouver et associer les symboles qui ont été déclarés dans le code source, ou importés en tant que métadonnées à partir d'un assembly.

De la même façon que les arborescences de syntaxe, les compilations sont immuables. Une fois créée, une compilation ne peut plus être modifiée, que ce soit par vous ou par tout autre utilisateur avec qui vous la partagez. Toutefois, vous pouvez créer une autre compilation à partir d'une compilation existante, en y apportant la modification souhaitée. Par exemple, vous pouvez créer une compilation semblable en tous points à une compilation existante, mais y ajouter un fichier source ou une référence d'assembly supplémentaire.

symboles

Un symbole représente un élément distinct qui est déclaré dans le code source, ou importé en tant que métadonnées à partir d'un assembly. Chaque élément (espace de noms, type, méthode, propriété, champ, événement, paramètre ou variable locale) est représenté par un symbole.

Diverses méthodes et propriétés du type Compilation vous aident à trouver les symboles. Par exemple, vous pouvez rechercher un symbole d'un type déclaré par son nom de métadonnées commun. Vous pouvez aussi accéder à la table de symboles entière comme une arborescence de symboles ayant pour racine l'espace de noms global.

Les symboles contiennent également des informations supplémentaires que le compilateur collecte à partir du code source ou des métadonnées, par exemple, les autres symboles référencés. Chaque genre de symbole est représenté par une interface distincte dérivée de l'interface ISymbol, qui a ses propres méthodes et propriétés pour décrire en détail les informations collectées par le compilateur. Nombre de ces propriétés référencent directement d'autres symboles. Par exemple, la propriété IMethodSymbol.ReturnType indique quel symbole de type est retourné par la méthode.

Les symboles fournissent une représentation des espaces de noms, types et membres commune au code source et aux métadonnées. Par exemple, une méthode déclarée dans le code source et une méthode importée des métadonnées sont toutes les deux représentées par un symbole IMethodSymbol ayant les mêmes propriétés.

Sur un plan conceptuel, les symboles sont similaires au système de type CLR représenté par l'API System.Reflection, mais ils sont plus complets dans la mesure où ils modélisent d'autres éléments que des types. Les espaces de noms, les variables locales et les étiquettes sont tous des symboles. De plus, les symboles sont une représentation des concepts du langage, pas des concepts du CLR. Il y a beaucoup de similitudes, mais aussi de nombreuses différences importantes. Par exemple, une méthode d'itérateur en C# ou Visual Basic est un symbole unique. Quand la méthode d'itérateur est convertie en métadonnées CLR, elle devient un type avec plusieurs méthodes.

Modèle sémantique

Un modèle sémantique représente toutes les informations sémantiques relatives à un fichier source unique. Vous pouvez utiliser ce modèle pour découvrir les éléments suivants :

- Les symboles référencés à un emplacement spécifique dans le code source.
- Le type résultant d’une expression.
- Tous les diagnostics (erreurs et avertissements).
- Les flux entrants et sortants des variables dans les régions du code source.
- Les réponses à des questions plus spéculatives.

Utiliser un espace de travail

Article • 10/05/2023

La couche des **espaces de travail** est l'emplacement à partir duquel sont effectuées l'analyse de code et la refactorisation dans les solutions entières. Dans cette couche, l'API des espaces de travail vous aide à organiser toutes les informations sur les projets d'une solution dans un modèle objet unique. Ce modèle vous permet d'accéder directement aux modèles objet de la couche du compilateur, tels que le texte source, les arborescences de syntaxe, les modèles sémantiques et les compilations, sans avoir à analyser des fichiers, configurer des options ou gérer des dépendances entre projets.

Les environnements hôtes, comme un IDE, vous fournissent un espace de travail correspondant à la solution ouverte. Vous pouvez également utiliser ce modèle en dehors d'un IDE, simplement en chargeant un fichier solution.

Espace de travail

Un espace de travail est une représentation active de la solution sous forme de collection de projets, chacun de ces projets contenant une collection de documents. Un espace de travail est généralement lié à un environnement hôte, qui change en permanence à mesure qu'un utilisateur tape ou manipule des propriétés.

La classe Workspace fournit l'accès au modèle actif de la solution. Quand un changement se produit dans l'environnement hôte, l'espace de travail déclenche les événements correspondants, et la propriété Workspace.CurrentSolution est mise à jour. Par exemple, quand un utilisateur tape dans un éditeur de texte correspondant à l'un des documents sources, l'espace de travail déclenche un événement pour signaler que le modèle global de la solution a changé et indiquer quel document a été modifié. Vous pouvez ensuite réagir à ces changements, par exemple, vérifier si le nouveau modèle est correct, surligner les zones d'intérêt ou suggérer une modification du code.

Vous pouvez également créer des espaces de travail autonomes qui sont déconnectés de l'environnement hôte ou qui sont utilisés dans une application sans environnement hôte.

Solutions, projets et documents

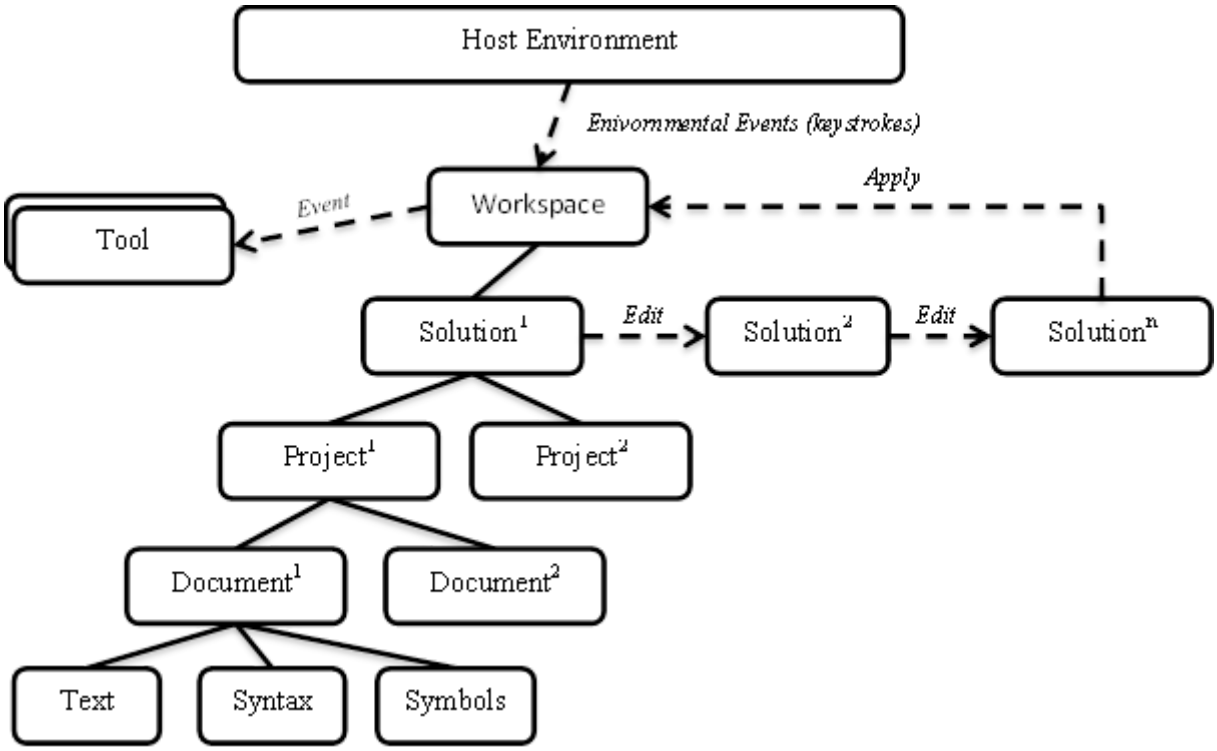
Même si un espace de travail change chaque fois qu'un utilisateur appuie sur une touche, vous pouvez utiliser le modèle de la solution en mode isolation.

Une solution est un modèle immuable des projets et des documents. Le modèle peut donc être partagé sans verrouillage ni duplication. Après avoir obtenu une instance de la solution à partir de la propriété Workspace.CurrentSolution, vous ne pouvez plus la changer. Toutefois, comme pour les arborescences de syntaxe et les compilations, vous pouvez modifier des solutions en créant d'autres instances basées sur des solutions existantes et en y apportant les changements souhaités. Pour répercuter vos changements dans l'espace de travail, vous devez explicitement réappliquer la solution modifiée dans l'espace de travail.

Un projet est une composante du modèle de solution immuable global. Il représente l'ensemble des documents du code source, des options d'analyse et de compilation, et des références d'assembly et entre projets. À partir d'un projet, vous pouvez accéder à la compilation correspondante sans avoir besoin de déterminer les dépendances du projet ou d'analyser des fichiers sources.

Un document est une autre composante du modèle de solution immuable global. Un document représente un fichier source unique à partir duquel vous pouvez accéder au texte du fichier, à l'arborescence de syntaxe et au modèle sémantique.

Le diagramme suivant illustre les relations entre l'espace de travail et l'environnement hôte, les outils, et la façon dont les modifications sont effectuées.



Résumé

Roslyn expose un ensemble d'API de compilateur et d'API d'espaces de travail. Ces API fournissent des informations détaillées sur votre code source et offrent une haute

fidélité avec les langages Visual Basic et C#. Le SDK .NET Compiler Platform réduit considérablement les difficultés inhérentes à la création d'applications et d'outils axés sur le code. Il offre de nombreuses possibilités d'innovation dans différents champs, notamment la métaprogrammation, la génération et la transformation de code, l'utilisation interactive des langages C# et Visual Basic et l'incorporation de C# et de Visual Basic dans des langages propres au domaine.

Explorer le code avec le visualiseur de syntaxe Roslyn dans Visual Studio

Article • 10/05/2023

Cet article fournit une vue d'ensemble de l'outil visualiseur de syntaxe inclus dans le SDK .NET Compiler Platform (« Roslyn »). Le visualiseur de syntaxe est une fenêtre Outil qui vous permet d'inspecter et d'explorer des arborescences de syntaxe. C'est un outil essentiel pour comprendre les modèles du code que vous voulez analyser. Cet outil s'avère aussi une aide précieuse au débogage quand vous développez vos propres applications à l'aide du SDK .NET Compiler Platform (« Roslyn »). Ouvrez l'outil au moment où vous créez vos premiers analyseurs. Le visualiseur facilite la compréhension des modèles utilisés par les API. Vous pouvez également utiliser des outils comme SharpLab ou LINQPad pour inspecter du code et comprendre des arborescences de syntaxe.

Instructions d'installation - Visual Studio Installer

Vous pouvez rechercher **SDK .NET Compiler Platform** dans **Visual Studio Installer** de deux façons :

Vue Installation avec Visual Studio Installer - Charges de travail

SDK .NET Compiler Platform n'est pas sélectionné automatiquement dans le cadre de la charge de travail Développement d'extensions Visual Studio. Vous devez le sélectionner comme composant facultatif.

1. Exécutez **Visual Studio Installer**.
2. Sélectionnez **Modifier**
3. Choisissez la charge de travail **Développement d'extensions Visual Studio**.
4. Ouvrez le nœud **Développement d'extensions Visual Studio** dans l'arborescence résumée.
5. Cochez la case pour **SDK .NET Compiler Platform**. Il se trouve en dernier sous les composants facultatifs.

Si vous le souhaitez, vous pouvez ajouter **l'éditeur DGML** pour afficher des graphes dans le visualiseur :

1. Ouvrez le nœud **Composants individuels** dans l'arborescence résumée.
2. Cochez la case pour **Éditeur DGML**.

Onglet Installation avec Visual Studio Installer - Composants individuels

1. Exécutez **Visual Studio Installer**.
2. Sélectionnez **Modifier**
3. Sélectionnez l'onglet **Composants individuels**.
4. Cochez la case pour **SDK .NET Compiler Platform**. Il se trouve en haut de la liste, sous la section **Compilateurs, outils de génération et runtimes**.

Si vous le souhaitez, vous pouvez ajouter **l'éditeur DGML** pour afficher des graphes dans le visualiseur :

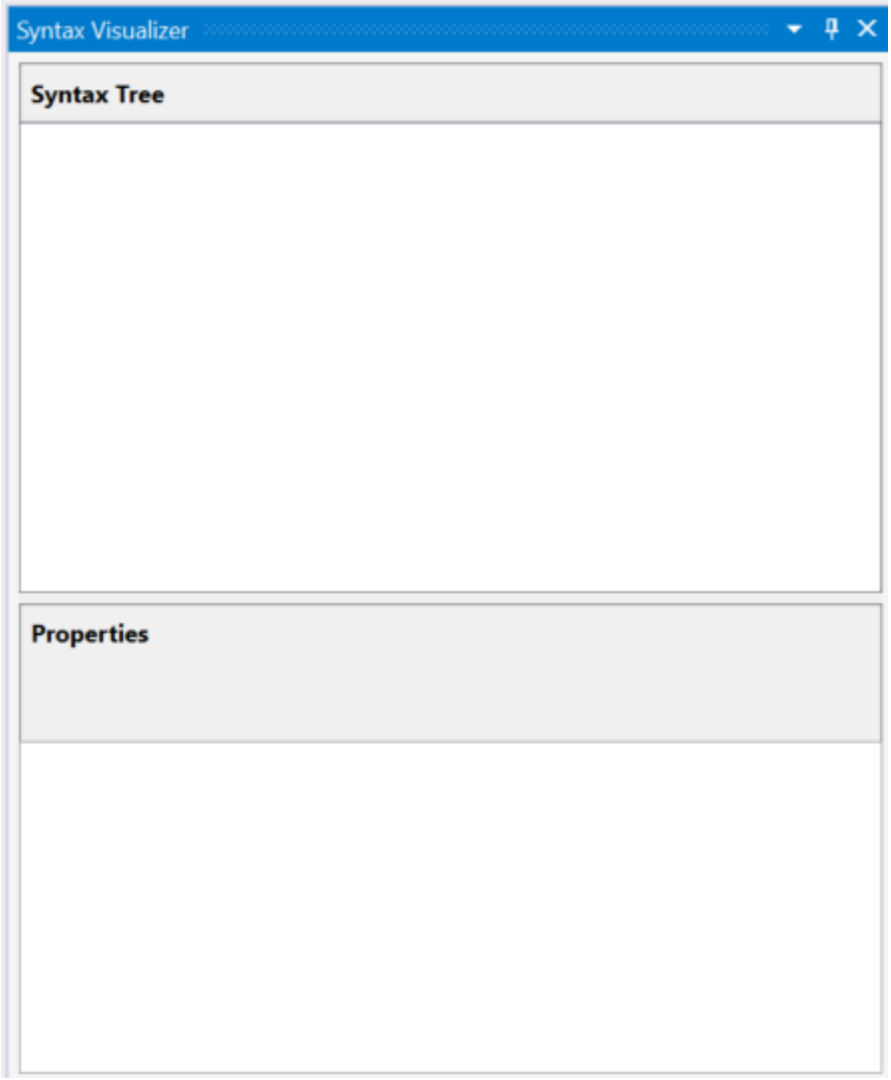
1. Cochez la case pour **Éditeur DGML**. Il se trouve sous la section **Outils de code**.

Familiarisez-vous avec les concepts utilisés dans le SDK .NET Compiler Platform en lisant l'article de vue d'ensemble. Celui-ci présente les arborescences de syntaxe, les nœuds, les jetons et les trivia.

Visualiseur de syntaxe

Le **Syntax Visualizer** permet d'inspecter l'arborescence de syntaxe du fichier de code C# ou Visual Basic dans la fenêtre actuelle de l'éditeur actif au sein de l'environnement de développement intégré Visual Studio. Pour lancer le visualiseur, vous pouvez cliquer sur **Affichage**>**Autres fenêtres**>**Visualiseur de syntaxe**. Vous pouvez également utiliser la barre d'outils **Lancement rapide** dans le coin supérieur droit. Tapez « syntaxe » pour faire apparaître la commande permettant d'ouvrir le **visualiseur de syntaxe**.

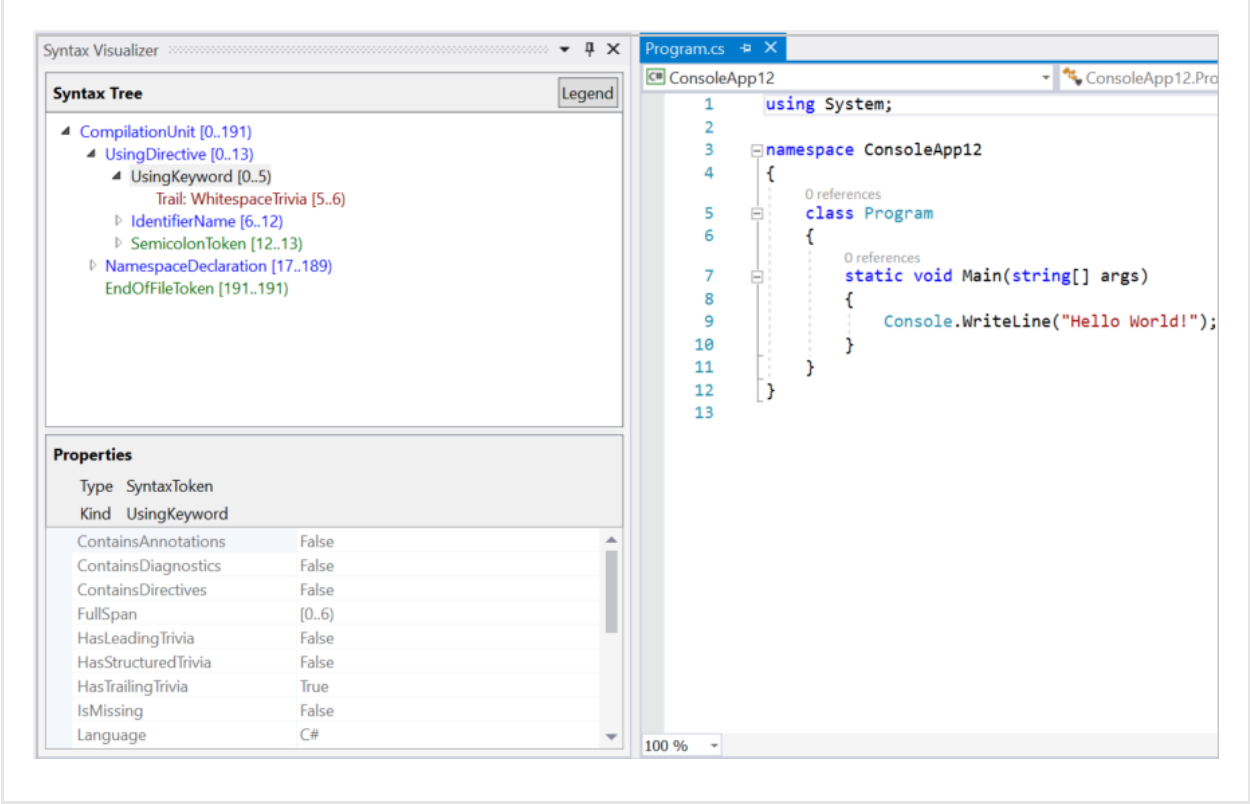
Cette commande ouvre ce dernier dans une fenêtre Outil flottante. Si aucune fenêtre d'éditeur de code n'est ouverte, l'affichage est vide, comme l'illustre la figure suivante.



Ancrez cette fenêtre Outil à un emplacement pratique dans Visual Studio, comme le côté gauche. Le visualiseur indique des informations sur le fichier de code actuel.

Créez un projet à l'aide de la commande **Fichier**>**Nouveau projet**. Vous pouvez créer un projet Visual Basic ou C#. Quand Visual Studio ouvre le fichier de code principal de ce projet, le visualiseur en présente l'arborescence de syntaxe. Vous pouvez ouvrir n'importe quel fichier C# ou Visual Basic existant dans cette instance Visual Studio pour que le visualiseur en présente l'arborescence de syntaxe. Si plusieurs fichiers de code sont ouverts dans Visual Studio, le visualiseur présente l'arborescence de syntaxe du fichier de code actif (celui qui a le focus clavier.)

C#



Comme l'illustrent les images précédentes, la fenêtre Outil du visualiseur présente l'arborescence de syntaxe en haut et une grille des propriétés en bas. La grille des propriétés présente les propriétés de l'élément actuellement sélectionné dans l'arborescence, notamment le *type* .NET et le *genre* (SyntaxKind) de l'élément.

Les arborescences de syntaxe comprennent trois types d'éléments : des *nœuds*, des *jetons* et des *trivia*. Pour en savoir plus sur ces types, lisez l'article sur l'utilisation de la syntaxe. Les éléments de chaque type sont représentés à l'aide d'une couleur différente. Cliquez sur le bouton « Légende » pour obtenir une vue d'ensemble des couleurs utilisées.

Chaque élément de l'arborescence présente également sa propre **étendue**. L'**étendue** correspond aux index (position de départ et de fin) de ce nœud dans le fichier texte. Dans le précédent exemple C#, le jeton « UsingKeyword [0..5) » a une **étendue** de cinq caractères de large, [0..5). La notation « [..) » signifie que l'index de départ fait partie de l'étendue, tandis que l'index de fin n'en fait pas partie.

Il existe deux façons de naviguer dans l'arborescence :

- Développez des éléments dans l'arborescence ou cliquez dessus. Le visualiseur sélectionne automatiquement le texte correspondant à l'étendue de cet élément dans l'éditeur de code.
- Cliquez sur le texte ou sélectionnez-le dans l'éditeur de code. Dans le précédent exemple Visual Basic, si vous sélectionnez la ligne contenant « Module Module1 » dans l'éditeur de code, le visualiseur accède automatiquement au nœud ModuleStatement correspondant dans l'arborescence.

Le visualiseur met en surbrillance l'élément dans l'arborescence dont l'étendue correspond le mieux à celle du texte sélectionné dans l'éditeur.

Le visualiseur actualise l'arborescence pour qu'elle corresponde aux modifications apportées dans le fichier de code actif. Ajoutez un appel à `Console.WriteLine()` dans `Main()`. Pendant la frappe, le visualiseur actualise l'arborescence.

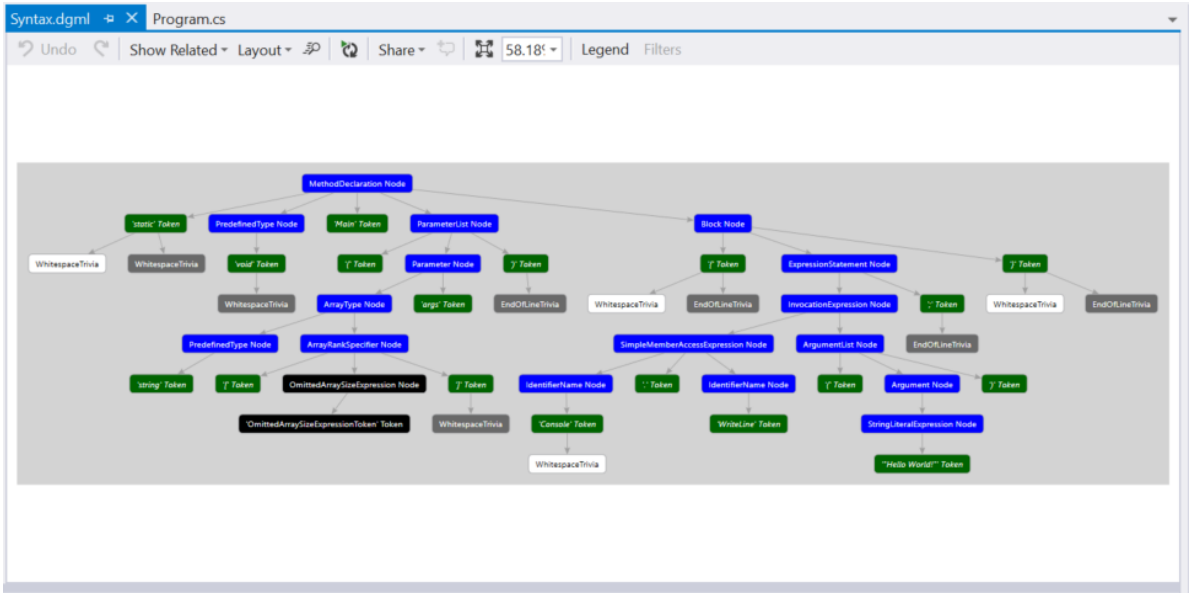
Interrompez la frappe une fois vous avez tapé `Console.`. L'arborescence colore certains éléments en rose. À ce stade, il existe des erreurs (également appelées « Diagnostics ») dans le code tapé. Ces erreurs sont liées à des nœuds, des jetons et des trivia dans l'arborescence de syntaxe. Le visualiseur indique les éléments auxquels sont liées des erreurs en colorant leur arrière-plan en rose. Vous pouvez inspecter les erreurs détectées sur un élément coloré en rose en plaçant le curseur au-dessus de lui. Le visualiseur montre uniquement les erreurs syntaxiques (erreurs liées à la syntaxe du code tapé) ; il n'indique pas les erreurs sémantiques.

Graphiques de syntaxe

Cliquez avec le bouton droit sur un élément dans l'arborescence, puis cliquez sur **Afficher un graphique de syntaxe orienté**.

C#

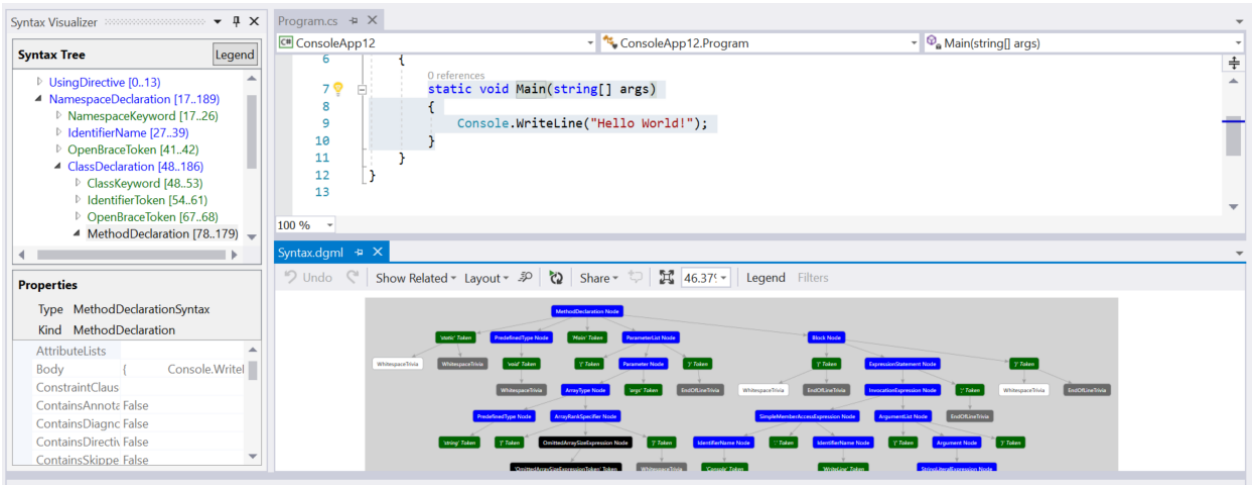
Le visualiseur affiche une représentation graphique de la sous-arborescence dont la racine est l'élément sélectionné. Essayez ces étapes pour le nœud **MethodDeclaration** correspondant à la méthode `Main()` dans l'exemple C#. Le visualiseur affiche un graphique de syntaxe qui ressemble à celui-ci :



La visionneuse de graphique de syntaxe comporte une option qui permet d'afficher la légende de son jeu de couleurs. Vous pouvez également placer le pointeur de la souris au-dessus d'éléments individuels dans le graphique de syntaxe pour afficher les propriétés correspondantes.

Vous pouvez afficher des graphiques de syntaxe pour différents éléments dans l'arborescence à plusieurs reprises. Ces graphiques s'affichent toujours dans la même fenêtre au sein de Visual Studio. Ancrez cette fenêtre à un emplacement pratique dans Visual Studio pour ne pas avoir à basculer d'un onglet à un autre afin d'afficher un nouveau graphique de syntaxe. En bas, sous les fenêtres de l'éditeur de code, s'avère souvent un emplacement pratique.

Voici la disposition d'ancrage à utiliser avec la fenêtre Outil du visualiseur et la fenêtre de graphique de syntaxe :



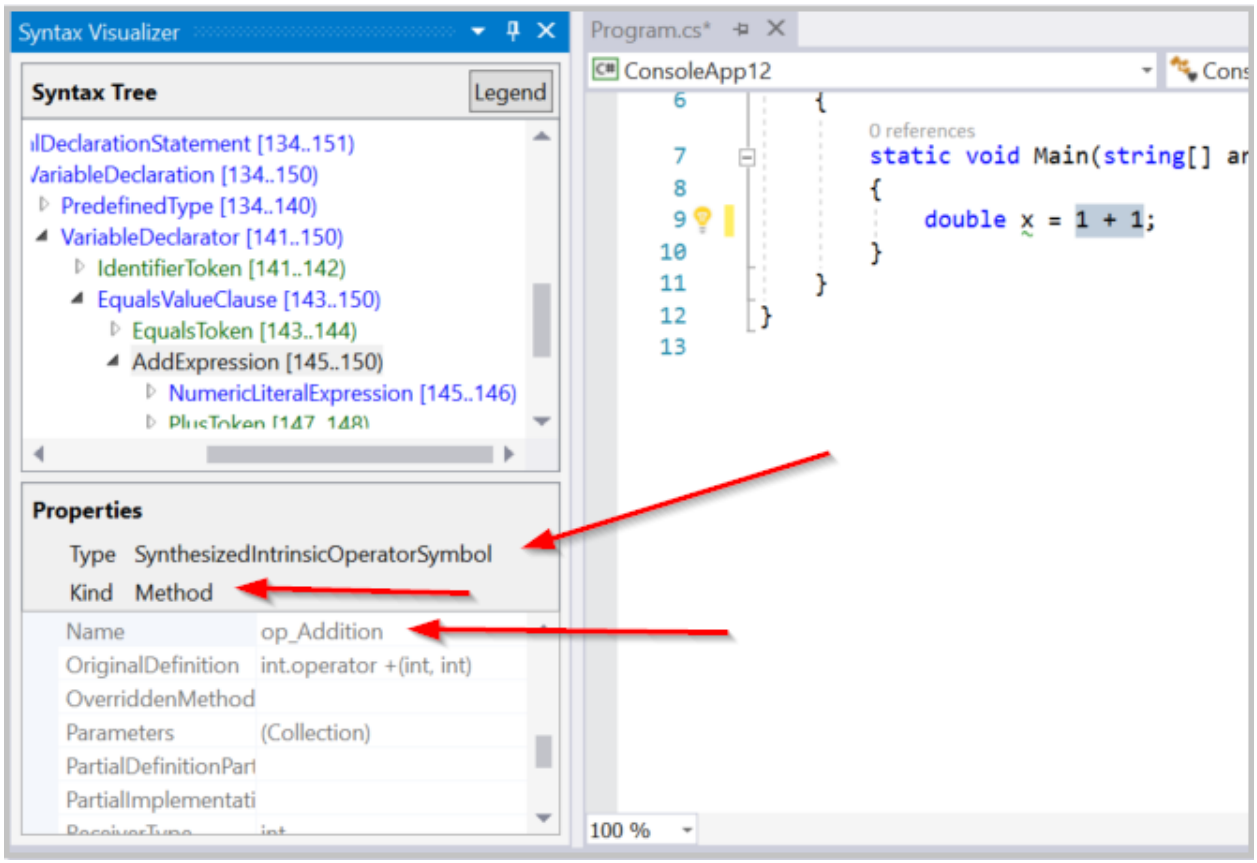
Une autre option consiste à placer la fenêtre de graphique de syntaxe sur un deuxième écran, dans une configuration à double écran.

Inspection de la sémantique

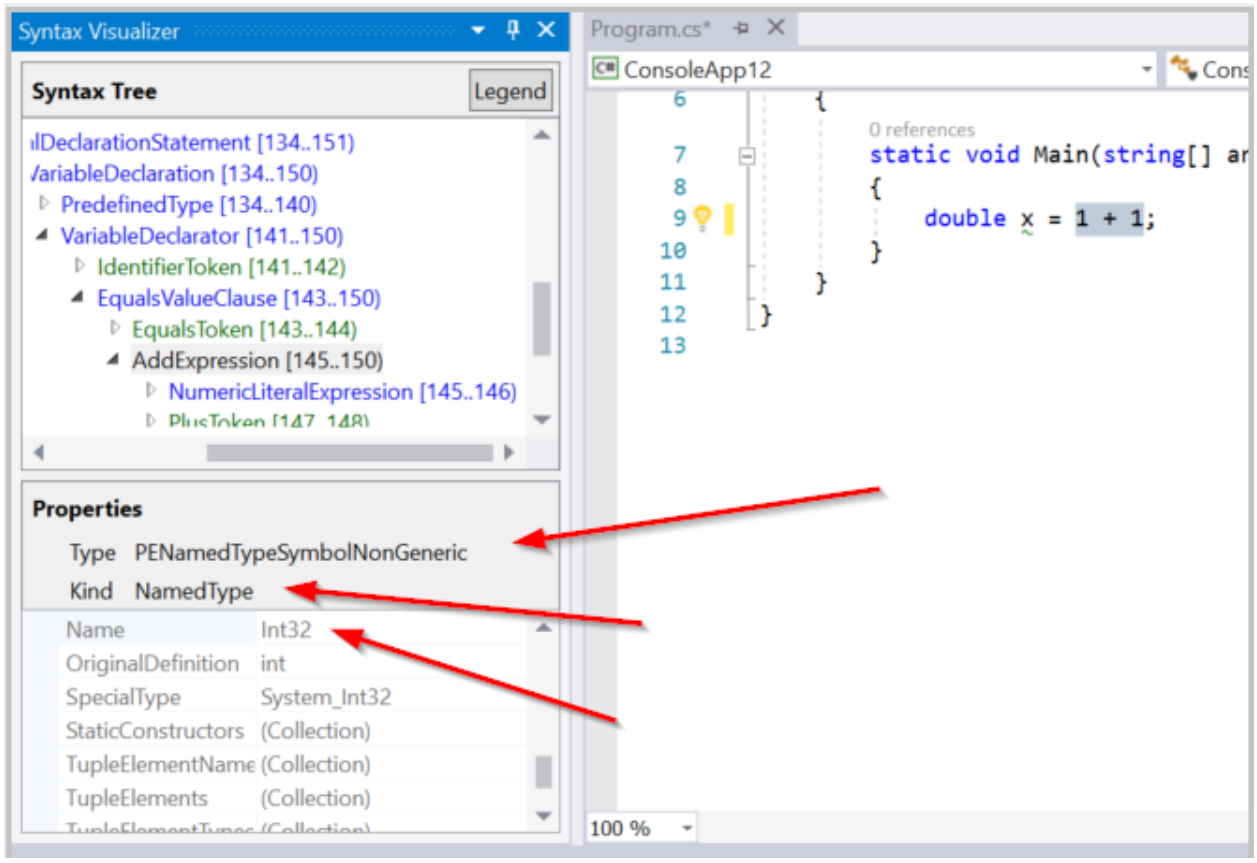
Le visualiseur de syntaxe permet une inspection rudimentaire des symboles et des informations sémantiques. Tapez `double x = 1 + 1;` au sein de Main() dans l'exemple C#. Ensuite, sélectionnez l'expression `1 + 1` dans la fenêtre de l'éditeur de code. Le visualiseur met en surbrillance le nœud **AddExpression** dans le visualiseur. Cliquez avec le bouton droit sur ce nœud **AddExpression**, puis cliquez sur **Afficher le symbole (le cas échéant)**. Notez que la plupart des éléments de menu présentent le qualificateur « le cas échéant ». Le visualiseur de syntaxe inspecte les propriétés d'un nœud, notamment celles qui ne sont éventuellement pas présentes pour tous les nœuds.

La grille des propriétés se met à jour dans le visualiseur comme l'illustre la figure suivante : le symbole de l'expression est un **SynthesizedIntrinsicOperatorSymbol** avec

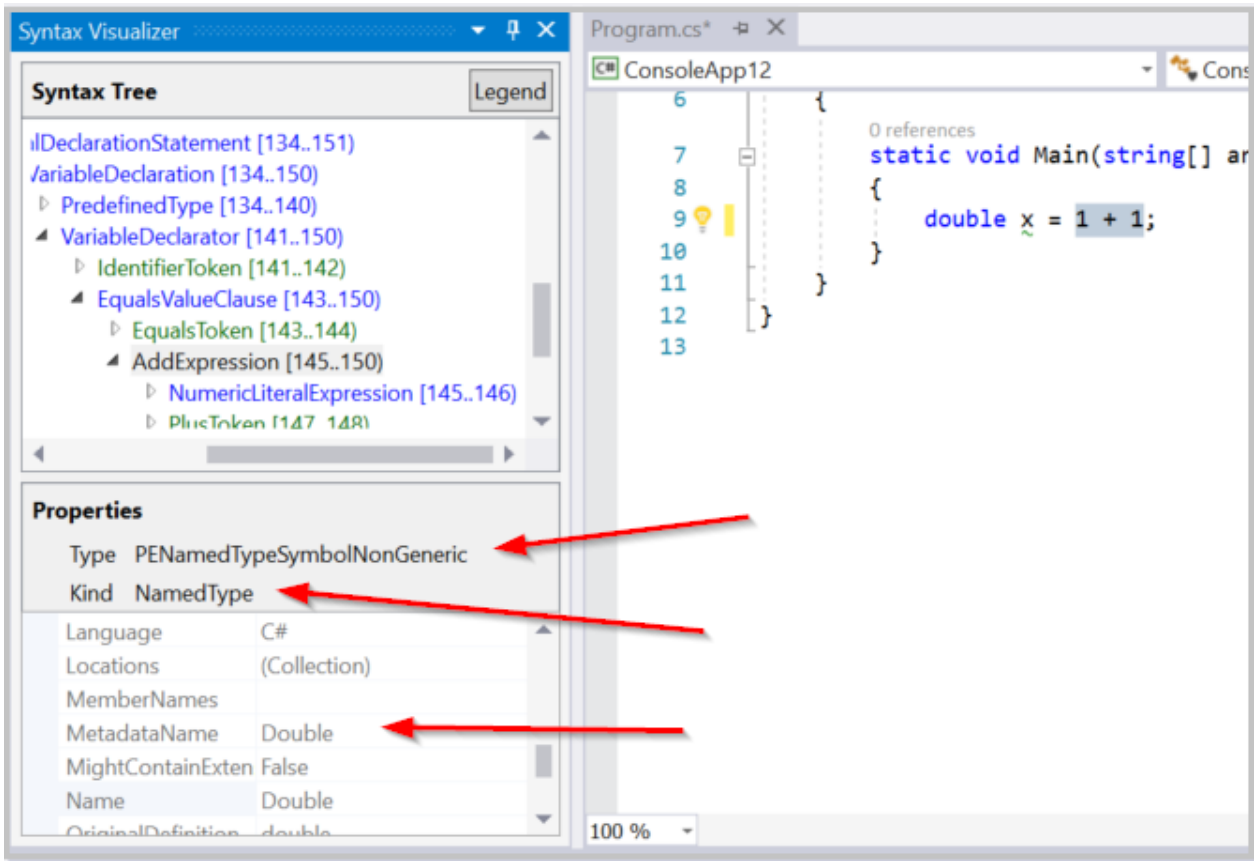
Kind = Method.



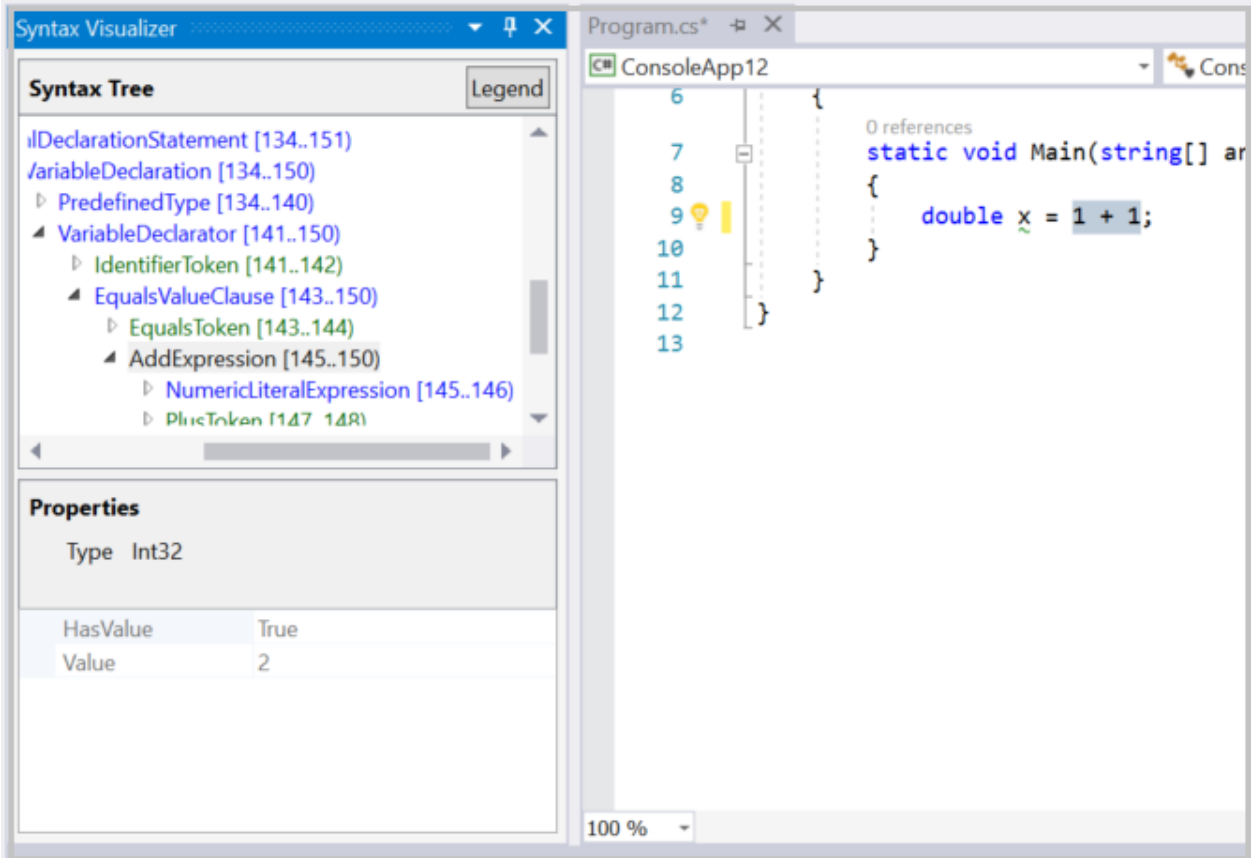
Essayez **Afficher TypeSymbol (le cas échéant)** pour le même nœud **AddExpression**. La grille des propriétés se met à jour dans le visualiseur comme l'illustre la figure suivante, en indiquant que le type de l'expression sélectionnée est `Int32`.



Essayez **Afficher un TypeSymbol converti (le cas échéant)** pour le même nœud **AddExpression**. La grille des propriétés se met à jour en indiquant que, bien que le type de l'expression soit `Int32`, le type converti de l'expression est `Double` comme l'illustre la figure suivante. Ce nœud inclut des informations de symbole de type converti, car l'expression `Int32` se produit dans un contexte où elle doit être convertie en `Double`. Cette conversion satisfait au type `Double` spécifié pour la variable `x` du côté gauche de l'opérateur d'assignation.



Enfin, essayez **Afficher la valeur constante (le cas échéant)** pour le même nœud **AddExpression**. La grille des propriétés montre que la valeur de l'expression est une constante au moment de la compilation de valeur `2`.



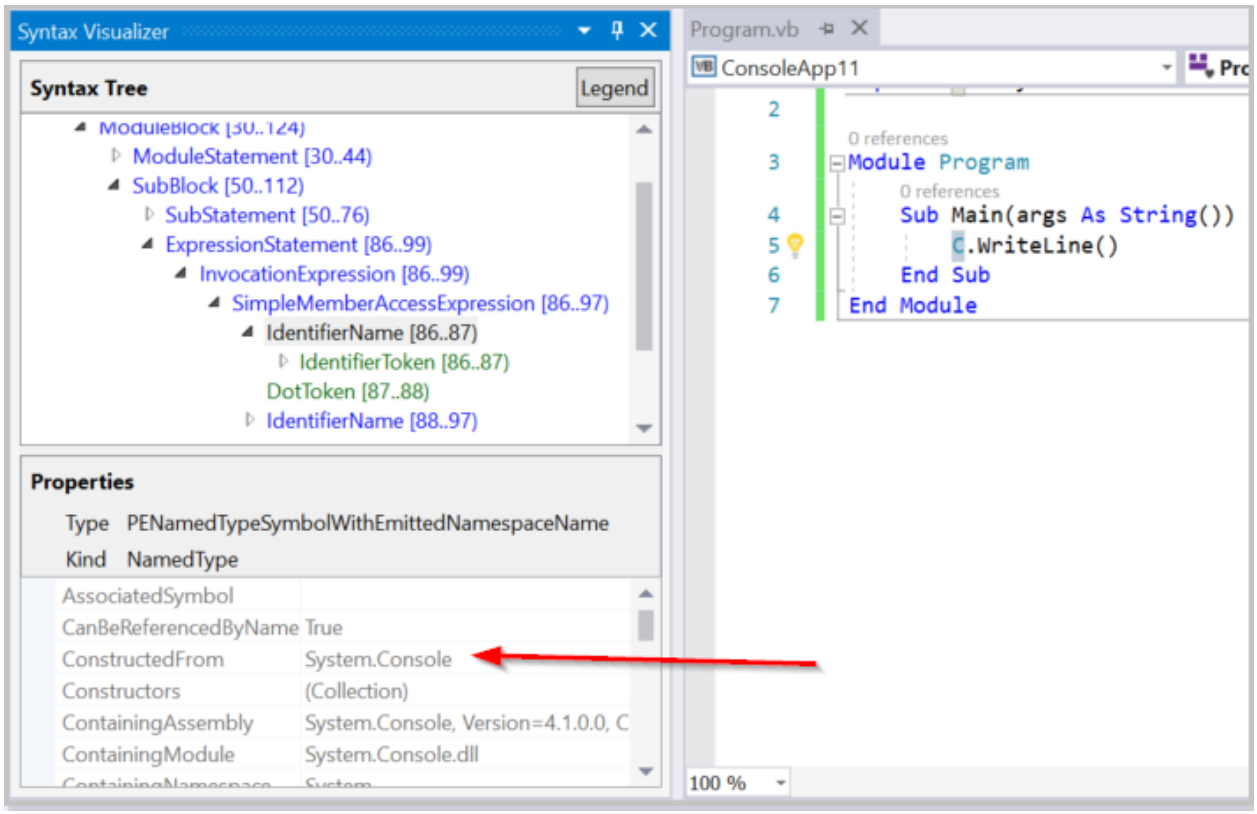
L'exemple précédent peut également être répliqué dans Visual Basic. Tapez `Dim x As Double = 1 + 1` dans un fichier Visual Basic. Sélectionnez l'expression `1 + 1` dans la fenêtre de l'éditeur de code. Le visualiseur met en surbrillance le nœud **AddExpression** correspondant dans le visualiseur. Répétez les étapes précédentes pour ce nœud **AddExpression** et vous obtiendrez normalement des résultats identiques.

Examinez plus de code dans Visual Basic. Mettez à jour votre fichier Visual Basic principal avec le code suivant :

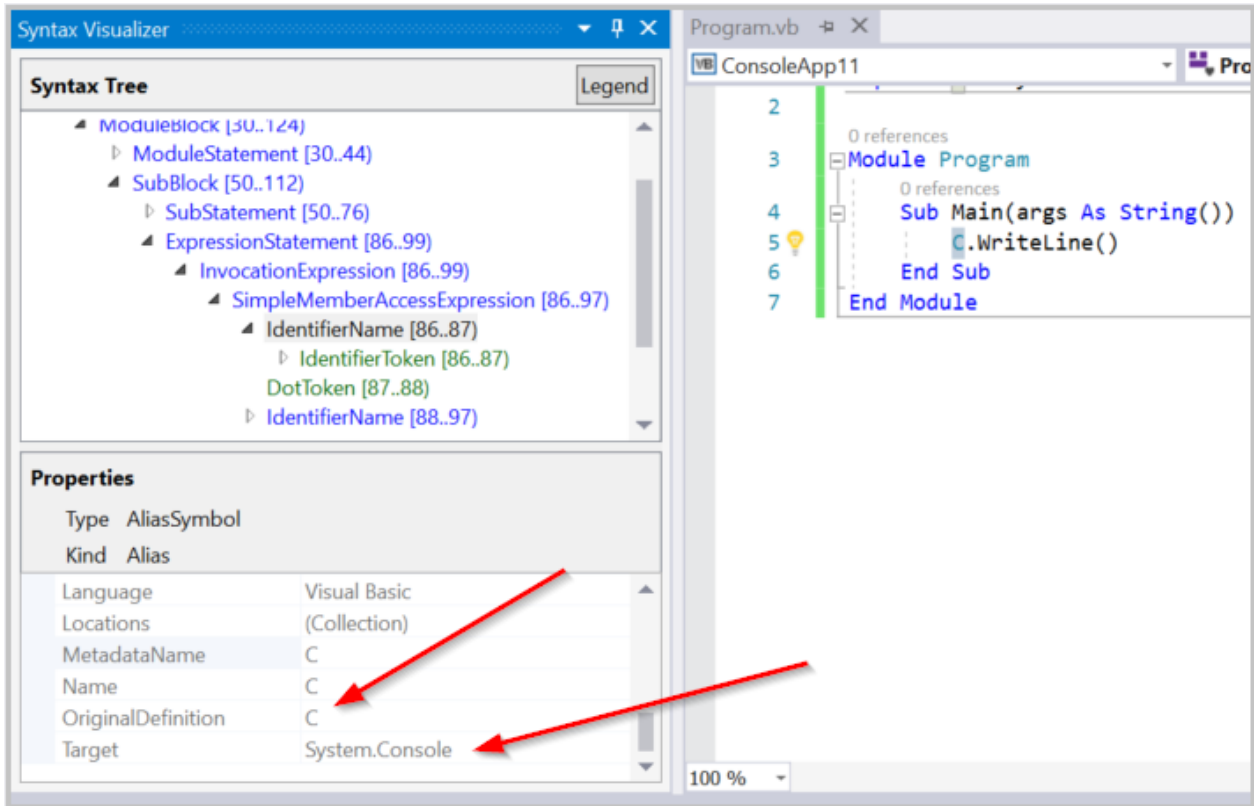
VB

```
Imports C = System.Console

Module Program
    Sub Main(args As String())
        C.WriteLine()
    End Sub
End Module
```

Ce code présente un alias nommé `C` qui se mappe sur le type `System.Console` en haut du fichier et utilise cet alias dans `Main()`. Sélectionnez l'utilisation de cet alias, le `C` dans `C.WriteLine()`, à l'intérieur de la méthode `Main()`. Le visualiseur sélectionne le nœud **IdentifierName** correspondant. Cliquez avec le bouton droit sur ce nœud et cliquez sur **Afficher le symbole (le cas échéant)**. La grille des propriétés indique que cet identificateur est lié au type `System.Console` comme l'illustre la figure suivante :

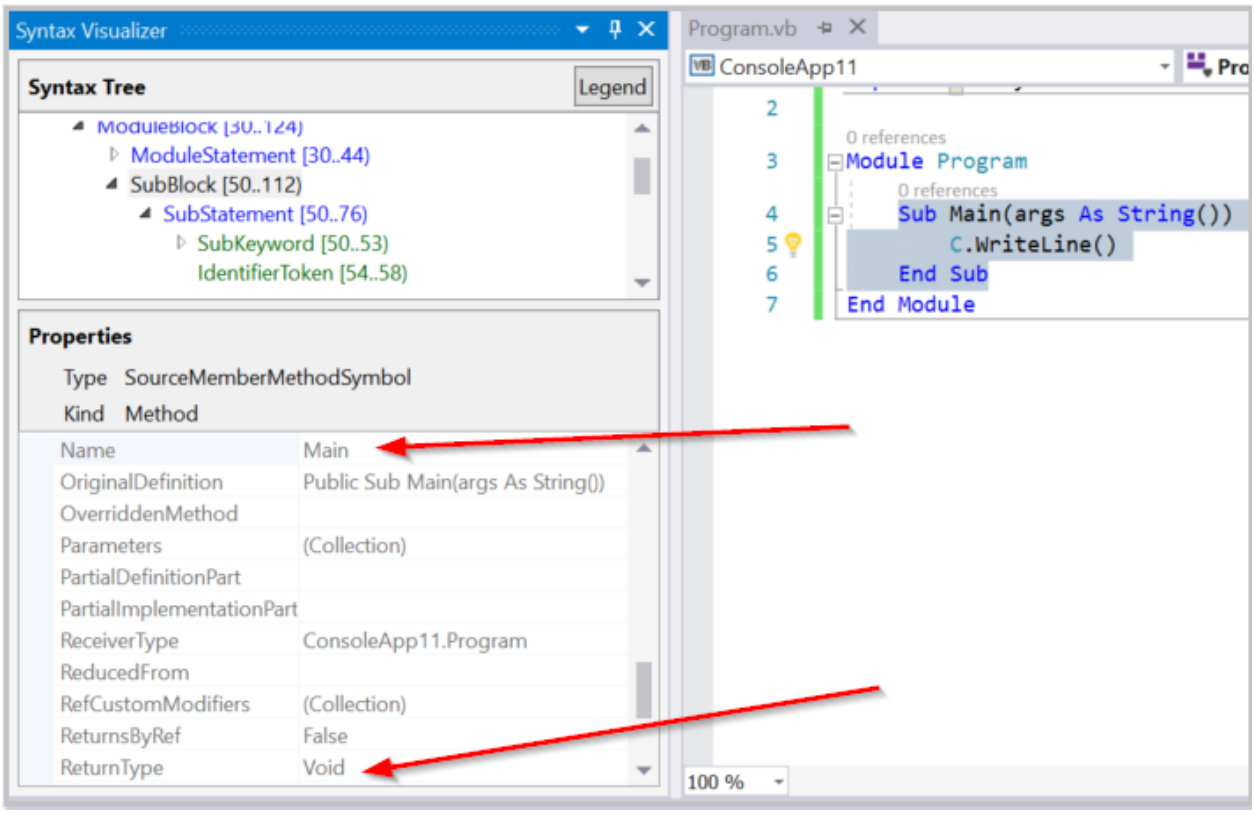


Essayez **Afficher AliasSymbol (le cas échéant)** pour le même nœud **IdentifierName**. La grille des propriétés indique que l'identificateur est un alias du nom de `C` qui est lié à la cible `System.Console`. En d'autres termes, la grille des propriétés fournit des informations sur l'**AliasSymbol** correspondant à l'identificateur `C`.



Examinez le symbole correspondant à un type, une méthode, une propriété qui ont été déclarés. Sélectionnez le nœud correspondant dans le visualiseur et cliquez sur **Afficher**

le symbole (le cas échéant). Sélectionnez la méthode `Sub Main()`, corps de la méthode compris. Cliquez sur **Afficher le symbole (le cas échéant)** pour le nœud **SubBlock** correspondant dans le visualiseur. La grille des propriétés montre que le **MethodSymbol** de ce **SubBlock** porte le nom `Main` avec un type de retour `Void`.



Les exemples Visual Basic ci-dessus sont facilement réplicables en C#. Tapez `using C = System.Console;` à la place de `Imports C = System.Console` pour l'alias. Les étapes précédentes en C# produisent des résultats identiques dans la fenêtre du visualiseur.

Les opérations d'inspection sémantique sont uniquement disponibles sur des nœuds. Elles ne le sont pas sur des jetons ni des trivia. Tous les nœuds ont des informations sémantiques intéressantes à inspecter. Quand un nœud n'a pas d'informations sémantiques intéressantes, le fait de cliquer sur **Afficher *Symbol (le cas échéant)** fait apparaître une grille des propriétés vide.

Pour en savoir plus sur les API permettant d'effectuer une analyse sémantique, consultez le document de vue d'ensemble de l'utilisation de la sémantique.

Fermeture du visualiseur de syntaxe

Vous pouvez fermer la fenêtre du visualiseur quand vous ne l'utilisez pas pour examiner le code source. Le visualiseur de syntaxe met à jour son affichage au fur et à mesure que vous parcourez le code, en modifiant la source. Cela peut s'avérer perturbant quand vous ne l'utilisez pas.

Générateurs de source

Article • 25/06/2023

Cet article fournit une vue d'ensemble des Générateurs de sources inclus dans le SDK .NET Compiler Platform (« Roslyn »). Les générateurs de source permettent aux développeurs C# d'inspecter le code utilisateur à mesure qu'il est compilé. Le générateur peut créer de nouveaux fichiers sources C# à la volée qui sont ajoutés à la compilation de l'utilisateur. De cette façon, vous disposez d'un code qui s'exécute pendant la compilation. Il inspecte votre programme pour produire des fichiers sources supplémentaires compilés avec le reste de votre code.

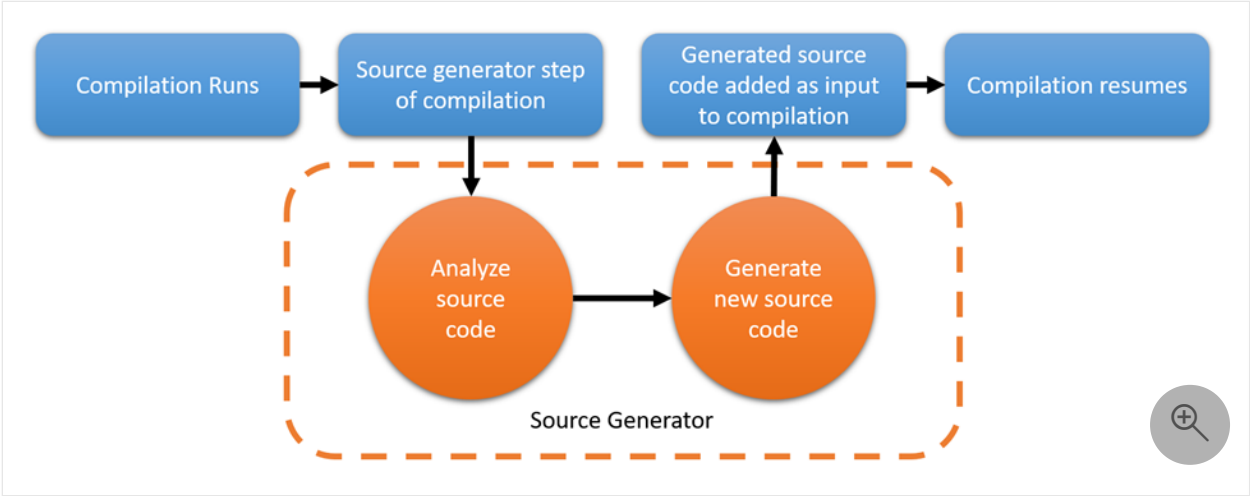
Un générateur de source est un nouveau type de composant que les développeurs C# peuvent écrire et qui vous permet d'effectuer deux choses importantes :

1. Récupérer un *objet de compilation* qui représente tout le code utilisateur en cours de compilation. Cet objet peut être inspecté et vous pouvez écrire du code qui fonctionne avec la syntaxe et les modèles sémantiques pour le code en cours de compilation, comme avec les analyseurs d'aujourd'hui.

2. Générez des fichiers sources C# qui peuvent être ajoutés à un objet de compilation pendant la compilation. Autrement dit, vous pouvez fournir du code source supplémentaire comme entrée dans une compilation pendant la compilation.

Lorsqu'ils sont combinés, ces deux éléments rendent les générateurs de source si utiles. Vous pouvez inspecter le code utilisateur avec toutes les métadonnées enrichies générées par le compilateur pendant la compilation. Votre générateur émet ensuite du code C# dans la même compilation, qui est basée sur les données que vous avez analysées. Si vous connaissez les analyseurs Roslyn, vous pouvez considérer les générateurs de source comme des analyseurs capables d'émettre du code source C#.

Les générateurs de source s'exécutent en tant que phase de compilation visualisées ci-dessous :



Un générateur de source est un assembly .NET Standard 2.0 chargé par le compilateur avec tous les analyseurs. Il est utilisable dans les environnements où les composants .NET Standard peuvent être chargés et exécutés.

> **ⓘ Important**
>
> Actuellement, seuls les assemblys .NET Standard 2.0 peuvent être utilisés comme générateurs de source.

Scénarios courants

Il existe trois approches générales pour inspecter le code utilisateur et générer des informations ou du code basé sur l'analyse utilisée par les technologies d'aujourd'hui :

- Réflexion du runtime.
- Jongler les tâches MSBuild.
- Tissage en langage intermédiaire (IL) (non abordé dans cet article).

Les générateurs de source peuvent être une amélioration par rapport à chaque approche.

Réflexion du runtime

La réflexion d'exécution est une technologie puissante qui a été ajoutée à .NET il y a longtemps. Il existe d'innombrables scénarios pour l'utiliser. Un scénario courant consiste à effectuer une analyse du code utilisateur lors du démarrage d'une application et à utiliser ces données pour générer des éléments.

Par exemple, ASP.NET Core utilise la réflexion lorsque votre service web s'exécute pour la première fois, afin de découvrir les constructions que vous avez définies et de sorte

qu'il puisse « brancher » des éléments tels que des contrôleurs et des pages Razor. Bien que cela vous permette d'écrire du code simple avec des abstractions puissantes, il s'accompagne d'une pénalité de performances au moment de l'exécution : lorsque votre service web ou votre application démarre pour la première fois, il/elle ne peut accepter aucune demande tant que tout le code de réflexion du runtime qui découvre des informations sur votre code n'est pas terminé. Bien que cette pénalité de performance ne soit pas énorme, il s'agit d'un coût fixe que vous ne pouvez pas améliorer vous-même dans votre propre application.

Avec un générateur de source, la phase de découverte du contrôleur au démarrage peut se produire au moment de la compilation. Un générateur peut analyser votre code source et émettre le code dont il a besoin pour « brancher » votre application. L'utilisation de générateurs de source peut accélérer les temps de démarrage, car une action se produisant au moment de l'exécution aujourd'hui peut être poussée au moment de la compilation.

Jongler les tâches MSBuild

Les générateurs de source peuvent améliorer les performances autrement que par la réflexion au moment de l'exécution, afin de découvrir aussi des types. Certains scénarios impliquent d'appeler la tâche C# MSBuild (appelée CSC) plusieurs fois, afin qu'elles puissent inspecter les données d'une compilation. Comme vous pouvez l'imaginer, appeler plusieurs fois le compilateur, affecte le temps total nécessaire à la création de votre application. Nous examinons comment les générateurs de sources peuvent être utilisés pour ne pas avoir à jongler les tâches MSBuild comme celle-ci, car les générateurs de source n'offrent pas seulement certains avantages en matière de performances, mais permettent également aux outils de fonctionner au bon niveau d'abstraction.

Une autre fonctionnalité que les générateurs de sources peuvent offrir est l'évitement de l'utilisation de certaines API « typées par chaîne », telles que la façon dont le routage ASP.NET Core fonctionne entre les contrôleurs et les pages Razor. Avec un générateur de source, le routage peut être fortement typé avec les chaînes nécessaires générées comme détail au moment de la compilation. Ceci réduirait le nombre de fois où un littéral de chaîne mal typé conduit à une requête qui n'atteint pas le contrôleur correct.

Démarrage avec les générateurs de source

Dans ce guide, vous allez explorer la création d'un générateur de source à l'aide de l'API IIncrementalGenerator.

1. Créer une application console .NET. Cet exemple utilise .NET 7.

2. Remplacez la classe `Program` par le code suivant : Le code suivant n'utilise pas d'instructions de niveau supérieur. Le formulaire classique est requis, car ce premier générateur de source écrit une méthode partielle dans cette classe `Program` :

   C#

   ```
   namespace ConsoleApp;

   partial class Program
   {
       static void Main(string[] args)
       {
           HelloFrom("Generated Code");
       }

       static partial void HelloFrom(string name);
   }
   ```

   > ⓘ **Notes**
   >
   > Vous pouvez exécuter cet exemple tel qu'il est, mais rien ne se passera pour l'instant.

3. Ensuite, nous allons créer un projet de générateur de source qui implémentera l'équivalent de la méthode `partial void HelloFrom`.

4. Créez un projet de bibliothèque standard .NET qui cible le moniker de framework cible (TFM) `netstandard2.0`. Ajoutez les packages NuGet *Microsoft.CodeAnalysis.Analyzers* et *Microsoft.CodeAnalysis.CSharp* :

   XML

   ```
   <Project Sdk="Microsoft.NET.Sdk">

     <PropertyGroup>
       <TargetFramework>netstandard2.0</TargetFramework>
       <LangVersion>11.0</LangVersion>
       <EnforceExtendedAnalyzerRules>true</EnforceExtendedAnalyzerRules>
     </PropertyGroup>

     <ItemGroup>
       <PackageReference Include="Microsoft.CodeAnalysis.CSharp"
   Version="4.6.0" PrivateAssets="all" />
       <PackageReference Include="Microsoft.CodeAnalysis.Analyzers"
   ```

```
Version="3.3.4" PrivateAssets="all" />
  </ItemGroup>

</Project>
```

> **Conseil**
>
> Le projet de générateur de source doit cibler le TFM `netstandard2.0`, sinon il ne fonctionnera pas.

5. Créez un nouveau fichier C# nommé *HelloSourceGenerator.cs* qui spécifie votre propre générateur de source comme suit :

C#

```
using Microsoft.CodeAnalysis;

namespace SourceGenerator;

public sealed class HelloSourceGenerator : IIncrementalGenerator
{
    public void Initialize(IncrementalGeneratorInitializationContext
context)
    {
        var compilationProvider = context.CompilationProvider;

        context.RegisterSourceOutput(
            compilationProvider,
            static (context, compilation) =>
        {
            // Code generation goes here
        });
    }
}
```

Un générateur de source doit à la fois implémenter l'interface Microsoft.CodeAnalysis.IIncrementalGenerator et disposer de Microsoft.CodeAnalysis.GeneratorAttribute.

1. Remplacez le contenu de la méthode IIncrementalGenerator.Initialize par l'implémentation suivante :

C#

```
using Microsoft.CodeAnalysis;

namespace SourceGenerator;
```

```
[Generator]
public sealed class HelloSourceGenerator : IIncrementalGenerator
{
    public void Initialize(IncrementalGeneratorInitializationContext
context)
    {
        var compilationIncrementalValue = context.CompilationProvider;

        context.RegisterSourceOutput(
            compilationIncrementalValue,
            static (context, compilation) =>
        {
            // Get the entry point method
            var mainMethod =
compilation.GetEntryPoint(context.CancellationToken);
            var typeName = mainMethod.ContainingType.Name;

            string source = $$"""
                // Auto-generated code
                namespace
{{mainMethod.ContainingNamespace.ToDisplayString()}};

                public static partial class {{typeName}}
                {
                    static partial void HelloFrom(string name) =>
                        Console.WriteLine($"Generator says: Hi from
'{name}'");
                }
                """;

            // Add the source code to the compilation
            context.AddSource($"{typeName}.g.cs", source);
        });
    }
}
```

La variable `compilationIncrementalValue` est utilisée pour stocker la valeur de modèle incrémentielle de la compilation qui est passée à la méthode `RegisterSourceOutput` de `context`. À partir du paramètre `compilation`, nous pouvons accéder au point d'entrée des compilations ou à la méthode `Main`. L'instance `mainMethod` est un IMethodSymbol, et elle représente une méthode ou un symbole de type méthode (y compris constructeur, destructeur, opérateur ou accesseur de propriété/événement). La méthode Microsoft.CodeAnalysis.Compilation.GetEntryPoint retourne le IMethodSymbol pour le point d'entrée du programme. D'autres méthodes vous permettent de trouver n'importe quel symbole de méthode dans un projet. À partir de cet objet, nous pouvons raisonner à propos de l'espace de noms contenant (le cas échéant) et le type. Le `source` de cet exemple est une chaîne interpolée qui modélise le

code source à générer, où les espaces interpolés sont remplis avec les informations d'espace de noms et de type contenues. Le `source` est ajouté à `context` avec un nom indicateur. Pour cet exemple, le générateur crée un fichier source généré qui contient une implémentation de la méthode `partial` dans l'application console. Vous pouvez écrire des générateurs de source pour ajouter n'importe quelle source de votre choix.

> **Conseil**
>
> Le paramètre `hintName` de la méthode **SourceProductionContext.AddSource** peut être n'importe quel nom unique. Il est courant de fournir une extension de fichier C# explicite telle que `".g.cs"` ou `".generated.cs"` pour le nom. Le nom de fichier permet d'identifier le fichier en tant que source générée.

2. Nous disposons maintenant d'un générateur fonctionnel, mais nous devons le connecter à notre application console. Modifiez le projet d'application console d'origine et ajoutez les éléments suivants, en remplaçant le chemin du projet par celui du projet .NET Standard que vous avez créé ci-dessus :

   XML

   ```xml
   <!-- Add this as a new ItemGroup, replacing paths and names
   appropriately -->
   <ItemGroup>
       <ProjectReference Include="..\PathTo\SourceGenerator.csproj"
                         OutputItemType="Analyzer"
                         ReferenceOutputAssembly="false" />
   </ItemGroup>
   ```

   Cette nouvelle référence n'est pas une référence de projet traditionnelle et doit être modifiée manuellement pour inclure les attributs `OutputItemType` et `ReferenceOutputAssembly`. Pour plus d'informations sur les attributs `OutputItemType` et `ReferenceOutputAssembly` de `ProjectReference`, consultez Éléments de projet MSBuild courants : ProjectReference.

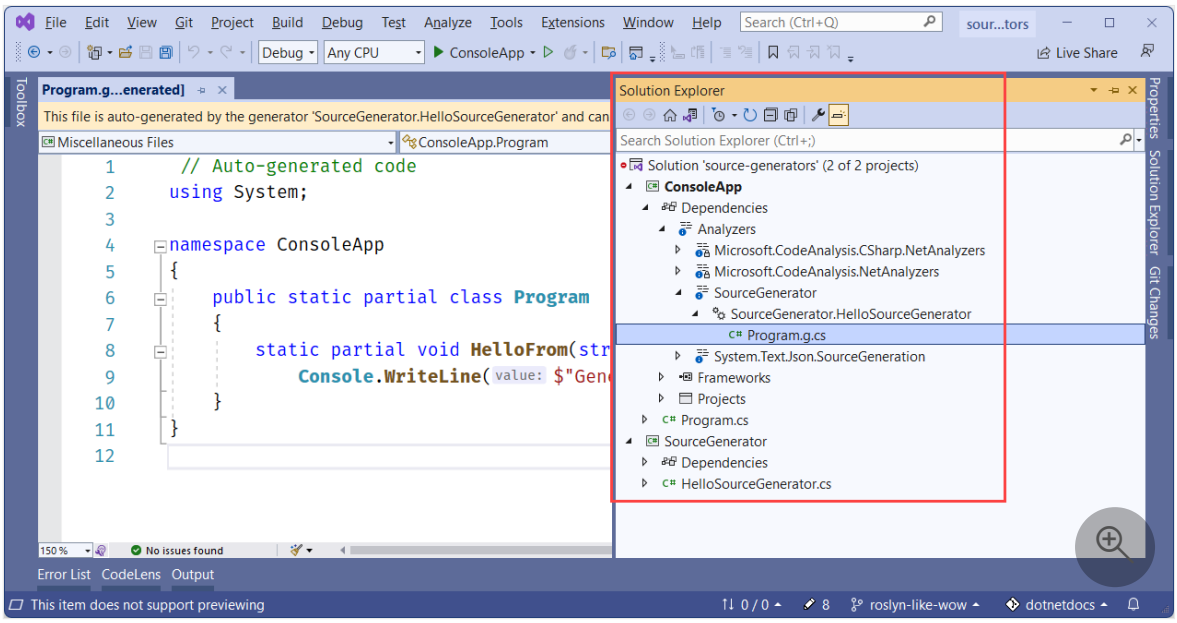
3. À présent, lorsque vous exécutez l'application console, vous devez voir que le code généré est exécuté et s'affiche à l'écran. L'application console elle-même n'implémente pas la méthode `HelloFrom`, mais plutôt sa source générée lors de la compilation à partir du projet Générateur de source. Le texte suivant est un exemple de sortie de l'application :

   Console

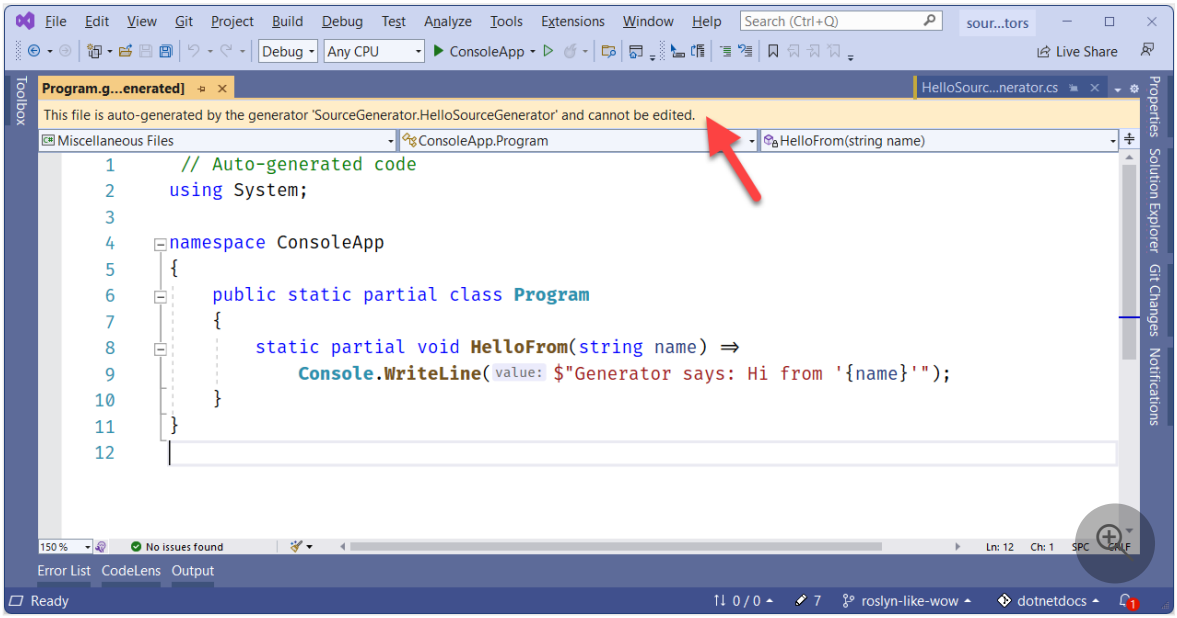
```
Generator says: Hi from 'Generated Code'
```

> **Notes**
>
> Vous devrez peut-être redémarrer Visual Studio pour voir IntelliSense et vous débarrasser des erreurs, car l'expérience d'outils est améliorée activement.

4. Si vous utilisez Visual Studio, vous pouvez voir les fichiers sources générés. Dans la fenêtre **Explorateur de solutions**, développez les **Dépendances**>**Analyzers**>**SourceGenerator**>**SourceGenerator.HelloSourceGeneratore**t double-cliquez sur le fichier *Program.g.cs*.



Lorsque vous ouvrez ce fichier généré, Visual Studio indique que le fichier est généré automatiquement et qu'il ne peut pas être modifié.



5. Vous pouvez également définir des propriétés de build pour enregistrer le fichier généré et contrôler l'emplacement où les fichiers générés sont stockés. Dans le fichier projet de l'application console, ajoutez l'élément `<EmitCompilerGeneratedFiles>` à un `<PropertyGroup>` et définissez sa valeur sur `true`. Regénérez votre projet. À présent, les fichiers générés sont créés sous *obj/Debug/net7.0/generated/SourceGenerator/SourceGenerator.HelloSourceGenerator*. Les composants du chemin d'accès sont mappés à la configuration de build, à la version cible de .Net Framework, au nom du projet du générateur de source et au nom de type complet du générateur. Vous pouvez choisir un dossier de sortie plus pratique en ajoutant l'élément `<CompilerGeneratedFilesOutputPath>` au fichier projet de l'application.

Étapes suivantes

Le cookbook des générateurs de source passe en revue certains de ces exemples avec quelques approches recommandées pour les résoudre. En outre, nous avons un ensemble d'échantillons disponibles sur GitHub que vous pouvez essayer vous-même.

Pour en savoir plus sur les générateurs de source, consultez les articles suivants :

- Document de conception générateurs de source
- Cookbook des générateurs de source

Bien démarrer avec l'analyse de la syntaxe

Article • 09/05/2023

Dans ce tutoriel, vous allez explorer **l'API Syntaxe**. L'API Syntaxe fournit un accès aux structures de données qui décrivent un programme C# ou Visual Basic. Ces structures de données ont suffisamment d'informations détaillées pour pouvoir représenter entièrement n'importe quel programme, quelle que soit sa taille. Ces structures peuvent décrire des programmes complets qui se compilent et s'exécutent correctement. Elles peuvent aussi décrire des programmes incomplets, au fil de leur écriture dans l'éditeur.

Pour permettre cette expression détaillée, les structures de données et les API qui composent l'API Syntaxe sont nécessairement complexes. Commençons par ce à quoi ressemble la structure de données pour le programme habituel « Hello World » :

C#

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Considérez le texte du programme précédent. Vous reconnaissez des éléments familiers. Le texte entier représente un même fichier source, ou une **unité de compilation**. Les trois premières lignes de ce fichier source sont des **directives using**. Le reste de la source est contenue dans une **déclaration d'espace de noms**. La déclaration d'espace de noms contient une **déclaration de classe** enfant. La déclaration de classe contient une **déclaration de méthode**.

L'API Syntaxe crée une arborescence dont la racine représente l'unité de compilation. Les nœuds de l'arborescence représentent les directives using, la déclaration d'espace de noms et tous les autres éléments du programme. L' arborescence continue jusqu'aux niveaux les plus bas : la chaîne « Hello World! » est un **jeton de littéral de chaîne** qui est

un descendant d'un **argument**. L'API Syntaxe fournit l'accès à la structure du programme. Vous pouvez rechercher des utilisations spécifiques du code, parcourir toute l'arborescence pour comprendre le code et créer des arborescences en modifiant l'arborescence existante.

Cette brève description fournit une vue d'ensemble des types d'informations accessibles avec l'API Syntaxe. L'API Syntaxe n'est rien de plus qu'une API formelle qui décrit les constructions de code familières que vous connaissez en C#. Les fonctionnalités complètes incluent des informations sur la façon dont le code est mis en forme, notamment les sauts de ligne, les espaces et l'indentation. Avec ces informations, vous pouvez représenter entièrement le code comme étant écrit et lu par des programmeurs ou par le compilateur. L'utilisation de cette structure vous permet d'interagir avec le code source à un niveau qui est vraiment significatif. Il ne s'agit plus de chaînes de texte, mais de données qui représentent la structure d'un programme C#.

Pour commencer, vous devez installer le kit **.NET Compiler Platform SDK** :

Instructions d'installation - Visual Studio Installer

Vous pouvez rechercher **SDK .NET Compiler Platform** dans **Visual Studio Installer** de deux façons :

Vue Installation avec Visual Studio Installer - Charges de travail

SDK .NET Compiler Platform n'est pas sélectionné automatiquement dans le cadre de la charge de travail Développement d'extensions Visual Studio. Vous devez le sélectionner comme composant facultatif.

1. Exécutez **Visual Studio Installer**.
2. Sélectionnez **Modifier**
3. Choisissez la charge de travail **Développement d'extensions Visual Studio**.
4. Ouvrez le nœud **Développement d'extensions Visual Studio** dans l'arborescence résumée.
5. Cochez la case pour **SDK .NET Compiler Platform**. Il se trouve en dernier sous les composants facultatifs.

Si vous le souhaitez, vous pouvez ajouter **l'éditeur DGML** pour afficher des graphes dans le visualiseur :

1. Ouvrez le nœud **Composants individuels** dans l'arborescence résumée.
2. Cochez la case pour **Éditeur DGML**.

Onglet Installation avec Visual Studio Installer - Composants individuels

1. Exécutez **Visual Studio Installer**.
2. Sélectionnez **Modifier**
3. Sélectionnez l'onglet **Composants individuels**.
4. Cochez la case pour **SDK .NET Compiler Platform**. Il se trouve en haut de la liste, sous la section **Compilateurs, outils de génération et runtimes**.

Si vous le souhaitez, vous pouvez ajouter **l'éditeur DGML** pour afficher des graphes dans le visualiseur :

1. Cochez la case pour **Éditeur DGML**. Il se trouve sous la section **Outils de code**.

Présentation des arborescences de syntaxe

Vous utilisez l'API Syntaxe pour les analyses de la structure du code C#. **L'API Syntaxe** expose les analyseurs, les arborescences de syntaxe, et des utilitaires pour l'analyse et la construction d'arborescences de syntaxe. C'est ainsi que vous recherchez le code pour des éléments de syntaxe spécifiques ou que vous lisez le code d'un programme.

Une arborescence de syntaxe est une structure de données utilisée par les compilateurs C# et Visual Basic pour comprendre les programmes C# et Visual Basic. Les arborescences de syntaxe sont produites par le même analyseur que celui qui s'exécute quand un projet est généré ou qu'un développeur appuie sur F5. Les arborescences de syntaxe sont parfaitement fidèles au langage : chaque élément d'information d'un fichier de code est représenté dans l'arborescence. L'écriture d'une arborescence de syntaxe en texte reproduit le texte d'origine exact qui a été analysé. Les arborescences de syntaxe sont également **immuables** : une fois créée, une arborescence de syntaxe ne peut jamais être modifiée. Les consommateurs des arborescences peuvent les analyser sur plusieurs threads, sans verrous ou d'autres mesures liées à la simultanéité des accès, sachant que les données ne changent jamais. Vous pouvez utiliser des API pour créer des arborescences qui sont le résultat de la modification d'une arborescence existante.

Les quatre principaux blocs de construction des arborescences de syntaxe sont :

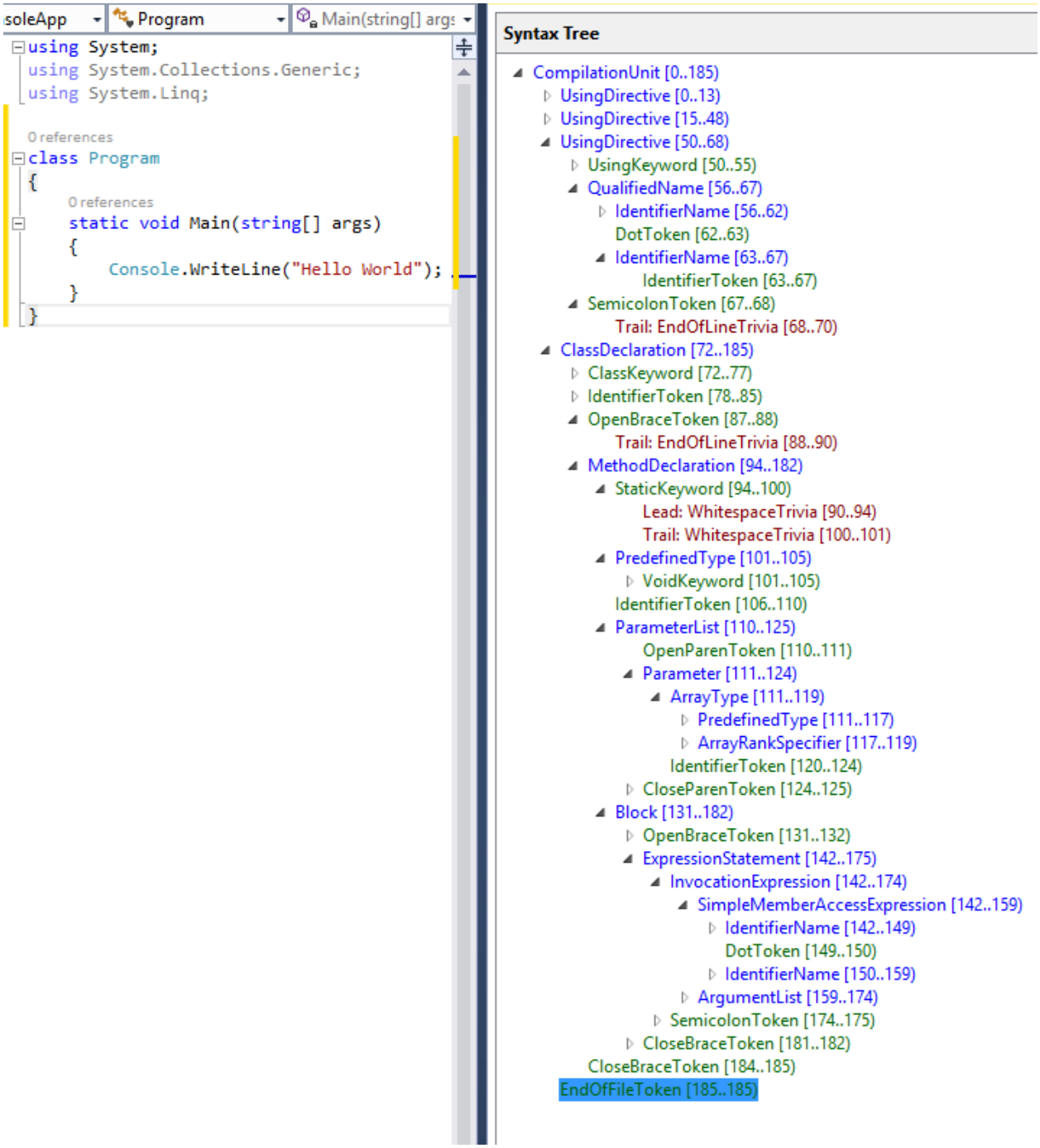
- La classe Microsoft.CodeAnalysis.SyntaxTree, une instance de celle-ci représentant une arborescence d'analyse entière. SyntaxTree est une classe abstraite avec des dérivés spécifiques à un langage. Vous utilisez les méthodes d'analyse de la classe

Microsoft.CodeAnalysis.CSharp.CSharpSyntaxTree (ou Microsoft.CodeAnalysis.VisualBasic.VisualBasicSyntaxTree) pour analyser du texte dans C# ou Visual Basic.

- La classe Microsoft.CodeAnalysis.SyntaxNode, dont les instances représentent des constructions syntaxiques, comme des déclarations, des instructions, des clauses et des expressions.
- La structure Microsoft.CodeAnalysis.SyntaxToken, qui représente un mot clé, un identificateur, un opérateur ou un signe de ponctuation individuel.
- Enfin, la structure Microsoft.CodeAnalysis.SyntaxTrivia, qui représente les éléments d'information non significatifs du point de vue syntaxique, comme l'espace entre des jetons, des directives de prétraitement et des commentaires.

Tout cela, les jetons et les nœuds sont composés de façon hiérarchique pour former une arborescence représentant complètement tous les éléments d'un fragment de code Visual Basic ou C#. Vous pouvez voir cette structure en utilisant la fenêtre **Visualiseur de syntaxe**. Dans Visual Studio, choisissez **Affichage**>**Autres fenêtres**>**Visualiseur de syntaxe**. Par exemple, le fichier source C# précédent examiné avec le **Visualiseur de syntaxe** se présente comme dans la figure suivante :

SyntaxNode : Bleu | **SyntaxToken** : Vert | **SyntaxTrivia** : Rouge



En parcourant cette arborescence, vous trouverez les instructions, expressions, jetons ou espaces, quels qu'ils soient, dans un fichier de code.

Vous pouvez rechercher n'importe quel élément dans un fichier de code à l'aide des API Syntaxe, mais la plupart des scénarios concernent l'examen de petits extraits de code, ou la recherche d'instructions ou de fragments particuliers. Les deux exemples qui suivent montrent des utilisations classiques d'exploration de la structure du code ou de recherche d'instructions individuelles.

Exploration des arborescences

Vous pouvez examiner les nœuds d'une arborescence de syntaxe de deux façons. Vous pouvez parcourir l'arborescence pour examiner chaque nœud, ou vous pouvez rechercher des éléments ou des nœuds spécifiques.

Exploration manuelle

Vous pouvez trouver le code complet de cet exemple dans notre dépôt GitHub.

> **Notes**
>
> Les types de l'arborescence de syntaxe utilisent l'héritage pour décrire les différents éléments de syntaxe qui sont valides à différents emplacements du programme. Utiliser ces API signifie souvent effectuer un cast des propriétés ou des membres de collection vers des types dérivés spécifiques. Dans les exemples suivants, l'affectation et les casts sont des instructions distinctes, qui utilisent des variables typées explicitement. Vous pouvez lire le code pour voir les types de retour de l'API et le type à l'exécution des objets retournés. Dans la pratique, il est plus courant d'utiliser des variables typées implicitement et de se baser sur des noms d'API pour décrire le type des objets examinés.

Créez un projet C# **Outil d'analyse du code autonome** :

- Dans Visual Studio, choisissez **Fichier**>**Nouveau**>**Projet** pour afficher la boîte de dialogue Nouveau projet.
- Sous **Visual C#**>**Extensibilité**, choisissez **Outil d'analyse du code autonome**.
- Nommez votre projet « **SyntaxTreeManualTraversal** » et cliquez sur OK.

Vous allez analyser le programme « Hello World! » de base indiqué plus haut. Ajoutez le texte pour le programme Hello World en tant que constante dans votre classe `Program` :

C#

```
        const string programText =
@"using System;
using System.Collections;
using System.Linq;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(""Hello, World!"");
```

```
        }
    }
}";
```

Ensuite, ajoutez le code suivant pour générer **l'arborescence de syntaxe** pour le texte du code dans la constante `programText`. Ajoutez la ligne suivante à votre méthode `Main` :

C#

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);
CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

Ces deux lignes créent l'arborescence et en récupèrent le nœud racine. Vous pouvez maintenant examiner les nœuds de l'arborescence. Ajoutez ces lignes à votre méthode `Main` pour afficher certaines des propriétés du nœud racine de l'arborescence :

C#

```
WriteLine($"The tree is a {root.Kind()} node.");
WriteLine($"The tree has {root.Members.Count} elements in it.");
WriteLine($"The tree has {root.Usings.Count} using statements. They are:");
foreach (UsingDirectiveSyntax element in root.Usings)
    WriteLine($"\t{element.Name}");
```

Exécutez l'application pour voir ce que votre code a découvert sur le nœud racine de cette arborescence.

En règle générale, vous explorez l'arborescence pour en savoir plus sur le code. Dans cet exemple, vous analysez du code que vous connaissez pour explorer les API. Ajoutez le code suivant pour examiner le premier membre du nœud `root` :

C#

```
MemberDeclarationSyntax firstMember = root.Members[0];
WriteLine($"The first member is a {firstMember.Kind()}.");
var helloWorldDeclaration = (NamespaceDeclarationSyntax)firstMember;
```

Ce membre est une Microsoft.CodeAnalysis.CSharp.Syntax.NamespaceDeclarationSyntax. Il représente tous les éléments de l'étendue de la déclaration `namespace HelloWorld`. Ajoutez le code suivant pour examiner les nœuds qui sont déclarés à l'intérieur de l'espace de noms `HelloWorld` :

C#

```
WriteLine($"There are {helloWorldDeclaration.Members.Count} members declared
in this namespace.");
WriteLine($"The first member is a
{helloWorldDeclaration.Members[0].Kind()}.");
```

Exécutez le programme pour voir ce que vous avez découvert.

Maintenant que vous savez que la déclaration est une Microsoft.CodeAnalysis.CSharp.Syntax.ClassDeclarationSyntax, déclarez une nouvelle variable de ce type pour examiner la déclaration de la classe. Cette classe contient seulement un membre : la méthode `Main`. Ajoutez le code suivant pour rechercher la méthode `Main` et en effectuer un cast en Microsoft.CodeAnalysis.CSharp.Syntax.MethodDeclarationSyntax.

C#

```
var programDeclaration =
(ClassDeclarationSyntax)helloWorldDeclaration.Members[0];
WriteLine($"There are {programDeclaration.Members.Count} members declared in
the {programDeclaration.Identifier} class.");
WriteLine($"The first member is a {programDeclaration.Members[0].Kind()}.");
var mainDeclaration =
(MethodDeclarationSyntax)programDeclaration.Members[0];
```

Le nœud de la déclaration de la méthode contient toutes les informations syntaxiques relatives à la méthode. Affichons le type de retour de la méthode `Main`, le nombre et les types des arguments, et le texte du corps de la méthode. Ajoutez le code suivant :

C#

```
WriteLine($"The return type of the {mainDeclaration.Identifier} method is
{mainDeclaration.ReturnType}.");
WriteLine($"The method has {mainDeclaration.ParameterList.Parameters.Count}
parameters.");
foreach (ParameterSyntax item in mainDeclaration.ParameterList.Parameters)
    WriteLine($"The type of the {item.Identifier} parameter is
{item.Type}.");
WriteLine($"The body text of the {mainDeclaration.Identifier} method
follows:");
WriteLine(mainDeclaration.Body?.ToFullString());

var argsParameter = mainDeclaration.ParameterList.Parameters[0];
```

Exécutez le programme pour voir toutes les informations que vous avez découvertes sur ce programme :

text

```
The tree is a CompilationUnit node.
The tree has 1 elements in it.
The tree has 4 using statements. They are:
        System
        System.Collections
        System.Linq
        System.Text
The first member is a NamespaceDeclaration.
There are 1 members declared in this namespace.
The first member is a ClassDeclaration.
There are 1 members declared in the Program class.
The first member is a MethodDeclaration.
The return type of the Main method is void.
The method has 1 parameters.
The type of the args parameter is string[].
The body text of the Main method follows:
        {
            Console.WriteLine("Hello, World!");
        }
```

Méthodes de requête

En plus de parcourir les arborescences, vous pouvez aussi explorer l'arborescence de syntaxe avec les méthodes de requête définies sur Microsoft.CodeAnalysis.SyntaxNode. Ces méthodes sont normalement immédiatement familières à toute personne connaissant XPath. Vous pouvez utiliser ces méthodes avec LINQ pour rechercher rapidement des éléments dans une arborescence. SyntaxNode a des méthodes de requête comme DescendantNodes, AncestorsAndSelf et ChildNodes.

Vous pouvez utiliser ces méthodes de requête pour rechercher l'argument de la méthode `Main`, au lieu de naviguer dans l'arborescence. Ajoutez le code suivant dans le bas de votre nouvelle méthode `Main` :

C#

```
var firstParameters = from methodDeclaration in root.DescendantNodes()
                                        .OfType<MethodDeclarationSyntax>()
                      where methodDeclaration.Identifier.ValueText == "Main"
                      select
methodDeclaration.ParameterList.Parameters.First();

var argsParameter2 = firstParameters.Single();

WriteLine(argsParameter == argsParameter2);
```

La première instruction utilise une expression LINQ et la méthode DescendantNodes pour localiser le même paramètre que dans l'exemple précédent.

Exécutez le programme : vous voyez que l'expression LINQ a trouvé le même paramètre que quand vous avez exploré manuellement l'arborescence.

L'exemple utilise des instructions `WriteLine` pour afficher des informations sur les arborescences de syntaxe au fil de leur exploration. Vous pouvez aussi découvrir beaucoup plus d'informations en exécutant le programme sous le débogueur. Vous pouvez examiner plus de propriétés et de méthodes qui font partie de l'arborescence de syntaxe créée pour le programme « Hello World ».

Parcoureurs de syntaxe

Vous voulez souvent rechercher tous les nœuds d'un type spécifique dans une arborescence de syntaxe, par exemple toutes les déclarations de propriété dans un fichier. En étendant la classe Microsoft.CodeAnalysis.CSharp.CSharpSyntaxWalker et en remplaçant la méthode VisitPropertyDeclaration(PropertyDeclarationSyntax), vous traitez chaque déclaration de propriété d'une arborescence de syntaxe sans connaître préalablement sa structure. CSharpSyntaxWalker est un type spécifique de CSharpSyntaxVisitor, qui visite de façon récursive un nœud et chacun de ses enfants.

Cet exemple implémente un CSharpSyntaxWalker qui examine une arborescence de syntaxe. Il collecte les directives `using` qu'il trouve et qui n'importent pas un espace de noms `System`.

Créez un projet C# **Outil d'analyse du code autonome** et nommez-le « **SyntaxWalker** ».

Vous pouvez trouver le code complet de cet exemple dans notre dépôt GitHub. L'exemple sur GitHub contient les deux projets décrits dans ce tutoriel.

Comme dans l'exemple précédent, vous pouvez définir une constante de chaîne pour contenir le texte du programme que vous voulez analyser :

C#

```
        const string programText =
@"using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;

namespace TopLevel
{
    using Microsoft;
    using System.ComponentModel;
```

```
    namespace Child1
    {
        using Microsoft.Win32;
        using System.Runtime.InteropServices;

        class Foo { }
    }

    namespace Child2
    {
        using System.CodeDom;
        using Microsoft.CSharp;

        class Bar { }
    }
}";
```

Ce texte source contient des directives `using` réparties dans quatre emplacements différents : le niveau fichier, dans l'espace de noms du plus haut niveau et dans les deux espaces de noms imbriqués. Cet exemple met en évidence un scénario de base pour l'utilisation de la classe CSharpSyntaxWalker pour interroger du code. Il serait fastidieux de visiter chaque nœud dans l'arborescence de syntaxe racine pour trouver les déclarations using. Au lieu de cela, vous créez une classe dérivée et vous remplacez la méthode qui est appelée seulement quand le nœud actuel de l'arborescence est une directive using. Votre visiteur n'effectue aucun travail sur les autres types de nœuds. Cette même méthode examine chacune des instructions `using` et génère une collection des espaces de noms qui ne figurent pas dans l'espace de noms `System`. Vous créez un CSharpSyntaxWalker qui examine toutes les instructions `using`, mais uniquement les instructions `using`.

Maintenant que vous avez défini le texte du programme, vous devez créer un `SyntaxTree` et obtenir la racine de cette arborescence :

C#

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);
CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

Ensuite, créez une classe. Dans Visual Studio, choisissez **Projet**>**Ajouter un nouvel élément**. Dans la boîte de dialogue **Ajouter un nouvel élément**, tapez *UsingCollector.cs* comme nom de fichier.

Vous implémentez la fonctionnalité de visiteur `using` dans la classe `UsingCollector`. Commencez par faire de la classe `UsingCollector` une dérivée CSharpSyntaxWalker.

```
```

C#

```
class UsingCollector : CSharpSyntaxWalker
```

Vous avez besoin d'un stockage pour contenir les nœuds d'espace de noms que vous collectez. Déclarez une propriété publique en lecture seule dans la classe `UsingCollector` ; vous utilisez cette variable pour stocker les nœuds UsingDirectiveSyntax que vous trouvez :

C#

```
public ICollection<UsingDirectiveSyntax> Usings { get; } = new
List<UsingDirectiveSyntax>();
```

La classe de base, CSharpSyntaxWalker, implémente la logique pour visiter chaque nœud de l'arborescence de syntaxe. La classe dérivée remplace les méthodes appelées pour les nœuds spécifiques qui vous intéressent. Dans ce cas, vous êtes intéressé par toutes les directives `using`. Cela signifie que vous devez remplacer la méthode VisitUsingDirective(UsingDirectiveSyntax). Le seul argument de cette méthode est un objet Microsoft.CodeAnalysis.CSharp.Syntax.UsingDirectiveSyntax. Utiliser des visiteurs a un avantage important : ils appellent les méthodes remplacées avec des arguments déjà castés au type de nœud spécifique. La classe Microsoft.CodeAnalysis.CSharp.Syntax.UsingDirectiveSyntax a une propriété Name qui stocke le nom de l'espace de noms importé. Il s'agit d'une Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax. Ajoutez le code suivant dans le remplacement de VisitUsingDirective(UsingDirectiveSyntax) :

C#

```
public override void VisitUsingDirective(UsingDirectiveSyntax node)
{
    WriteLine($"\tVisitUsingDirective called with {node.Name}.");
    if (node.Name.ToString() != "System" &&
        !node.Name.ToString().StartsWith("System."))
    {
        WriteLine($"\t\tSuccess. Adding {node.Name}.");
        this.Usings.Add(node);
    }
}
```

Comme avec l'exemple précédent, vous avez ajouté différentes instructions `WriteLine` pour faciliter la compréhension de cette méthode. Vous pouvez voir quand elle est appelée, ainsi que les arguments qui lui sont passés à chaque fois.

Enfin, vous devez ajouter deux lignes de code pour créer le `UsingCollector` et faire en sorte qu'il visite le nœud racine, en collectant toutes les instructions `using`. Ensuite, ajoutez une boucle `foreach` pour afficher toutes les instructions `using` trouvées par votre collecteur :

C#

```
var collector = new UsingCollector();
collector.Visit(root);
foreach (var directive in collector.Usings)
{
    WriteLine(directive.Name);
}
```

Compilez et exécutez le programme. Vous devez normalement voir la sortie suivante :

Console

```
        VisitUsingDirective called with System.
        VisitUsingDirective called with System.Collections.Generic.
        VisitUsingDirective called with System.Linq.
        VisitUsingDirective called with System.Text.
        VisitUsingDirective called with Microsoft.CodeAnalysis.
                Success. Adding Microsoft.CodeAnalysis.
        VisitUsingDirective called with Microsoft.CodeAnalysis.CSharp.
                Success. Adding Microsoft.CodeAnalysis.CSharp.
        VisitUsingDirective called with Microsoft.
                Success. Adding Microsoft.
        VisitUsingDirective called with System.ComponentModel.
        VisitUsingDirective called with Microsoft.Win32.
                Success. Adding Microsoft.Win32.
        VisitUsingDirective called with System.Runtime.InteropServices.
        VisitUsingDirective called with System.CodeDom.
        VisitUsingDirective called with Microsoft.CSharp.
                Success. Adding Microsoft.CSharp.
Microsoft.CodeAnalysis
Microsoft.CodeAnalysis.CSharp
Microsoft
Microsoft.Win32
Microsoft.CSharp
Press any key to continue . . .
```

Félicitations ! Vous avez utilisé **l'API Syntaxe** pour rechercher des types spécifiques d'instructions et de déclarations C# dans du code source C#.

Bien démarrer avec l'analyse sémantique

Article • 10/05/2023

Ce tutoriel suppose que vous êtes familiarisé avec l'API Syntaxe. L'article Bien démarrer avec l'analyse syntaxique fournit une introduction suffisante.

Dans ce tutoriel, vous découvrez les API **Symbole** et **Liaison**. Ces API fournissent des informations sur la *signification sémantique* d'un programme. Elles vous permettent de poser et de répondre à des questions sur les types représentés par un symbole dans votre programme.

Vous devez installer le SDK **.NET Compiler Platform** :

Instructions d'installation - Visual Studio Installer

Vous pouvez rechercher **SDK .NET Compiler Platform** dans **Visual Studio Installer** de deux façons :

Vue Installation avec Visual Studio Installer - Charges de travail

SDK .NET Compiler Platform n'est pas sélectionné automatiquement dans le cadre de la charge de travail Développement d'extensions Visual Studio. Vous devez le sélectionner comme composant facultatif.

1. Exécutez **Visual Studio Installer**.
2. Sélectionnez **Modifier**
3. Choisissez la charge de travail **Développement d'extensions Visual Studio**.
4. Ouvrez le nœud **Développement d'extensions Visual Studio** dans l'arborescence résumée.
5. Cochez la case pour **SDK .NET Compiler Platform**. Il se trouve en dernier sous les composants facultatifs.

Si vous le souhaitez, vous pouvez ajouter **l'éditeur DGML** pour afficher des graphes dans le visualiseur :

1. Ouvrez le nœud **Composants individuels** dans l'arborescence résumée.
2. Cochez la case pour **Éditeur DGML**.

Onglet Installation avec Visual Studio Installer - Composants individuels

1. Exécutez **Visual Studio Installer**.
2. Sélectionnez **Modifier**
3. Sélectionnez l'onglet **Composants individuels**.
4. Cochez la case pour **SDK .NET Compiler Platform**. Il se trouve en haut de la liste, sous la section **Compilateurs, outils de génération et runtimes**.

Si vous le souhaitez, vous pouvez ajouter **l'éditeur DGML** pour afficher des graphes dans le visualiseur :

1. Cochez la case pour **Éditeur DGML**. Il se trouve sous la section **Outils de code**.

Présentation des compilations et des symboles

Au fil de votre utilisation du SDK du compilateur .NET, vous allez mieux comprendre les différences entre l'API Syntaxe et l'API Sémantique. **L'API Syntaxe** vous permet d'examiner la *structure* d'un programme. Cependant, des informations plus détaillées sur la sémantique ou la *signification* d'un programme sont souvent nécessaires. Si la syntaxe d'un fichier ou d'un extrait de code autonome Visual Basic ou C# peut être analysé de façon isolée, cela ne veut pas dire grand-chose de poser des questions comme « quel est le type de cette variable ?» dans le vide. La signification d'un nom de type peut dépendre de références d'assembly, d'importations d'espaces de noms ou d'autres fichiers de code. Ces questions sont traitées avec **l'API Sémantique**, plus précisément avec la classe Microsoft.CodeAnalysis.Compilation.

Une instance de Compilation est analogue à un projet tel qu'il est vu par le compilateur et représente tout ce qui est nécessaire pour compiler un programme Visual Basic ou C#. La **compilation** comprend l'ensemble des fichiers sources à compiler, les références d'assembly et les options du compilateur. Vous pouvez analyser la signification du code en utilisant toutes les autres informations de ce contexte. Une Compilation vous permet de rechercher des **symboles**, qui sont des entités comme des types, des espaces de noms, des membres et des variables, qui sont référencées par des noms et d'autres expressions. Le processus consistant à associer des noms et des expressions à des **symboles** est appelé la **liaison**.

Comme Microsoft.CodeAnalysis.SyntaxTree, Compilation est une classe abstraite avec des dérivés spécifiques à un langage. Quand vous créez une instance de compilation, vous devez appeler une méthode de fabrique sur la classe Microsoft.CodeAnalysis.CSharp.CSharpCompilation (ou Microsoft.CodeAnalysis.VisualBasic.VisualBasicCompilation).

Interrogation des symboles

Dans ce tutoriel, vous examinez à nouveau le programme « Hello World ». Cette fois-ci, vous interrogez les symboles du programme pour comprendre quels types ces symboles représentent. Vous interrogez les types dans un espace de noms et vous découvrez comment trouver les méthodes disponibles sur un type.

Vous pouvez trouver le code complet de cet exemple dans notre dépôt GitHub.

> **Notes**
>
> Les types de l'arborescence de syntaxe utilisent l'héritage pour décrire les différents éléments de syntaxe qui sont valides à différents emplacements du programme. Utiliser ces API signifie souvent effectuer un cast des propriétés ou des membres de collection vers des types dérivés spécifiques. Dans les exemples suivants, l'affectation et les casts sont des instructions distinctes, qui utilisent des variables typées explicitement. Vous pouvez lire le code pour voir les types de retour de l'API et le type à l'exécution des objets retournés. Dans la pratique, il est plus courant d'utiliser des variables typées implicitement et de se baser sur des noms d'API pour décrire le type des objets examinés.

Créez un projet C# **Outil d'analyse du code autonome** :

- Dans Visual Studio, choisissez **Fichier**>**Nouveau**>**Projet** pour afficher la boîte de dialogue Nouveau projet.
- Sous **Visual C#**>**Extensibilité**, choisissez **Outil d'analyse du code autonome**.
- Nommez votre projet « **SemanticQuickStart** » et cliquez sur OK.

Vous allez analyser le programme « Hello World ! » de base indiqué plus haut. Ajoutez le texte pour le programme Hello World en tant que constante dans votre classe `Program` :

C#

```
        const string programText =
@"using System;
using System.Collections.Generic;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(""Hello, World!"");
```

```
        }
    }
}";
```

Ensuite, ajoutez le code suivant pour générer l'arborescence de syntaxe pour le texte du code dans la constante `programText`. Ajoutez la ligne suivante à votre méthode `Main` :

C#

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);

CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

Ensuite, générez une CSharpCompilation à partir de l'arborescence que vous avez déjà créée. L'exemple « Hello World » est basé sur les types String et Console. Vous devez référencer l'assembly qui déclare ces deux types dans votre compilation. Ajoutez la ligne suivante à votre méthode `Main` pour créer une compilation de votre arborescence de syntaxe, en incluant la référence à l'assembly approprié :

C#

```
var compilation = CSharpCompilation.Create("HelloWorld")
    .AddReferences(MetadataReference.CreateFromFile(
        typeof(string).Assembly.Location))
    .AddSyntaxTrees(tree);
```

La méthode CSharpCompilation.AddReferences ajoute des références à la compilation. La méthode MetadataReference.CreateFromFile charge un assembly en tant que référence.

Interrogation du modèle sémantique

Une fois que vous avez une Compilation, vous pouvez la demander pour un SemanticModel pour n'importe quelle SyntaxTree contenue dans Compilation. Vous pouvez considérer le modèle sémantique comme étant la source de toutes les informations que vous obtiendriez normalement d'IntelliSense. Un SemanticModel peut répondre à des questions comme « Quels noms se trouvent dans l'étendue à cet emplacement ? », « Quels membres sont accessibles à partir de cette méthode ? », « Quelles variables sont utilisées dans ce bloc de texte ? » et « À quoi ce nom ou cette expression font-ils référence ? » Ajoutez cette déclaration pour créer un modèle sémantique :

C#

```
SemanticModel model = compilation.GetSemanticModel(tree);
```

Liaison d'un nom

Le Compilation crée SemanticModel à partir de SyntaxTree. Après avoir créé le modèle, vous pouvez l'interroger pour rechercher la première directive `using` et récupérer les informations des symboles pour l'espace de noms `System`. Ajoutez ces deux lignes à votre méthode `Main` pour créer le modèle sémantique et pour récupérer le symbole pour la première instruction using :

C#

```
// Use the syntax tree to find "using System;"
UsingDirectiveSyntax usingSystem = root.Usings[0];
NameSyntax systemName = usingSystem.Name;

// Use the semantic model for symbol information:
SymbolInfo nameInfo = model.GetSymbolInfo(systemName);
```

Le code précédent montre comment lier le nom dans la première directive `using` pour récupérer un Microsoft.CodeAnalysis.SymbolInfo pour l'espace de noms `System`. Le code précédent montre aussi que vous utilisez le **modèle de syntaxe** pour trouver la structure du code ; vous utilisez le **modèle sémantique** pour comprendre sa signification. Le **modèle de syntaxe** recherche la chaîne `System` dans l'instruction using. Le **modèle sémantique** a toutes les informations sur les types définis dans l'espace de noms `System`.

À partir de l'objet SymbolInfo, vous pouvez obtenir Microsoft.CodeAnalysis.ISymbol en utilisant la propriété SymbolInfo.Symbol. Cette propriété retourne le symbole auquel cette expression fait référence. Pour les expressions qui ne font référence à rien (comme les littéraux numériques), cette propriété est `null`. Quand SymbolInfo.Symbol n'est pas null, ISymbol.Kind indique le type du symbole. Dans cet exemple, la propriété ISymbol.Kind est un SymbolKind.Namespace. Ajoutez le code suivant à votre méthode `Main`. Il récupère le symbole pour l'espace de noms `System`, puis affiche tous les espaces de noms enfants déclarés dans l'espace de noms `System` :

C#

```
var systemSymbol = (INamespaceSymbol?)nameInfo.Symbol;
if (systemSymbol?.GetNamespaceMembers() is not null)
{
    foreach (INamespaceSymbol ns in systemSymbol?.GetNamespaceMembers()!)
```

```
    {
        Console.WriteLine(ns);
    }
}
```

Exécutez le programme ; vous devez normalement voir la sortie suivante :

Sortie

```
System.Collections
System.Configuration
System.Deployment
System.Diagnostics
System.Globalization
System.IO
System.Numerics
System.Reflection
System.Resources
System.Runtime
System.Security
System.StubHelpers
System.Text
System.Threading
Press any key to continue . . .
```

Notes

La sortie n'inclut pas chaque espace de noms qui est un espace de noms enfant de l'espace de noms `System`. Elle affiche chaque espace de noms qui est présent dans cette compilation, qui référence seulement l'assembly où `System.String` est déclaré. Les espaces de noms déclarés dans d'autres assemblys ne sont pas connus de cette compilation.

Liaison d'une expression

Le code précédent montre comment trouver un symbole en le liant à un nom. D'autres expressions existent dans un programme C# qui peuvent être liées, mais qui ne sont pas des noms. Pour montrer cette possibilité, accédons à la liaison vers un littéral de chaîne simple.

Le programme « Hello World » contient un Microsoft.CodeAnalysis.CSharp.Syntax.LiteralExpressionSyntax, la chaîne « Hello, World! » affichée dans la console.

Vous trouvez la chaîne « Hello World » en recherchant le seul littéral de chaîne dans le programme. Ensuite, une fois que vous avez localisé le nœud de syntaxe, vous obtenez les informations de type pour ce nœud auprès du modèle sémantique. Ajoutez le code suivant à votre méthode `Main` :

```
C#

// Use the syntax model to find the literal string:
LiteralExpressionSyntax helloWorldString = root.DescendantNodes()
.OfType<LiteralExpressionSyntax>()
.Single();

// Use the semantic model for type information:
TypeInfo literalInfo = model.GetTypeInfo(helloWorldString);
```

La struct Microsoft.CodeAnalysis.TypeInfo inclut une propriété TypeInfo.Type qui permet d'accéder aux informations sémantiques sur le type du littéral. Dans cet exemple, il s'agit du type `string`. Ajoutez une déclaration qui affecte cette propriété à une variable locale :

```
C#

var stringTypeSymbol = (INamedTypeSymbol?)literalInfo.Type;
```

Pour terminer ce tutoriel, construisons une requête LINQ qui crée une séquence de toutes les méthodes publiques déclarées sur le type `string` qui retournent un type `string`. Cette requête est complexe : nous allons donc la construire ligne par ligne, puis la reconstruire pour former une seule requête. La source de cette requête est la séquence de tous les membres déclarés sur le type `string` :

```
C#

var allMembers = stringTypeSymbol?.GetMembers();
```

Cette séquence source contient tous les membres, notamment les propriétés et les champs ; filtrez-la avec la méthode ImmutableArray<T>.OfType pour trouver les éléments qui sont des objets Microsoft.CodeAnalysis.IMethodSymbol :

```
C#

var methods = allMembers?.OfType<IMethodSymbol>();
```

Ensuite, ajoutez un autre filtre pour retourner seulement les méthodes qui sont publiques et qui retournent un type `string` :

C#

```
var publicStringReturningMethods = methods?
    .Where(m => SymbolEqualityComparer.Default.Equals(m.ReturnType,
stringTypeSymbol) &&
    m.DeclaredAccessibility == Accessibility.Public);
```

Sélectionnez seulement la propriété name et seulement les noms distincts en supprimant les surcharges :

C#

```
var distinctMethods = publicStringReturningMethods?.Select(m =>
m.Name).Distinct();
```

Vous pouvez aussi créer toute la requête avec la syntaxe de requête LINQ, puis afficher tous les noms de méthode dans la console :

C#

```
foreach (string name in (from method in stringTypeSymbol?
                         .GetMembers().OfType<IMethodSymbol>()
                         where
SymbolEqualityComparer.Default.Equals(method.ReturnType, stringTypeSymbol)
&&
                         method.DeclaredAccessibility ==
Accessibility.Public
                         select method.Name).Distinct())
{
    Console.WriteLine(name);
}
```

Générez et exécutez le programme. Vous devez normalement voir la sortie suivante :

Sortie

```
Join
Substring
Trim
TrimStart
TrimEnd
Normalize
PadLeft
PadRight
ToLower
```

```
ToLowerInvariant
ToUpper
ToUpperInvariant
ToString
Insert
Replace
Remove
Format
Copy
Concat
Intern
IsInterned
Press any key to continue . . .
```

Vous avez utilisé l'API Sémantique pour rechercher et afficher des informations sur les symboles qui font partie de ce programme.

Bien démarrer avec la transformation de la syntaxe

Article • 09/05/2023

Ce tutoriel s'appuie sur les concepts et techniques décrits dans les démarrages rapides Bien démarrer avec l'analyse de la syntaxe et Bien démarrer avec l'analyse sémantique. Si vous ne l'avez pas déjà fait, vous devez terminer ces démarrages rapides avant de commencer celui-ci.

Dans ce démarrage rapide, vous découvrez les techniques de création et de transformation des arborescences de syntaxe. En association avec les techniques que vous avez apprises dans les démarrages rapides précédents, vous créez votre première refactorisation de ligne de commande !

Instructions d'installation - Visual Studio Installer

Vous pouvez rechercher **SDK .NET Compiler Platform** dans **Visual Studio Installer** de deux façons :

Vue Installation avec Visual Studio Installer - Charges de travail

SDK .NET Compiler Platform n'est pas sélectionné automatiquement dans le cadre de la charge de travail Développement d'extensions Visual Studio. Vous devez le sélectionner comme composant facultatif.

1. Exécutez **Visual Studio Installer**.
2. Sélectionnez **Modifier**
3. Choisissez la charge de travail **Développement d'extensions Visual Studio**.
4. Ouvrez le nœud **Développement d'extensions Visual Studio** dans l'arborescence résumée.
5. Cochez la case pour **SDK .NET Compiler Platform**. Il se trouve en dernier sous les composants facultatifs.

Si vous le souhaitez, vous pouvez ajouter **l'éditeur DGML** pour afficher des graphes dans le visualiseur :

1. Ouvrez le nœud **Composants individuels** dans l'arborescence résumée.

2. Cochez la case pour **Éditeur DGML**.

Onglet Installation avec Visual Studio Installer - Composants individuels

1. Exécutez **Visual Studio Installer**.
2. Sélectionnez **Modifier**
3. Sélectionnez l'onglet **Composants individuels**.
4. Cochez la case pour **SDK .NET Compiler Platform**. Il se trouve en haut de la liste, sous la section **Compilateurs, outils de génération et runtimes**.

Si vous le souhaitez, vous pouvez ajouter **l'éditeur DGML** pour afficher des graphes dans le visualiseur :

1. Cochez la case pour **Éditeur DGML**. Il se trouve sous la section **Outils de code**.

Immuabilité et plateforme de compilateur .NET

L'**immuabilité** est un principe fondamental de la plateforme de compilateur .NET. Une fois créées, les structures de données immuables ne peuvent plus être modifiées. Les structures de données immuables peuvent être partagées et analysées simultanément par plusieurs consommateurs, en toute sécurité. Il n'existe aucun risque qu'un consommateur n'en perturbe un autre de manière imprévisible. Votre analyseur n'a pas besoin de verrous ou d'autres mesures de concurrence. Cette règle s'applique aux arborescences de syntaxe, compilations, symboles, modèles sémantiques et toute autre structure de données que vous rencontrez. Au lieu de modifier des structures existantes, les API créent de nouveaux objets en fonction des différences spécifiés dans les anciens objets. Vous appliquez ce concept aux arborescences de syntaxe pour créer de nouvelles arborescences à l'aide de transformations.

Créer et transformer des arborescences

Vous choisissez une des deux stratégies pour les transformations de syntaxe. Les **méthodes de fabrique** sont particulièrement utiles lorsque vous recherchez des nœuds spécifiques à remplacer, ou des emplacements spécifiques dans lesquelles vous souhaitez insérer le nouveau code. Les **modules de réécriture** sont recommandés lorsque vous souhaitez analyser un projet complet et y rechercher des modèles de code à remplacer.

Créer des nœuds avec des méthodes de fabrique

La première transformation de syntaxe montre les méthodes de fabrique. Vous allez remplacer une instruction `using System.Collections;` par une instruction `using System.Collections.Generic;`. Cet exemple montre comment créer des objets Microsoft.CodeAnalysis.CSharp.CSharpSyntaxNode à l'aide des méthodes de fabrique Microsoft.CodeAnalysis.CSharp.SyntaxFactory. Pour chaque type de **nœud**, **jeton** ou **trivia**, il existe une méthode de fabrique qui crée une instance de ce type. Vous créez des arborescences de syntaxe en composant hiérarchiquement des nœuds de façon ascendante. Puis vous transformerez le programme existant en remplaçant les nœuds existants par la nouvelle arborescence que vous avez créée.

Démarrez Visual Studio, puis créez un nouveau projet C# **Outil d'analyse du code autonome**. Dans Visual Studio, choisissez **Fichier**>**Nouveau**>**Projet** pour afficher la boîte de dialogue Nouveau projet. Sous **Visual C#**>**Extensibilité**, choisissez **Outil d'analyse du code autonome**. Ce démarrage rapide inclut deux exemples de projets, par conséquent, nommez la solution **SyntaxTransformationQuickStart**, puis nommez le projet **ConstructionCS**. Cliquez sur **OK**.

Ce projet utilise les méthodes de classe Microsoft.CodeAnalysis.CSharp.SyntaxFactory pour construire un élément Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax représentant l'espace de noms `System.Collections.Generic`.

Ajoutez la directive using suivante en début de `Program.cs`.

```
C#

using static Microsoft.CodeAnalysis.CSharp.SyntaxFactory;
using static System.Console;
```

Vous allez créer des **nœuds de syntaxe du nom** pour générer l'arborescence qui représente l'instruction `using System.Collections.Generic;`. NameSyntax est la classe de base pour les quatre types de noms qui s'affichent en C#. Vous composez ces quatre types de noms à la fois pour créer n'importe quel nom pouvant s'afficher en langage C# :

- Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax, qui représente des noms d'identificateur unique simples comme `System` et `Microsoft`.
- Microsoft.CodeAnalysis.CSharp.Syntax.GenericNameSyntax, qui représente un nom de type ou de méthode générique comme `List<int>`.
- Microsoft.CodeAnalysis.CSharp.Syntax.QualifiedNameSyntax, qui représente un nom qualifié de formulaire `<left-name>.<right-identifier-or-generic-name>` comme `System.IO`.

- Microsoft.CodeAnalysis.CSharp.Syntax.AliasQualifiedNameSyntax, qui représente un nom en utilisant un alias d'assembly externe comme `LibraryV2::Foo`.

Vous utilisez la méthode IdentifierName(String) pour créer un nœud NameSyntax. Ajoutez le code suivant à votre méthode `Main`, dans `Program.cs` :

```
C#

NameSyntax name = IdentifierName("System");
WriteLine($"\tCreated the identifier {name}");
```

Le code précédent crée un objet IdentifierNameSyntax et l'assigne à la variable `name`. La plupart des API Roslyn renvoient des classes de base pour les rendre plus faciles à utiliser avec les types associés. La variable `name`, une NameSyntax, peut être réutilisée lorsque vous générez la QualifiedNameSyntax. N'utilisez pas l'inférence de type lorsque vous générez l'exemple. Vous allez automatiser cette étape dans ce projet.

Vous avez créé le nom. Maintenant, il est temps de créer d'autres nœuds dans l'arborescence en générant une QualifiedNameSyntax. La nouvelle arborescence utilise `name` comme partie gauche du nom et une nouvelle IdentifierNameSyntax pour l'espace de noms `Collections` comme partie droite de la QualifiedNameSyntax. Ajoutez le code suivant à `program.cs` :

```
C#

name = QualifiedName(name, IdentifierName("Collections"));
WriteLine(name.ToString());
```

Exécutez à nouveau le code et affichez les résultats. Vous créez une arborescence de nœuds qui représente le code. Vous allez continuer à utiliser ce modèle pour générer la QualifiedNameSyntax pour l'espace de noms `System.Collections.Generic`. Ajoutez le code suivant à `Program.cs` :

```
C#

name = QualifiedName(name, IdentifierName("Generic"));
WriteLine(name.ToString());
```

Exécutez à nouveau le programme pour constater que vous avez généré l'arborescence du code à ajouter.

Créer une arborescence modifiée

Vous avez créé petite arborescence de syntaxe contenant une instruction. Les API permettant de créer de nouveaux nœuds représentent la solution idéale pour créer des instructions uniques ou d'autres petits blocs de code. Toutefois, pour générer de plus grands blocs de code, vous devez utiliser des méthodes qui remplacent les nœuds ou insèrent des nœuds dans une arborescence existante. N'oubliez pas que les arborescences de syntaxe sont immuables. L'**API Syntaxe** ne fournit aucun mécanisme permettant de modifier une arborescence de syntaxe existante après la construction. Au lieu de cela, elle fournit des méthodes qui génèrent de nouvelles arborescences en fonction des modifications apportées aux arborescences existantes. Les méthodes `With*` sont définies dans des classes concrètes qui dérivent de SyntaxNode ou dans des méthodes d'extension déclarées dans la classe SyntaxNodeExtensions. Ces méthodes créent un nouveau nœud en appliquant les modifications apportées aux propriétés enfants d'un nœud existant. En outre, la méthode d'extension ReplaceNode peut être utilisée pour remplacer un nœud descendant dans une sous-arborescence. Cette méthode met également à jour le parent pour pointer vers l'enfant qui vient d'être créé, et répète ce processus pour l'arborescence complète. Ce processus est appelé « *re-spining* » d'arborescence.

L'étape suivante consiste à créer une arborescence qui représente un (petit) programme complet puis à le modifier. Ajoutez le code suivant au début de la classe `Program` :

C#

```
        private const string sampleCode =
@"using System;
using System.Collections;
using System.Linq;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(""Hello, World!"");
        }
    }
}";
```

> **Notes**
>
> L'exemple de code utilise l'espace de noms `System.Collections` et non pas l'espace de noms `System.Collections.Generic`.

Ensuite, ajoutez le code suivant en bas de la méthode `Main` pour analyser le texte et créer une arborescence :

```
C#

SyntaxTree tree = CSharpSyntaxTree.ParseText(sampleCode);
var root = (CompilationUnitSyntax)tree.GetRoot();
```

Cet exemple utilise la méthode WithName(NameSyntax) pour remplacer le nom dans un nœud UsingDirectiveSyntax par celui construit dans le code précédent.

Créez un nouveau nœud UsingDirectiveSyntax à l'aide de la méthode WithName(NameSyntax) pour mettre à jour le nom `System.Collections` avec le nom que vous avez créé dans le code précédent. Ajoutez le code suivant dans le bas de la méthode `Main` :

```
C#

var oldUsing = root.Usings[1];
var newUsing = oldUsing.WithName(name);
WriteLine(root.ToString());
```

Exécutez le programme et examinez attentivement la sortie. `newUsing` n'a pas été placé dans l'arborescence racine. L'arborescence d'origine n'a pas été modifiée.

Ajoutez le code suivant à l'aide de la méthode d'extension ReplaceNode pour créer une nouvelle arborescence. La nouvelle arborescence est le résultat du remplacement de l'importation existante par le nœud `newUsing` mis à jour. Vous affectez cette nouvelle arborescence à la variable `root` existante :

```
C#

root = root.ReplaceNode(oldUsing, newUsing);
WriteLine(root.ToString());
```

Réexécutez le programme. Cette fois, l'arborescence importe correctement l'espace de noms `System.Collections.Generic`.

Transformer des arborescences avec `SyntaxRewriters`

Les méthodes `With*` et ReplaceNode constituent une solution pratique pour transformer des branches individuelles en une arborescence de syntaxe. La classe Microsoft.CodeAnalysis.CSharp.CSharpSyntaxRewriter effectue plusieurs transformations

sur une arborescence de syntaxe. La classe Microsoft.CodeAnalysis.CSharp.CSharpSyntaxRewriter est une sous-classe de la classe Microsoft.CodeAnalysis.CSharp.CSharpSyntaxVisitor<TResult>. CSharpSyntaxRewriter applique une transformation à un type spécifique de SyntaxNode. Vous pouvez appliquer des transformations à plusieurs types d'objets SyntaxNode là où elles apparaissent dans une arborescence de syntaxe. Le second projet de ce démarrage rapide crée une refactorisation de ligne de commande qui supprime des types explicites dans des déclarations de variables locales partout où une inférence de type peut être utilisée.

Créez un projet C# **Outil d'analyse du code autonome**. Dans Visual Studio, cliquez avec le bouton droit sur le nœud de la solution `SyntaxTransformationQuickStart`. Choisissez **Ajouter**>**Nouveau projet** pour afficher la boîte de dialogue **Nouveau projet**. Sous **Visual C#**>**Extensibilité**, choisissez **Outil d'analyse du code autonome**. Nommez votre projet `TransformationCS`, puis cliquez sur OK.

La première étape consiste à créer une classe dérivée de CSharpSyntaxRewriter pour effectuer vos transformations. Ajoutez un nouveau fichier de classe au projet. Dans Visual Studio, choisissez **Projet**>**Ajouter une classe...**. Dans la boîte de dialogue **Ajouter un nouvel élément**, tapez `TypeInferenceRewriter.cs` pour le nom de fichier.

Ajoutez le code suivant à l'aide de directives dans le fichier `TypeInferenceRewriter.cs` :

```
C#

using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;
```

Ensuite, étendez la classe `TypeInferenceRewriter` à la classe CSharpSyntaxRewriter :

```
C#

public class TypeInferenceRewriter : CSharpSyntaxRewriter
```

Ajoutez le code suivant pour déclarer un champ en lecture seule privé qui contiendra une SemanticModel et initialisez-le dans le constructeur. Vous aurez besoin de ce champ ultérieurement pour déterminer où l'inférence de type peut être utilisée :

```
C#

private readonly SemanticModel SemanticModel;
```

```
public TypeInferenceRewriter(SemanticModel semanticModel) => SemanticModel =
semanticModel;
```

Remplacez la méthode VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax) :

C#

```
public override SyntaxNode
VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax node)
{

}
```

Notes

De nombreuses API Roslyn déclarent des types de retour représentant des classes de base des types Runtime réels retournés. Dans de nombreux scénarios, il est possible de remplacer un type de nœud par un autre, voire de le supprimer. Dans cet exemple, la méthode **VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax)** retourne un **SyntaxNode**, au lieu du type dérivé de **LocalDeclarationStatementSyntax**. Ce module de réécriture retourne un nouveau nœud **LocalDeclarationStatementSyntax** en fonction du nœud existant.

Ce démarrage rapide gère les déclarations de variables locales. Vous pouvez l'étendre à d'autres déclarations telles que des boucles `foreach`, des boucles `for`, des expressions LINQ et des expressions lambda. En outre, ce module de réécriture transformera les déclarations dans la forme la plus simple :

C#

```
Type variable = expression;
```

Si vous souhaitez explorer votre propre déclaration, vous pouvez étendre l'exemple terminé pour ces types de déclarations de variables :

C#

```
// Multiple variables in a single declaration.
Type variable1 = expression1,
     variable2 = expression2;
// No initializer.
Type variable;
```

Ajoutez le code suivant au corps de la méthode `VisitLocalDeclarationStatement` pour ignorer la réécriture de ces formes de déclarations :

```
C#

if (node.Declaration.Variables.Count > 1)
{
    return node;
}
if (node.Declaration.Variables[0].Initializer == null)
{
    return node;
}
```

La méthode indique qu'aucune réécriture n'a lieu en retournant le paramètre `node` inchangé. Si aucune de ces expressions `if` n'est vraie, le nœud représente une déclaration possible avec initialisation. Ajoutez les instructions suivantes pour extraire le nom du type spécifié dans la déclaration et liez-le à l'aide du champ SemanticModel pour obtenir un symbole de type :

```
C#

var declarator = node.Declaration.Variables.First();
var variableTypeName = node.Declaration.Type;

var variableType = (ITypeSymbol)SemanticModel
    .GetSymbolInfo(variableTypeName)
    .Symbol;
```

Ajoutez maintenant cette instruction pour lier l'expression de l'initialiseur :

```
C#

var initializerInfo =
SemanticModel.GetTypeInfo(declarator.Initializer.Value);
```

Enfin, ajoutez l'instruction `if` suivante pour remplacer le nom de type existant par le mot clé `var` si le type de l'expression de l'initialiseur correspond au type spécifié :

```
C#

if (SymbolEqualityComparer.Default.Equals(variableType,
initializerInfo.Type))
{
    TypeSyntax varTypeName = SyntaxFactory.IdentifierName("var")
```

```
            .WithLeadingTrivia(variableTypeName.GetLeadingTrivia())
            .WithTrailingTrivia(variableTypeName.GetTrailingTrivia());

        return node.ReplaceNode(variableTypeName, varTypeName);
    }
    else
    {
        return node;
    }
```

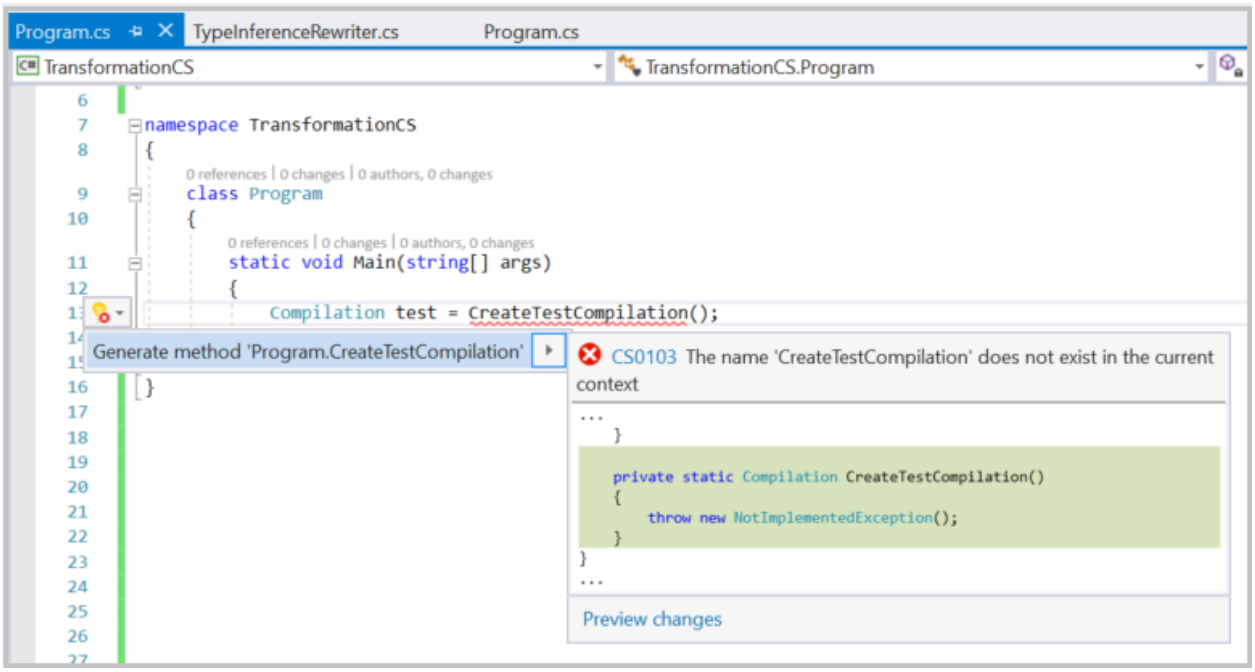
Cette condition est nécessaire car la déclaration peut effectuer un cast de l'expression de l'initialiseur dans une classe de base ou une interface. Si vous le souhaitez, les types à gauche et à droite de l'affectation ne correspondent pas. Dans ce cas, la suppression du type explicite modifierait la sémantique d'un programme. `var` est spécifié comme un identificateur plutôt qu'un mot clé car `var` est un mot clé contextuel. Les trivias de début et de fin (espaces blancs) sont transférés de l'ancien nom de type vers le mot clé `var` afin de conserver les espaces blancs verticaux et l'indentation. Il est plus simple d'utiliser `ReplaceNode` plutôt que `With*` pour transformer la LocalDeclarationStatementSyntax, car le nom de type est en fait le petit-enfant de l'instruction de déclaration.

Vous avez terminé le `TypeInferenceRewriter`. Revenez maintenant à votre fichier `Program.cs` pour terminer l'exemple. Créez un test Compilation et obtenez le SemanticModel à partir de celui-ci. Utilisez SemanticModel pour tester votre `TypeInferenceRewriter`. Vous effectuerez cette étape en dernier. En attendant, déclarez une variable d'espace réservé représentant votre compilation de test :

C#

```
Compilation test = CreateTestCompilation();
```

Après une pause, vous devriez voir apparaître une erreur sous forme de tilde, indiquant qu'il n'existe aucune méthode `CreateTestCompilation`. Appuyez sur **Ctrl+point** pour ouvrir l'ampoule, puis sur Entrée pour appeler la commande **Générer un stub de méthode**. Cette commande va générer un stub de méthode pour la méthode `CreateTestCompilation` dans la classe `Program`. Vous y reviendrez pour remplir cette méthode :



Écrivez le code suivant pour effectuer une itération sur chaque SyntaxTree dans le test de Compilation. Pour chacune d'elles, initialisez un nouveau `TypeInferenceRewriter` avec le SemanticModel pour cet arborescence :

C#

```
foreach (SyntaxTree sourceTree in test.SyntaxTrees)
{
    SemanticModel model = test.GetSemanticModel(sourceTree);

    TypeInferenceRewriter rewriter = new TypeInferenceRewriter(model);

    SyntaxNode newSource = rewriter.Visit(sourceTree.GetRoot());

    if (newSource != sourceTree.GetRoot())
    {
        File.WriteAllText(sourceTree.FilePath, newSource.ToFullString());
    }
}
```

Dans l'instruction `foreach` que vous avez créée, ajoutez le code suivant pour effectuer la transformation sur chaque arborescence source. Ce code écrit de façon conditionnelle la nouvelle arborescence transformée si des modifications ont été effectuées. Votre module de réécriture doit uniquement modifier une arborescence si elle rencontre une ou plusieurs déclarations de variables locales qui pourraient être simplifiées à l'aide de l'inférence de type :

C#

```
SyntaxNode newSource = rewriter.Visit(sourceTree.GetRoot());
```

```
if (newSource != sourceTree.GetRoot())
{
    File.WriteAllText(sourceTree.FilePath, newSource.ToFullString());
}
```

Vous devriez voir des tildes sous le code `File.WriteAllText`. Sélectionnez l'ampoule et ajoutez l'instruction `using System.IO;` nécessaire.

Vous avez presque terminé. Il ne reste plus qu'une étape : la création d'un Compilation test. Étant donné que vous n'avez pas utilisé du tout l'inférence de type pendant ce démarrage rapide, cela aurait été un cas de test parfait. Malheureusement, la création d'une compilation à partir d'un fichier de projet C# dépasse le cadre de cette procédure pas à pas. Heureusement, si vous avez suivi attentivement les instructions, il reste un espoir. Remplacez le contenu de la méthode `CreateTestCompilation` par le code suivant. Il crée une compilation de test correspondant par coïncidence au projet décrit dans ce démarrage rapide :

C#

```
String programPath = @"..\..\..\Program.cs";
String programText = File.ReadAllText(programPath);
SyntaxTree programTree =
               CSharpSyntaxTree.ParseText(programText)
                               .WithFilePath(programPath);

String rewriterPath = @"..\..\..\TypeInferenceRewriter.cs";
String rewriterText = File.ReadAllText(rewriterPath);
SyntaxTree rewriterTree =
               CSharpSyntaxTree.ParseText(rewriterText)
                               .WithFilePath(rewriterPath);

SyntaxTree[] sourceTrees = { programTree, rewriterTree };

MetadataReference mscorlib =
        MetadataReference.CreateFromFile(typeof(object).Assembly.Location);
MetadataReference codeAnalysis =

MetadataReference.CreateFromFile(typeof(SyntaxTree).Assembly.Location);
MetadataReference csharpCodeAnalysis =

MetadataReference.CreateFromFile(typeof(CSharpSyntaxTree).Assembly.Location)
;

MetadataReference[] references = { mscorlib, codeAnalysis,
csharpCodeAnalysis };

return CSharpCompilation.Create("TransformationCS",
    sourceTrees,
    references,
    new CSharpCompilationOptions(OutputKind.ConsoleApplication));
```

Croisez les doigts et exécutez le projet. Dans Visual Studio, choisissez **Déboguer**>**Démarrer le débogage**. Visual Studio devrait vous avertir que les fichiers de votre projet ont été modifiés. Cliquez sur «**Oui pour tout**» pour recharger les fichiers modifiés. Examinez ces fichiers pour vérifier la qualité de votre travail. Remarquez à quel point le code serait plus propre sans tous ces spécificateurs de type explicites et redondants.

Félicitations ! Vous avez utilisé les **API du compilateur** pour écrire votre propre refactorisation qui recherche certains modèles syntaxiques dans tous les fichiers d'un projet C#, analyse la sémantique du code source correspondant à ces modèles puis la transforme. Vous avez officiellement réussi à créer une refactorisation !

Tutoriel : Écrire votre premier analyseur et correctif de code

Article • 10/05/2023

Le SDK .NET Compiler Platform fournit les outils dont vous avez besoin pour créer des diagnostics personnalisés (analyseurs), des correctifs de code, une refactorisation du code et des suppresseurs de diagnostic qui ciblent le code C# ou Visual Basic. Un **analyseur** contient le code qui reconnaît les violations de votre règle. Votre **correctif de code** contient le code qui résout la violation. Les règles que vous implémentez peuvent aller de la structure du code au style de codage et aux conventions d'affectation de noms, et bien plus encore. .NET Compiler Platform fournit le framework permettant d'exécuter l'analyse alors que les développeurs écrivent du code, et toutes les fonctionnalités de l'IU Visual Studio pour corriger le code : afficher des tildes dans l'éditeur, renseigner la liste d'erreurs Visual Studio, créer des suggestions « ampoule » et afficher un aperçu détaillé des corrections suggérées.

Dans ce tutoriel, vous allez explorer la création d'un **analyseur** et d'un **correctif de code** associé à l'aide des API Roslyn. Un analyseur consiste à effectuer une analyse du code source et signaler un problème à l'utilisateur. Si vous le souhaitez, un correctif de code peut être associé à l'analyseur pour représenter une modification du code source de l'utilisateur. Ce tutoriel crée un analyseur qui recherche des déclarations de variables locales qui pourraient être déclarées à l'aide du modificateur `const` mais qui ne le sont pas. Le correctif de code associé modifie ces déclarations pour ajouter le modificateur `const`.

Prérequis

- Visual Studio 2019 version 16.8 ou ultérieure

Vous devez installer le **SDK .NET Compiler Platform** avec Visual Studio Installer :

Instructions d'installation - Visual Studio Installer

Vous pouvez rechercher **SDK .NET Compiler Platform** dans **Visual Studio Installer** de deux façons :

Vue Installation avec Visual Studio Installer - Charges de travail

SDK .NET Compiler Platform n'est pas sélectionné automatiquement dans le cadre de la charge de travail Développement d'extensions Visual Studio. Vous devez le sélectionner comme composant facultatif.

1. Exécutez **Visual Studio Installer**.
2. Sélectionnez **Modifier**
3. Choisissez la charge de travail **Développement d'extensions Visual Studio**.
4. Ouvrez le nœud **Développement d'extensions Visual Studio** dans l'arborescence résumée.
5. Cochez la case pour **SDK .NET Compiler Platform**. Il se trouve en dernier sous les composants facultatifs.

Si vous le souhaitez, vous pouvez ajouter **l'éditeur DGML** pour afficher des graphes dans le visualiseur :

1. Ouvrez le nœud **Composants individuels** dans l'arborescence résumée.
2. Cochez la case pour **Éditeur DGML**.

Onglet Installation avec Visual Studio Installer - Composants individuels

1. Exécutez **Visual Studio Installer**.
2. Sélectionnez **Modifier**
3. Sélectionnez l'onglet **Composants individuels**.
4. Cochez la case pour **SDK .NET Compiler Platform**. Il se trouve en haut de la liste, sous la section **Compilateurs, outils de génération et runtimes**.

Si vous le souhaitez, vous pouvez ajouter **l'éditeur DGML** pour afficher des graphes dans le visualiseur :

1. Cochez la case pour **Éditeur DGML**. Il se trouve sous la section **Outils de code**.

Il existe plusieurs étapes pour créer et valider votre analyseur :

1. Créez la solution.
2. Enregistrez le nom de l'analyseur et sa description.
3. Créez un rapport sur les avertissements et les recommandations de l'analyseur.
4. Implémentez le correctif de code pour accepter les recommandations.
5. Améliorez l'analyse par le biais de tests unitaires.

Créez la solution

- Dans Visual Studio, choisissez **Fichier** > **Nouveau** > **Projet...** pour afficher la boîte de dialogue Nouveau projet.
- Sous **Visual C#** > **Extensibilité**, choisissez **Analyseur avec correctif de code (.NET Standard)**.
- Nommez votre projet « **MakeConst** », puis cliquez sur OK.

> **Notes**
>
> Vous pouvez obtenir une erreur de compilation (*MSB4062 : Impossible de charger la tâche « CompareBuildTaskVersion »*). Pour la corriger, mettez à jour les packages NuGet dans la solution avec le Gestionnaire de package NuGet ou utilisez `Update-Package` dans la fenêtre Console du Gestionnaire de package.

Explorer le modèle d'analyseur

Le modèle d'analyseur avec correctif de code crée cinq projets :

- **MakeConst**, qui contient l'analyseur.
- **MakeConst.CodeFixes**, qui contient le correctif de code.
- **MakeConst.Package**, qui est utilisé pour produire un package NuGet pour l'analyseur et le correctif de code.
- **MakeConst.Test**, qui est un projet de test unitaire.
- **MakeConst.Vsix**, qui est le projet de démarrage par défaut qui démarre une deuxième instance de Visual Studio qui a chargé votre nouvel analyseur. Appuyez sur F5 pour démarrer le projet VSIX.

> **Notes**
>
> Les analyseurs doivent cibler .NET Standard 2.0, car ils peuvent s'exécuter dans les environnements .NET Core (builds en ligne de commande) et .NET Framework (Visual Studio).

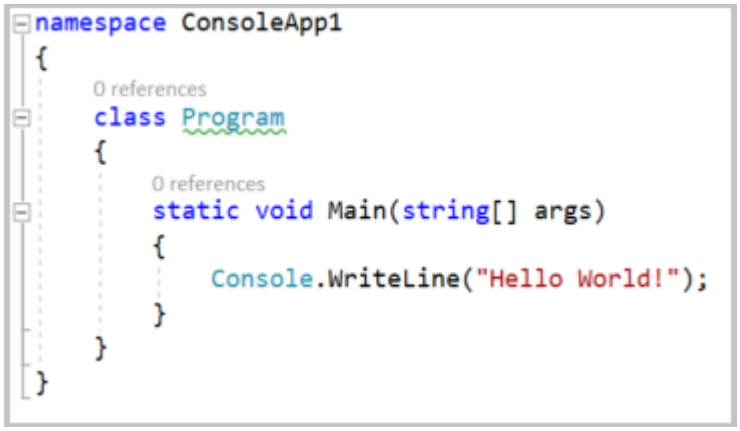
> **Conseil**
>
> Lorsque vous exécutez votre analyseur, vous démarrez une deuxième copie de Visual Studio. Cette deuxième copie utilise un hive de Registre différent pour stocker les paramètres. Cela vous permet de différencier les paramètres Visual dans

les deux copies de Visual Studio. Vous pouvez choisir un autre thème pour l'exécution expérimentale de Visual Studio. En outre, ne rendez pas vos paramètres itinérants et ne vous connectez pas à votre compte Visual Studio à l'aide de l'exécution expérimentale de Visual Studio. Cela permet de conserver les paramètres différents.

La ruche inclut non seulement l'analyseur en cours de développement, mais aussi tous les analyseurs précédents ouverts. Pour réinitialiser la ruche Roslyn, vous devez la supprimer manuellement depuis *%LocalAppData%\Microsoft\VisualStudio*. Le nom de dossier de la ruche Roslyn se termine par `Roslyn`, par exemple, `16.0_9ae182f9Roslyn`. Notez que vous devrez peut-être nettoyer la solution et la reconstruire après la suppression de la ruche.

Dans la deuxième instance de Visual Studio que vous venez de démarrer, créez un projet d'application console C# (toutes les versions cibles de .NET Framework iront, car les analyseurs fonctionnent au niveau de la source). Pointez sur le jeton avec un soulignement ondulé et le texte d'avertissement fourni par un analyseur apparaît.

Le modèle crée un analyseur qui émet un avertissement sur chaque déclaration de type dont le nom de type contient des lettres minuscules, comme indiqué dans la figure suivante :



Le modèle fournit également un correctif de code qui modifie n'importe quel nom de type contenant des caractères minuscules en majuscules. Vous pouvez cliquer sur l'ampoule affichée avec l'avertissement pour voir les modifications suggérées. Le fait d'accepter les modifications suggérées met à jour le nom de type et toutes les références à ce type dans la solution. Maintenant que vous avez vu l'analyseur initial en action, fermez la deuxième instance de Visual Studio et revenez à votre projet d'analyseur.

Vous n'êtes pas obligé de démarrer une deuxième copie de Visual Studio et de créer un nouveau code pour tester chaque modification de votre analyseur. Le modèle crée

également un projet de test unitaire pour vous. Ce projet contient deux tests. `TestMethod1` montre le format classique d'un test qui analyse le code sans déclencher un diagnostic. `TestMethod2` montre le format d'un test qui déclenche un diagnostic, puis applique un correctif de code proposé. À mesure que vous générez votre analyseur et votre correctif de code, vous écrirez des tests pour des structures de codes différentes afin de vérifier votre travail. Les tests unitaires pour les analyseurs sont beaucoup plus rapides que de les tester de manière interactive avec Visual Studio.

> **Conseil**
>
> Les tests unitaires d'analyseur sont un excellent outil lorsque vous savez quelles constructions de code doivent et ne doivent pas déclencher votre analyseur. Le chargement de votre analyseur dans une autre copie de Visual Studio est un excellent outil pour explorer et rechercher des constructions auxquelles vous n'avez peut-être pas encore pensé.

Dans ce tutoriel, vous écrivez un analyseur qui signale à l'utilisateur les déclarations de variables locales qui peuvent être converties en constantes locales. Considérons par exemple le code suivant :

C#

```
int x = 0;
Console.WriteLine(x);
```

Dans le code ci-dessus, une valeur constante est attribuée à `x` et n'est jamais modifiée. Elle peut être déclarée à l'aide du modificateur `const` :

C#

```
const int x = 0;
Console.WriteLine(x);
```

L'analyse pour déterminer si une variable peut être déclarée constante est impliquée, nécessitant une analyse syntaxique, une analyse constante de l'expression de l'initialiseur et une analyse du flux de données pour s'assurer que la variable n'est jamais accessible en écriture. .NET Compiler Platform fournit les API qui simplifient l'exécution de cette analyse.

Créer des enregistrements d'analyseur

Le modèle crée la classe `DiagnosticAnalyzer` initiale, dans le fichier *MakeConstAnalyzer.cs*. Cet analyseur initial montre deux propriétés importantes de chaque analyseur.

- Chaque analyseur de diagnostic doit fournir un attribut `[DiagnosticAnalyzer]` qui décrit le langage sur lequel il opère.
- Chaque analyseur de diagnostic doit dériver (directement ou indirectement) de la classe DiagnosticAnalyzer.

Le modèle montre également les fonctionnalités de base qui font partie de n'importe quel analyseur :

1. Enregistrer les actions. Les actions représentent les modifications de code qui doivent déclencher votre analyseur pour qu'il examine les violations dans le code. Lorsque Visual Studio détecte des modifications de code qui correspondent à une action enregistré, il appelle la méthode enregistrée de votre analyseur.
2. Créer des diagnostics. Quand votre analyseur détecte une violation, il crée un objet de diagnostic que Visual Studio utilise pour informer l'utilisateur de la violation.

Vous enregistrez des actions dans votre substitution de la méthode DiagnosticAnalyzer.Initialize(AnalysisContext). Dans ce tutoriel, vous allez consulter des **nœuds de syntaxe** pour rechercher des déclarations locales et voir lesquelles ont des valeurs constantes. Si une déclaration peut être une constante, votre analyseur crée et signale un diagnostic.

La première étape consiste à mettre à jour les constantes d'inscription et la méthode `Initialize` pour que ces constantes indiquent votre analyseur « Make Const ». La plupart des constantes de chaîne est définie dans le fichier de ressources de chaîne. Cette pratique facilite la localisation. Ouvrez le fichier *Resources.resx* pour le projet d'analyseur **MakeConst**. Cela affiche l'éditeur de ressources. Mettez à jour les ressources de chaîne comme suit :

- Remplacez `AnalyzerDescription` par « Variables that are not modified should be made constants. ».
- Remplacez `AnalyzerMessageFormat` par « Variable '{0}' can be made constant ».
- Remplacez `AnalyzerTitle` par « Variable can be made constant ».

Lorsque vous avez terminé, l'éditeur de ressources doit apparaître tel que montré dans l'illustration suivante :

	Name	Value	Comment
	AnalyzerDescription	Variables that are not modified should be made constants.	An optional longer localizable description of the diagnostic.
	AnalyzerMessageFormat	Variable '{0}' can be made constant	The format-able message the diagnostic displays.
▶	AnalyzerTitle	Variable can be made constant	The title of the diagnostic.
*			

Les modifications restantes ont lieu dans le fichier de l'analyseur. Ouvrez *MakeConstAnalyzer.cs* dans Visual Studio. Modifiez l'action enregistrée d'une action qui agit sur les symboles à une action qui agit sur la syntaxe. Dans la méthode `MakeConstAnalyzerAnalyzer.Initialize`, recherchez la ligne qui enregistre l'action sur les symboles :

C#

```
context.RegisterSymbolAction(AnalyzeSymbol, SymbolKind.NamedType);
```

Remplacez-la par la ligne suivante :

C#

```
context.RegisterSyntaxNodeAction(AnalyzeNode,
SyntaxKind.LocalDeclarationStatement);
```

Après cette modification, vous pouvez supprimer la méthode `AnalyzeSymbol`. Cet outil examine SyntaxKind.LocalDeclarationStatement, et non les instructions SymbolKind.NamedType. Notez que `AnalyzeNode` est souligné par des traits ondulés rouges. Le code que vous venez d'ajouter référence une méthode `AnalyzeNode` qui n'a pas été déclarée. Déclarez cette méthode en utilisant le code suivant :

C#

```
private void AnalyzeNode(SyntaxNodeAnalysisContext context)
{
}
```

Remplacez `Category` par « Usage » dans *MakeConstAnalyzer.cs*, comme montré dans le code suivant :

C#

```
private const string Category = "Usage";
```

Rechercher des déclarations locales qui pourraient être constantes

Il est temps d'écrire la première version de la méthode `AnalyzeNode`. Elle doit rechercher une déclaration locale unique qui peut être `const` mais ne l'est pas, comme le code suivant :

```
C#

int x = 0;
Console.WriteLine(x);
```

La première étape consiste à rechercher les déclarations locales. Ajoutez le code suivant à `AnalyzeNode` dans *MakeConstAnalyzer.cs* :

```
C#

var localDeclaration = (LocalDeclarationStatementSyntax)context.Node;
```

Ce cast réussit toujours, car votre analyseur a enregistré les modifications apportées aux déclarations locales, et uniquement les déclarations locales. Aucun autre type de nœud ne déclenche un appel à votre méthode `AnalyzeNode`. Ensuite, vérifiez la présence de modificateurs `const` dans la déclaration. Si vous en trouvez, retournez immédiatement. Le code suivant recherche les modificateurs `const` sur la déclaration locale :

```
C#

// make sure the declaration isn't already const:
if (localDeclaration.Modifiers.Any(SyntaxKind.ConstKeyword))
{
    return;
}
```

Enfin, vous devez vérifier que la variable peut être `const`. Cela signifie de s'assurer qu'elle n'est jamais assignée après son initialisation.

Vous allez effectuer une analyse sémantique à l'aide de SyntaxNodeAnalysisContext. Vous utilisez l'argument `context` pour déterminer si la déclaration de variable locale peut être rendue `const`. Microsoft.CodeAnalysis.SemanticModel représente toutes les informations sémantiques dans un seul fichier source. Vous pouvez en apprendre plus dans l'article qui traite des modèles sémantiques. Vous allez utiliser Microsoft.CodeAnalysis.SemanticModel pour effectuer une analyse de flux de données

sur l'instruction de déclaration locale. Ensuite, vous utilisez les résultats de cette analyse de flux de données pour vous assurer que la variable locale n'est pas écrite avec une nouvelle valeur ailleurs. Appelez la méthode d'extension GetDeclaredSymbol pour récupérer ILocalSymbol pour la variable et vérifiez qu'il n'est pas contenu avec la collection DataFlowAnalysis.WrittenOutside de l'analyse de flux de données. Ajoutez le code suivant à la fin de la méthode `AnalyzeNode` :

C#

```
// Perform data flow analysis on the local declaration.
DataFlowAnalysis dataFlowAnalysis =
context.SemanticModel.AnalyzeDataFlow(localDeclaration);

// Retrieve the local symbol for each variable in the local declaration
// and ensure that it is not written outside of the data flow analysis
region.
VariableDeclaratorSyntax variable =
localDeclaration.Declaration.Variables.Single();
ISymbol variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable,
context.CancellationToken);
if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
{
    return;
}
```

Le code qui vient d'être ajouté garantit que la variable n'est pas modifiée et peut par conséquent devenir `const`. Il est temps de générer le diagnostic. Ajoutez le code suivant comme dernière ligne dans `AnalyzeNode` :

C#

```
context.ReportDiagnostic(Diagnostic.Create(Rule, context.Node.GetLocation(),
localDeclaration.Declaration.Variables.First().Identifier.ValueText));
```

Vous pouvez vérifier votre progression en appuyant sur F5 pour exécuter votre analyseur. Vous pouvez charger l'application console que vous avez créée précédemment et ajouter le code de test suivant :

C#

```
int x = 0;
Console.WriteLine(x);
```

L'ampoule doit apparaître et votre analyseur doit signaler un diagnostic. Toutefois, en fonction de votre version de Visual Studio, vous verrez :

- L'ampoule, qui utilise toujours le correctif de code généré par le modèle et vous indique qu'il peut être converti en majuscules.
- Un message de bannière en haut de l'éditeur indiquant que « MakeConstCodeFixProvider a rencontré une erreur et a été désactivé ». C'est parce que le fournisseur de correctifs de code n'a pas encore été modifié et s'attend toujours à trouver les éléments `TypeDeclarationSyntax` au lieu des éléments `LocalDeclarationStatementSyntax`.

La section suivante explique comment écrire le correctif de code.

Écrire le correctif de code

Un analyseur peut fournir un ou plusieurs correctifs de code. Un correctif de code définit une modification qui traite le problème signalé. Pour l'analyseur que vous avez créé, vous pouvez fournir un correctif de code qui insère le mot clé const :

diff

```
- int x = 0;
+ const int x = 0;
Console.WriteLine(x);
```

L'utilisateur le choisit à partir de l'IU ampoule dans l'éditeur et Visual Studio modifie le code.

Ouvrez le fichier *CodeFixResources.resx* et remplacez `CodeFixTitle` par « Make constant ».

Ouvrez le fichier *MakeConstCodeFixProvider.cs* ajouté par le modèle. Ce correctif de code est déjà associé à l'ID de diagnostic produit par votre analyseur de diagnostic, mais il n'implémente pas encore la transformation de code adéquate.

Ensuite, supprimez la méthode `MakeUppercaseAsync`. Elle n'est plus applicable.

Tous les fournisseurs de correctif de code dérivent de CodeFixProvider. Ils substituent tous CodeFixProvider.RegisterCodeFixesAsync(CodeFixContext) pour signaler les correctifs de code disponibles. Dans `RegisterCodeFixesAsync`, remplacez le type de nœud ancêtre que vous recherchez par LocalDeclarationStatementSyntax pour faire correspondre le diagnostic :

C#

```
var declaration =
root.FindToken(diagnosticSpan.Start).Parent.AncestorsAndSelf().OfType<LocalD
```

```
eclarationStatementSyntax>().First();
```

Ensuite, modifiez la dernière ligne pour enregistrer un correctif de code. Votre correctif crée un nouveau document obtenu en ajoutant le modificateur `const` à une déclaration existante :

C#

```
// Register a code action that will invoke the fix.
context.RegisterCodeFix(
    CodeAction.Create(
        title: CodeFixResources.CodeFixTitle,
        createChangedDocument: c => MakeConstAsync(context.Document,
declaration, c),
        equivalenceKey: nameof(CodeFixResources.CodeFixTitle)),
    diagnostic);
```

Notez les traits de soulignement ondulés rouges dans le code que vous venez d'ajouter sur le symbole `MakeConstAsync`. Ajoutez une déclaration pour `MakeConstAsync` comme le code suivant :

C#

```
private static async Task<Document> MakeConstAsync(Document document,
    LocalDeclarationStatementSyntax localDeclaration,
    CancellationToken cancellationToken)
{
}
```

Votre nouvelle méthode `MakeConstAsync` transformera Document qui représente le fichier source de l'utilisateur en un nouveau Document qui contient à présent une déclaration `const`.

Vous créez un nouveau jeton de mot clé `const` à insérer au début de l'instruction de déclaration. Veillez tout d'abord à supprimer les trivia de début dans le premier jeton de l'instruction de déclaration et à les associer au jeton `const`. Ajoutez le code suivant à la méthode `MakeConstAsync` :

C#

```
// Remove the leading trivia from the local declaration.
SyntaxToken firstToken = localDeclaration.GetFirstToken();
SyntaxTriviaList leadingTrivia = firstToken.LeadingTrivia;
LocalDeclarationStatementSyntax trimmedLocal =
localDeclaration.ReplaceToken(
    firstToken, firstToken.WithLeadingTrivia(SyntaxTriviaList.Empty));
```

```
// Create a const token with the leading trivia.
SyntaxToken constToken = SyntaxFactory.Token(leadingTrivia,
SyntaxKind.ConstKeyword,
SyntaxFactory.TriviaList(SyntaxFactory.ElasticMarker));
```

Ensuite, ajoutez le jeton `const` dans la déclaration à l'aide du code suivant :

C#

```
// Insert the const token into the modifiers list, creating a new modifiers
list.
SyntaxTokenList newModifiers = trimmedLocal.Modifiers.Insert(0, constToken);
// Produce the new local declaration.
LocalDeclarationStatementSyntax newLocal = trimmedLocal
    .WithModifiers(newModifiers)
    .WithDeclaration(localDeclaration.Declaration);
```

Ensuite, mettez en forme la nouvelle déclaration en fonction des règles de mise en forme en C#. La mise en forme de vos modifications pour correspondre au code existant crée une meilleure expérience. Ajoutez l'instruction suivante immédiatement après le code existant :

C#

```
// Add an annotation to format the new local declaration.
LocalDeclarationStatementSyntax formattedLocal =
newLocal.WithAdditionalAnnotations(Formatter.Annotation);
```

Un nouvel espace de noms est requis pour ce code. Ajoutez la directive `using` suivante en haut du fichier :

C#

```
using Microsoft.CodeAnalysis.Formatting;
```

L'étape finale consiste à apporter votre modification. Ce processus comprend trois étapes :

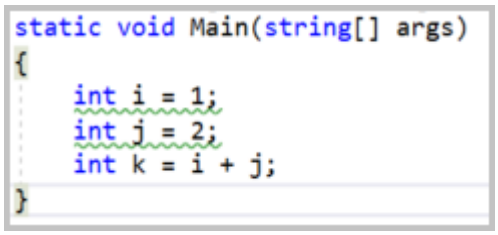
1. Obtenir un handle pour le document existant.
2. Créer un document en remplaçant la déclaration existante par la nouvelle déclaration.
3. Retourner le nouveau document.

Ajoutez le code suivant à la fin de la méthode `MakeConstAsync` :

C#

```
// Replace the old local declaration with the new local declaration.
SyntaxNode oldRoot = await
document.GetSyntaxRootAsync(cancellationToken).ConfigureAwait(false);
SyntaxNode newRoot = oldRoot.ReplaceNode(localDeclaration, formattedLocal);

// Return document with transformed tree.
return document.WithSyntaxRoot(newRoot);
```

Votre correctif de code est prêt à être testé. Appuyez sur F5 pour exécuter le projet d'analyseur dans une deuxième instance de Visual Studio. Dans la deuxième instance de Visual Studio, créez un projet Application console C# et ajoutez quelques déclarations de variables locales initialisées avec des valeurs constantes à la méthode Main. Vous verrez que ces éléments sont signalés en tant qu'avertissements comme indiqué ci-dessous.



Vous avez bien progressé. Des traits de soulignement ondulés s'affichent sous les déclarations qui peuvent devenir `const`. Mais il reste encore du travail. Cela fonctionne si vous ajoutez `const` aux déclarations qui commencent par `i`, puis `j` et enfin `k`. Mais, si le modificateur `const` est ajouté dans un autre ordre, en commençant par `k`, l'analyseur crée des erreurs : `k` ne peut pas être déclaré `const`, sauf si `i` et `j` sont tous deux déjà `const`. Vous devez effectuer d'autres analyses afin de vous assurer que vous gérez les différentes façons dont les variables peuvent être déclarées et initialisées.

Créer des tests unitaires

Votre analyseur et votre correctif de code travaillent sur un cas simple d'une déclaration unique qui peut être devenir constante. Il existe de nombreuses instructions de déclaration possibles où cette implémentation commet des erreurs. Vous allez traiter ces cas en travaillant avec la bibliothèque de tests unitaires écrite par le modèle. C'est beaucoup plus rapide que d'ouvrir plusieurs fois une deuxième copie de Visual Studio.

Ouvrez le fichier *MakeConstUnitTests.cs* dans le projet de test unitaire. Le modèle a créé deux tests qui suivent les deux modèles courants pour un test unitaire d'analyseur et de correctif de code. `TestMethod1` montre le modèle pour un test qui garantit que

l'analyseur ne signale pas un diagnostic lorsqu'il ne le devrait pas. `TestMethod2` montre le modèle pour créer un rapport de diagnostic et exécuter le correctif de code.

Le modèle utilise les packages Microsoft.CodeAnalysis.Testing pour les tests unitaires.

> **Conseil**
>
> La bibliothèque de tests prend en charge une syntaxe de balisage spéciale, notamment :
>
> - `[|text|]` : indique qu'un diagnostic est signalé pour `text`. Par défaut, ce formulaire peut être utilisé seulement pour tester des analyseurs avec exactement un `DiagnosticDescriptor` fourni par `DiagnosticAnalyzer.SupportedDiagnostics`.
> - `{|ExpectedDiagnosticId:text|}` : indique qu'un diagnostic avec Id `ExpectedDiagnosticId` est signalé pour `text`.

Remplacez les tests de modèle dans la classe `MakeConstUnitTest` par la méthode de test suivante :

C#

```
        [TestMethod]
        public async Task LocalIntCouldBeConstant_Diagnostic()
        {
            await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
    static void Main()
    {
        [|int i = 0;|]
        Console.WriteLine(i);
    }
}
", @"
using System;

class Program
{
    static void Main()
    {
        const int i = 0;
        Console.WriteLine(i);
    }
}
```

```
");
        }
```

Exécutez ce test pour vous assurer qu'il réussit. Dans Visual Studio, ouvrez **l'Explorateur de tests** en sélectionnant **Test**>**Windows**>**Explorateur de tests**. Ensuite, sélectionnez **Exécuter tout**.

Créer des tests pour les déclarations valides

En règle générale, les analyseurs doivent s'arrêter aussi rapidement que possible, en effectuant un minimum de travail. Visual Studio appelle les analyseurs enregistrés lorsque l'utilisateur modifie le code. La réactivité est essentielle. Il existe plusieurs cas de test pour le code qui ne doivent pas déclencher votre diagnostic. Votre analyseur a déjà géré l'un de ces tests, le cas où une variable est assignée après avoir été initialisée. Ajoutez la méthode de test suivante pour représenter ce cas :

C#

```
        [TestMethod]
        public async Task VariableIsAssigned_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int i = 0;
        Console.WriteLine(i++);
    }
}
");
        }
```

Ce test réussit également. Ensuite, ajoutez des méthodes de test pour les conditions que vous n'avez pas encore gérées :

- Déclarations qui sont déjà `const`, car elles sont déjà constantes :

  C#

  ```
          [TestMethod]
          public async Task VariableIsAlreadyConst_NoDiagnostic()
          {
              await VerifyCS.VerifyAnalyzerAsync(@"
  using System;
  ```

```
class Program
{
    static void Main()
    {
        const int i = 0;
        Console.WriteLine(i);
    }
}
");
        }
```

- Déclarations sans initialiseur, car il n'existe aucune valeur à utiliser :

```
C#

        [TestMethod]
        public async Task NoInitializer_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int i;
        i = 0;
        Console.WriteLine(i);
    }
}
");
        }
```

- Déclarations où l'initialiseur n'est pas une constante, car elles ne peuvent pas être des constantes de compilation :

```
C#

        [TestMethod]
        public async Task InitializerIsNotConstant_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int i = DateTime.Now.DayOfYear;
        Console.WriteLine(i);
```

```
    }
}
");
        }
```

Cela peut être encore plus compliqué, car C# accepte que plusieurs déclarations forment une seule instruction. Prenez en compte la constante de chaîne de cas de test suivante :

C#

```
        [TestMethod]
        public async Task MultipleInitializers_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int i = 0, j = DateTime.Now.DayOfYear;
        Console.WriteLine(i);
        Console.WriteLine(j);
    }
}
");
        }
```

La variable `i` peut devenir constante, mais la variable `j` ne le peut pas. Par conséquent, cette instruction ne peut pas devenir une déclaration constante.

Réexécutez vos tests, et vous verrez ces nouveaux cas de test échouent.

Mettre à jour votre analyseur afin d'ignorer les déclarations correctes

Vous avez besoin de certaines améliorations dans la méthode `AnalyzeNode` de votre analyseur pour éliminer le code correspondant à ces conditions. Ce sont toutes des conditions associées, donc des modifications similaires résoudront toutes ces conditions. Dans `AnalyzeNode`, effectuez les changements suivants :

- Votre analyse sémantique a examiné une déclaration de variable unique. Ce code doit se trouver dans une boucle `foreach` qui examine toutes les variables déclarées dans la même instruction.

- Chaque variable déclarée doit avoir un initialiseur.
- L'initialiseur de chaque variable déclarée doit être une constante de compilation.

Dans votre méthode `AnalyzeNode`, remplacez l'analyse sémantique d'origine :

C#

```
// Perform data flow analysis on the local declaration.
DataFlowAnalysis dataFlowAnalysis =
context.SemanticModel.AnalyzeDataFlow(localDeclaration);

// Retrieve the local symbol for each variable in the local declaration
// and ensure that it is not written outside of the data flow analysis
region.
VariableDeclaratorSyntax variable =
localDeclaration.Declaration.Variables.Single();
ISymbol variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable,
context.CancellationToken);
if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
{
    return;
}
```

par l'extrait de code suivant :

C#

```
// Ensure that all variables in the local declaration have initializers that
// are assigned with constant values.
foreach (VariableDeclaratorSyntax variable in
localDeclaration.Declaration.Variables)
{
    EqualsValueClauseSyntax initializer = variable.Initializer;
    if (initializer == null)
    {
        return;
    }

    Optional<object> constantValue =
context.SemanticModel.GetConstantValue(initializer.Value,
context.CancellationToken);
    if (!constantValue.HasValue)
    {
        return;
    }
}

// Perform data flow analysis on the local declaration.
DataFlowAnalysis dataFlowAnalysis =
context.SemanticModel.AnalyzeDataFlow(localDeclaration);

foreach (VariableDeclaratorSyntax variable in
```

```
localDeclaration.Declaration.Variables)
{
    // Retrieve the local symbol for each variable in the local declaration
    // and ensure that it is not written outside of the data flow analysis
region.
    ISymbol variableSymbol =
context.SemanticModel.GetDeclaredSymbol(variable,
context.CancellationToken);
    if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
    {
        return;
    }
}
```

La première boucle `foreach` examine chaque déclaration de variable à l'aide de l'analyse syntaxique. La première vérification garantit que la variable a un initialiseur. La deuxième vérification garantit que l'initialiseur est une constante. La deuxième boucle contient l'analyse sémantique d'origine. Les vérifications sémantiques sont dans une boucle distincte, car elles ont un impact plus important sur les performances. Réexécutez vos tests. Ils doivent tous réussir.

Ajouter la touche finale

Vous avez presque terminé. Il existe quelques conditions de plus que votre analyseur doit gérer. Visual Studio appelle les analyseurs lorsque l'utilisateur écrit du code. Souvent, votre analyseur sera appelé lorsque du code ne se compile pas. La méthode `AnalyzeNode` de votre analyseur de diagnostic ne vérifie pas si la valeur de constante est convertible en type de variable. Par conséquent, l'implémentation actuelle convertit sans problème une déclaration incorrecte comme `int i = "abc"` en constante locale. Ajoutez une méthode de test pour ce cas :

C#

```
        [TestMethod]
        public async Task DeclarationIsInvalid_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int x = {|CS0029:""abc""|};
    }
}
```

```
");
        }
```

En outre, les types de référence ne sont pas gérés correctement. La seule valeur de constante autorisée pour un type de référence est `null`, sauf dans le cas de System.String, qui autorise des littéraux de chaîne. En d'autres termes, `const string s = "abc"` est légal, mais pas `const object s = "abc"`. Cet extrait de code vérifie cette condition :

C#

```
        [TestMethod]
        public async Task DeclarationIsNotString_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        object s = ""abc"";
    }
}
");
        }
```

Pour être exhaustif, vous devez ajouter un autre test afin de vous assurer que vous pouvez créer une déclaration de constante pour une chaîne. L'extrait de code suivant définit le code qui déclenche le diagnostic et le code une fois que le correctif a été appliqué :

C#

```
        [TestMethod]
        public async Task StringCouldBeConstant_Diagnostic()
        {
            await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
    static void Main()
    {
        [|string s = ""abc"";|]
    }
}
", @"
using System;
```

```
class Program
{
    static void Main()
    {
        const string s = ""abc"";
    }
}
");
        }
```

Enfin, si une variable est déclarée avec le mot clé `var`, le correctif de code fait la mauvaise chose et génère une déclaration `const var`, qui n'est pas prise en charge par le langage C#. Pour corriger ce bogue, le correctif de code doit remplacer le mot clé `var` par le nom du type déduit :

C#

```
        [TestMethod]
        public async Task VarIntDeclarationCouldBeConstant_Diagnostic()
        {
            await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
    static void Main()
    {
        [|var item = 4;|]
    }
}
", @"
using System;

class Program
{
    static void Main()
    {
        const int item = 4;
    }
}
");
        }

        [TestMethod]
        public async Task VarStringDeclarationCouldBeConstant_Diagnostic()
        {
            await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
```

```
    static void Main()
    {
        [|var item = ""abc"";|]
    }
}
", @"
using System;

class Program
{
    static void Main()
    {
        const string item = ""abc"";
    }
}
");
        }
```

Heureusement, tous les bogues ci-dessus peuvent être gérés à l'aide des mêmes techniques que celles que vous venez d'apprendre.

Pour corriger le premier bogue, commencez par ouvrir *MakeConstAnalyzer.cs* et recherchez la boucle foreach où chacun des initialiseurs de la déclaration locale est vérifié pour s'assurer que des valeurs constantes leur sont attribués. Immédiatement *avant* la première boucle foreach, appelez `context.SemanticModel.GetTypeInfo()` pour récupérer des informations détaillées sur le type déclaré de la déclaration locale :

C#

```
TypeSyntax variableTypeName = localDeclaration.Declaration.Type;
ITypeSymbol variableType =
context.SemanticModel.GetTypeInfo(variableTypeName,
context.CancellationToken).ConvertedType;
```

Ensuite, dans votre boucle `foreach`, vérifiez chaque initialiseur pour vous assurer qu'il peut être converti dans le type de variable. Ajoutez la vérification suivante après avoir vérifié que l'initialiseur est une constante :

C#

```
// Ensure that the initializer value can be converted to the type of the
// local declaration without a user-defined conversion.
Conversion conversion =
context.SemanticModel.ClassifyConversion(initializer.Value, variableType);
if (!conversion.Exists || conversion.IsUserDefined)
{
    return;
}
```

La modification suivante s'appuie sur la précédente. Avant l'accolade fermante de la première boucle foreach, ajoutez le code suivant pour vérifier le type de la déclaration locale si la constante est une chaîne ou une valeur null.

C#

```
// Special cases:
//  * If the constant value is a string, the type of the local declaration
//    must be System.String.
//  * If the constant value is null, the type of the local declaration must
//    be a reference type.
if (constantValue.Value is string)
{
    if (variableType.SpecialType != SpecialType.System_String)
    {
        return;
    }
}
else if (variableType.IsReferenceType && constantValue.Value != null)
{
    return;
}
```

Vous devez écrire un peu plus de code dans votre fournisseur de correctif de code pour remplacer le mot clé `var` par le nom de type correct. Revenez à *MakeConstCodeFixProvider.cs*. Le code que vous ajouterez effectue les étapes suivantes :

- Vérifiez si la déclaration est une déclaration `var`, et dans ce cas :
- Créez un nouveau type pour le type déduit.
- Assurez-vous que la déclaration de type n'est pas un alias. Le cas échéant, il est permis de déclarer `const var`.
- Assurez-vous que `var` n'est pas un nom de type dans ce programme. (Dans ce cas, `const var` est autorisé).
- Simplifier le nom de type complet

On dirait que beaucoup de code est nécessaire pour cela. Ce n'est pas le cas. Remplacez la ligne qui déclare et initialise `newLocal` par le code suivant. Il se place immédiatement après l'initialisation de `newModifiers` :

C#

```
// If the type of the declaration is 'var', create a new type name
// for the inferred type.
VariableDeclarationSyntax variableDeclaration =
localDeclaration.Declaration;
TypeSyntax variableTypeName = variableDeclaration.Type;
if (variableTypeName.IsVar)
```

```
{
    SemanticModel semanticModel = await
document.GetSemanticModelAsync(cancellationToken).ConfigureAwait(false);

    // Special case: Ensure that 'var' isn't actually an alias to another
type
    // (e.g. using var = System.String).
    IAliasSymbol aliasInfo = semanticModel.GetAliasInfo(variableTypeName,
cancellationToken);
    if (aliasInfo == null)
    {
        // Retrieve the type inferred for var.
        ITypeSymbol type = semanticModel.GetTypeInfo(variableTypeName,
cancellationToken).ConvertedType;

        // Special case: Ensure that 'var' isn't actually a type named
'var'.
        if (type.Name != "var")
        {
            // Create a new TypeSyntax for the inferred type. Be careful
            // to keep any leading and trailing trivia from the var keyword.
            TypeSyntax typeName =
SyntaxFactory.ParseTypeName(type.ToDisplayString())
                .WithLeadingTrivia(variableTypeName.GetLeadingTrivia())
                .WithTrailingTrivia(variableTypeName.GetTrailingTrivia());

            // Add an annotation to simplify the type name.
            TypeSyntax simplifiedTypeName =
typeName.WithAdditionalAnnotations(Simplifier.Annotation);

            // Replace the type in the variable declaration.
            variableDeclaration =
variableDeclaration.WithType(simplifiedTypeName);
        }
    }
}
// Produce the new local declaration.
LocalDeclarationStatementSyntax newLocal =
trimmedLocal.WithModifiers(newModifiers)
                            .WithDeclaration(variableDeclaration);
```

Vous devez ajouter une directive `using` pour utiliser le type Simplifier :

C#

```
using Microsoft.CodeAnalysis.Simplification;
```

Exécutez vos tests. Ils doivent tous réussir. Félicitez-vous en exécutant votre analyseur terminé. Appuyez sur `Ctrl`+`F5` pour exécuter le projet d'analyseur dans une deuxième instance de Visual Studio avec l'extension Roslyn Preview chargée.

- Dans la deuxième instance de Visual Studio, créez un projet Application console C# et ajoutez `int x = "abc";` à la méthode Main. Grâce à la correction du premier bogue, aucun avertissement ne doit être signalé pour cette déclaration de variable locale (même s'il existe une erreur du compilateur comme prévu).
- Ensuite, ajoutez `object s = "abc";` à la méthode Main. En raison de la correction du deuxième bogue, aucun avertissement ne doit être signalé.
- Enfin, ajoutez une autre variable locale qui utilise le mot clé `var`. Un avertissement est signalé et une suggestion apparaît en dessous à gauche.
- Déplacez le signe d'insertion de l'éditeur sur le trait de soulignement ondulé et appuyez sur `Ctrl`+`.`. pour afficher le correctif de code suggéré. Lors de la sélection de votre correctif de code, notez que le mot clé `var` est désormais géré correctement.

Enfin, ajoutez le code suivant :

C#

```
int i = 2;
int j = 32;
int k = i + j;
```

Après ces modifications, vous obtenez des traits de soulignement ondulés rouges uniquement sur les deux premières variables. Ajoutez `const` à `i` et à `j`, et vous obtenez un nouvel avertissement sur `k`, car cet élément peut désormais devenir `const`.

Félicitations ! Vous avez créé votre première extension .NET Compiler Platform qui effectue une analyse de code à la volée pour détecter un problème et fournit une solution rapide pour résoudre ce problème. Tout au long du processus, vous avez découvert la plupart des API de code qui font partie du SDK .NET Compiler Platform (API Roslyn). Vous pouvez comparer votre travail avec l'exemple terminé dans notre référentiel GitHub d'exemples.

Autres ressources

- Bien démarrer avec l'analyse de la syntaxe
- Bien démarrer avec l'analyse sémantique

Guide de programmation C#

Article • 25/02/2023

Cette section fournit des informations détaillées sur les fonctionnalités clés du langage C# et les fonctionnalités accessibles à C# via .NET.

La majeure partie de cette section suppose que vous connaissez déjà en partie C# et les concepts de programmation généraux. Si vous débutez avec la programmation ou avec C#, vous pouvez consulter Introduction aux tutoriels C# ou le tutoriel .NET sur navigateur, où aucune connaissance préalable en programmation n'est nécessaire.

Pour plus d'informations sur des mots clés, opérateurs et directives de préprocesseur spécifiques, consultez Informations de référence sur C#. Pour plus d'informations sur la spécification du langage C#, consultez la page Spécification du langage C#.

Sections du programme

À l'intérieur d'un programme C#

Main() et arguments de ligne de commande

Sections du langage

InstructionsOpérateurs et expressionsMembres expression-bodiedComparaisons d'égalité

Types

Programmation orientée objet

Interfaces

Délégués

Tableaux

Chaînes

Propriétés

Indexeurs

Événements

Génériques

Itérateurs

Expressions de requête LINQ

Espaces de noms

Pointeurs et code unsafe

Commentaires sur la documentation XML

Sections de la plateforme

Domaines d'application

Assemblys dans .NET

Attributs

Regroupements

Exceptions et gestion des exceptions

Système de fichiers et Registre (Guide de programmation C#)

Interopérabilité

Réflexion

Voir aussi

- Référence C#

Concepts de programmation (C#)

Article • 25/02/2023

Cette section explique les concepts de programmation du langage C#.

Dans cette section

Titre	Description
Assemblys dans .NET	Explique comment créer et utiliser des assemblys.
Programmation asynchrone avec Async et Await (C#)	Explique comment écrire des solutions asynchrones à l'aide des mots clés async et await en C#. Comprend une procédure pas à pas.
Attributs (C#)	Explique comment fournir des informations supplémentaires sur des éléments de programmation tels que les types, les champs, les méthodes et les propriétés à l'aide d'attributs.
Collections (C#)	Décrit quelques-uns des types de collections fournis par .NET. Montre comment utiliser des collections simples et des collections de paires clé/valeur.
Covariance et contravariance (C#)	Montre comment activer la conversion implicite de paramètres de type générique dans les interfaces et les délégués.
Arborescences d'expressions (C#)	Explique comment utiliser des arborescences d'expressions pour activer la modification dynamique de code exécutable.
Itérateurs (C#)	Décrit les itérateurs, utilisés pour parcourir les collections et retourner les éléments un par un.
LINQ (Language-Integrated Query) (C#)	Décrit les puissantes capacités de requêtage de la syntaxe du langage C# et le modèle d'interrogation de bases de données relationnelles, de documents XML, de jeux de données et de collections en mémoire.
Réflexion (C#)	Explique comment utiliser la réflexion pour créer dynamiquement une instance d'un type, pour lier le type à un objet existant ou pour obtenir le type d'un objet existant, et comment invoquer ses méthodes ou accéder à ses champs et propriétés.
Sérialisation (C#)	Décrit les concepts clés de la sérialisation binaire, XML et SOAP.

Sections connexes

- Conseils sur les performances

  Décrit plusieurs règles de base qui peuvent aider à augmenter les performances des applications.

Programmation asynchrone

Article • 10/05/2023

Si vous avez besoin de code utilisant les E/S de manière intensive (par exemple, pour récupérer des données d'un réseau, accéder à une base de données, ou lire et écrire dans un fichier), optez pour la programmation asynchrone. L'écriture de code asynchrone est également indiquée si votre code utilise le processeur de manière intensive, notamment pour effectuer un calcul complexe.

C# fournit un modèle de programmation asynchrone de niveau de langage. Ce modèle facilite l'écriture de code asynchrone, en vous évitant d'effectuer des rappels multiples ou de vous conformer à une bibliothèque qui prend en charge l'asynchronisme. Il suit ce que l'on appelle le modèle asynchrone basé sur des tâches (TAP).

Vue d'ensemble du modèle asynchrone

La programmation asynchrone est basée sur les objets `Task` et `Task<T>`, qui modélisent les opérations asynchrones. Ces objets sont exposés à l'aide des mots clés `async` et `await`. Dans la plupart des cas, le modèle est assez simple :

- Pour du code utilisant les E/S de manière intensive, vous spécifiez une opération qui retourne un objet `Task` ou `Task<T>` dans une méthode `async`.
- Pour du code utilisant le processeur de manière intensive, vous spécifiez une opération qui est démarrée sur un thread d'arrière-plan avec la méthode Task.Run.

Le mot clé `await` trouve ici toute son utilité. Il cède le contrôle à l'appelant de la méthode qui a effectué l'opération `await`. Au final, c'est ce qui rend une interface utilisateur réactive ou un service élastique. Bien qu'il existe des façons d'aborder le code asynchrone autre que `async` et `await`, cet article se concentre sur les constructions au niveau du langage.

Exemple de code utilisant les E/S de manière intensive : télécharger des données d'un service web

Imaginons que vous souhaitez écrire du code qui télécharge des données d'un service web quand un utilisateur appuie sur un bouton, mais qui ne bloque pas le thread d'interface utilisateur. Cela peut être effectué comme suit, à l'aide de la classe System.Net.Http.HttpClient :

```
C#
```

```
private readonly HttpClient _httpClient = new HttpClient();

downloadButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI as the request
    // from the web service is happening.
    //
    // The UI thread is now free to perform other work.
    var stringData = await _httpClient.GetStringAsync(URL);
    DoSomethingWithData(stringData);
};
```

Le code exprime l'intention (télécharger des données de façon asynchrone) sans que cela nécessite des interactions compliquées avec les objets `Task`.

Exemple de code utilisant le processeur de manière intensive : effectuer un calcul dans un jeu

Supposons que vous développez un jeu pour mobile dans lequel l'appui sur un bouton peut causer des dommages à de nombreux ennemis à l'écran. Le calcul des dommages infligés peut nécessiter beaucoup de ressources. Si ce calcul est effectué sur le thread d'interface utilisateur, le jeu risque d'être considérablement ralenti pendant la durée du calcul.

La meilleure façon de gérer cette situation est de démarrer un thread d'arrière-plan qui effectue le travail à l'aide de `Task.Run`, et attend son résultat utilisant `await`. Ainsi, l'interface utilisateur conserve les mêmes performances pendant que le travail est effectué.

C#

```
private DamageResult CalculateDamageDone()
{
    // Code omitted:
    //
    // Does an expensive calculation and returns
    // the result of that calculation.
}

calculateButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI while CalculateDamageDone()
    // performs its work. The UI thread is free to perform other work.
    var damageResult = await Task.Run(() => CalculateDamageDone());
    DisplayDamage(damageResult);
};
```

Ce code exprime clairement l'intention de l'événement de clic du bouton. Il ne nécessite pas de gérer un thread d'arrière-plan manuellement et il effectue le travail de façon à ne rien bloquer.

Les dessous du code

Sur le plan du langage C#, le compilateur transforme votre code en une machine à états qui effectue le suivi des événements, tels que la suspension de l'exécution en présence d'un `await` et la reprise de l'exécution à la fin d'un travail en arrière-plan.

D'un point de vue théorique, il s'agit d'une implémentation du modèle de promesses d'asynchronisme.

Points clés à comprendre

- Le code asynchrone peut être utilisé pour du code utilisant les E/S ou le processeur de manière intensive, mais il est utilisé de manière différente dans chaque scénario.
- Le code asynchrone utilise les objets `Task<T>` et `Task`, qui sont des constructions servant à modéliser le travail effectué en arrière-plan.
- Le mot clé `async` définit une méthode comme asynchrone, ce qui vous permet d'utiliser le mot clé `await` dans le corps de la méthode.
- Quand le mot clé `await` est utilisé, il suspend la méthode d'appel et cède le contrôle à son appelant jusqu'à ce que la tâche awaited soit terminée.
- Le mot clé `await` peut uniquement être utilisé dans une méthode asynchrone.

Déterminer si un travail utilise le processeur ou les E/S de manière intensive

Les deux premiers exemples de ce guide ont montré comment utiliser `async` et `await` pour un travail utilisant les E/S et le processeur de manière intensive. Il est primordial de savoir déterminer si un travail à effectuer utilise les E/S ou le processeur de manière intensive, car ce point peut avoir un impact important sur les performances de votre code et entraîner une utilisation incorrecte de certaines constructions.

Voici deux questions à vous poser avant d'écrire du code :

1. Votre code doit-il « attendre » quelque chose, par exemple des données d'une base de données ?

   Si la réponse est « oui », le travail **utilise les E/S de manière intensive**.

2. Le code doit-il effectuer un calcul très complexe ?

   Si la réponse est « oui », le travail **utilise le processeur de manière intensive**.

Si le travail à faire **utilise les E/S de manière intensive**, utilisez `async` et `await` *sans* `Task.Run`. Vous *ne devez pas* utiliser la bibliothèque parallèle de tâches.

Si le travail à faire **utilise le processeur de manière intensive** et que la réactivité est une exigence, utilisez `async` et `await`, mais transférez le travail sur un autre thread *avec* `Task.Run`. Si le travail est approprié pour la concurrence et le parallélisme, envisagez également d'utiliser la bibliothèque parallèle de tâches.

De plus, vous devez toujours mesurer les performances d'exécution de votre code. Par exemple, vous constaterez peut-être que le coût d'un travail utilisant le processeur de manière intensive n'est pas si élevé que cela par rapport à la surcharge des changements de contexte induits par le multithreading. Chaque solution ayant ses compromis, choisissez le meilleur compromis pour votre scénario.

Autres exemples

Les exemples suivants montrent diverses façons d'écrire du code asynchrone dans C#. Ils correspondent à plusieurs scénarios différents que vous êtes susceptible de rencontrer.

Extraire des données d'un réseau

Cet extrait de code télécharge le code HTML à partir de la page d'accueil du site https://dotnetfoundation.org et compte le nombre d'occurrences de la chaîne « .NET » dans le code HTML. Il utilise ASP.NET pour définir une méthode de contrôleur d'API web qui exécute cette tâche et retourne le nombre.

> ⓘ **Notes**
>
> Si vous prévoyez d'effectuer une analyse HTML dans le code de production, n'utilisez pas d'expressions régulières. Utilisez plutôt une bibliothèque d'analyse.

C#

```
private readonly HttpClient _httpClient = new HttpClient();

[HttpGet, Route("DotNetCount")]
public async Task<int> GetDotNetCount()
{
    // Suspends GetDotNetCount() to allow the caller (the web server)
```

```
    // to accept another request, rather than blocking on this one.
    var html = await
_httpClient.GetStringAsync("https://dotnetfoundation.org");

    return Regex.Matches(html, @"\.NET").Count;
}
```

Voici le même scénario écrit pour une application Windows universelle, qui effectue la même tâche quand l'utilisateur appuie sur un bouton :

C#

```
private readonly HttpClient _httpClient = new HttpClient();

private async void OnSeeTheDotNetsButtonClick(object sender, RoutedEventArgs
e)
{
    // Capture the task handle here so we can await the background task
later.
    var getDotNetFoundationHtmlTask =
_httpClient.GetStringAsync("https://dotnetfoundation.org");

    // Any other work on the UI thread can be done here, such as enabling a
Progress Bar.
    // This is important to do here, before the "await" call, so that the
user
    // sees the progress bar before execution of this method is yielded.
    NetworkProgressBar.IsEnabled = true;
    NetworkProgressBar.Visibility = Visibility.Visible;

    // The await operator suspends OnSeeTheDotNetsButtonClick(), returning
control to its caller.
    // This is what allows the app to be responsive and not block the UI
thread.
    var html = await getDotNetFoundationHtmlTask;
    int count = Regex.Matches(html, @"\.NET").Count;

    DotNetCountLabel.Text = $"Number of .NETs on dotnetfoundation.org:
{count}";

    NetworkProgressBar.IsEnabled = false;
    NetworkProgressBar.Visibility = Visibility.Collapsed;
}
```

Attendre la fin de plusieurs tâches

Vous pouvez avoir un scénario qui nécessite de récupérer plusieurs éléments de données simultanément. L'API `Task` fournit deux méthodes, Task.WhenAll et

Task.WhenAny, que vous pouvez utiliser pour écrire du code asynchrone qui spécifie une attente non bloquante sur plusieurs travaux en arrière-plan.

Cet exemple vous montre comment récupérer des données `User` pour plusieurs `userId`.

C#

```
public async Task<User> GetUserAsync(int userId)
{
    // Code omitted:
    //
    // Given a user Id {userId}, retrieves a User object corresponding
    // to the entry in the database with {userId} as its Id.
}

public static async Task<IEnumerable<User>> GetUsersAsync(IEnumerable<int>
userIds)
{
    var getUserTasks = new List<Task<User>>();
    foreach (int userId in userIds)
    {
        getUserTasks.Add(GetUserAsync(userId));
    }

    return await Task.WhenAll(getUserTasks);
}
```

Voici une façon plus succincte d'écrire ce code, en utilisant LINQ :

C#

```
public async Task<User> GetUserAsync(int userId)
{
    // Code omitted:
    //
    // Given a user Id {userId}, retrieves a User object corresponding
    // to the entry in the database with {userId} as its Id.
}

public static async Task<User[]> GetUsersAsync(IEnumerable<int> userIds)
{
    var getUserTasks = userIds.Select(id => GetUserAsync(id)).ToArray();
    return await Task.WhenAll(getUserTasks);
}
```

Si vous choisissez de combiner LINQ avec du code asynchrone, pour réduire la quantité de code, faites-le avec précaution. Du fait que LINQ utilise l'exécution différée, les appels asynchrones ne sont pas effectués immédiatement comme c'est le cas dans une boucle `foreach`, sauf si vous forcez l'itération de la séquence générée avec un appel à

`.ToList()` ou `.ToArray()`. L'exemple ci-dessus utilise Enumerable.ToArray pour exécuter la requête avec impatience et stocker les résultats dans un tableau. Cela force le code `id => GetUserAsync(id)` à s'exécuter et à démarrer la tâche.

Informations et conseils importants

Avec la programmation asynchrone, gardez à l'esprit certains détails qui peuvent empêcher un comportement inattendu.

- **Les méthodes `async` doivent contenir un mot clé `await` dans leur corps pour pouvoir être suspendues.**

  Il ne faut pas oublier ce point. Si `await` n'est pas utilisé dans le corps d'une méthode `async`, le compilateur C# génère un avertissement, mais le code est compilé et exécuté comme s'il s'agissait d'une méthode standard. Cela est incroyablement inefficace, car la machine à états générée par le compilateur C# pour la méthode asynchrone n'accomplit rien.

- **Ajoutez « Async » comme suffixe de chaque nom de méthode asynchrone que vous écrivez.**

  Cette convention de .NET aide à différencier les méthodes synchrones et asynchrones. Cela n'est pas obligatoirement pour certaines méthodes qui ne sont pas explicitement appelées par votre code (par exemple, les gestionnaires d'événements et les méthodes de contrôleur web). L'attribution d'un nom explicite pour ces méthodes a moins d'importance.

- `async void` **doit être utilisé uniquement pour les gestionnaires d'événements.**

  L'utilisation de `async void` est le seul moyen de permettre le fonctionnement des gestionnaires d'événements asynchrones, car les événements n'ont pas de types de retour (et ne peuvent donc pas utiliser les objets `Task` et `Task<T>`). Toute autre utilisation de la méthode `async void` ne suit pas le modèle TAP et peut être difficile à implémenter, comme expliqué ci-après :
  - Les exceptions levées dans une méthode `async void` ne peuvent pas être interceptées en dehors de cette méthode.
  - Les méthodes `async void` sont très difficiles à tester.
  - Les méthodes `async void` peuvent avoir des effets secondaires gênants si l'appelant n'attend pas de méthodes asynchrones.

- **Définissez le thread avec précaution si vous utilisez des expressions lambda asynchrones dans des expressions LINQ.**

Dans LINQ, les expressions lambda utilisent l'exécution différée, ce qui signifie que l'exécution du code peut s'arrêter à un point que vous n'aviez pas prévu. L'introduction de tâches bloquantes dans ce code peut facilement provoquer un interblocage si le code n'est pas écrit correctement. De plus, l'imbrication de code asynchrone peut rendre la logique d'exécution du code plus compliquée. Combiner du code asynchrone et du code LINQ offre beaucoup de possibilités, mais nécessite d'être fait avec précaution et de manière claire.

- **Écrivez du code qui attend certaines tâches de façon non bloquante.**

  Si vous choisissez de bloquer le thread actuel pour attendre la fin d'une `Task`, vous risquez de provoquer des interblocages et le blocage de threads de contexte, et de gérer moins facilement les erreurs. Le tableau suivant fournit des conseils pour attendre la fin de tâches de façon non bloquante :

Élément à utiliser...	Au lieu de...	Pour...
`await`	`Task.Wait` ou `Task.Result`	Extraire le résultat d'une tâche en arrière-plan
`await Task.WhenAny`	`Task.WaitAny`	Attendre la fin d'une tâche
`await Task.WhenAll`	`Task.WaitAll`	Attendre la fin de toutes les tâches
`await Task.Delay`	`Thread.Sleep`	Attendre pendant une période

- **Envisagez d'utiliser `ValueTask` le cas échéant**

  Le retour d'un objet `Task` à partir de méthodes async peut introduire des goulots d'étranglement au niveau des performances dans certains chemins. `Task` est un type référence. Si vous l'utilisez, vous allouez donc un objet. Dans les cas où une méthode déclarée avec le modificateur `async` retourne un résultat mis en cache, ou si elle s'exécute de manière synchrone, le coût en termes de temps induit par les allocations supplémentaires peut s'avérer significatif dans les sections de code critiques pour les performances. Cela peut devenir coûteux si ces allocations se produisent dans des boucles serrées. Pour plus d'informations, consultez Types de retour asynchrones généralisés.

- **Envisagez d'utiliser `ConfigureAwait(false)`**

  Une question courante est la suivante : « Quand dois-je utiliser la méthode Task.ConfigureAwait(Boolean) ? ». La méthode permet à une instance `Task` de

configurer son awaiter. Il s'agit d'une considération importante et sa définition incorrecte peut potentiellement avoir des implications sur les performances et même des interblocages. Pour plus d'informations sur `ConfigureAwait`, consultez la FAQ ConfigureAwait.

- **Limitez l'écriture de code avec état.**

  Écrivez du code qui ne dépend pas de l'état d'objets globaux ou de l'exécution de certaines méthodes. Le code doit uniquement dépendre des valeurs de retour des méthodes. Pourquoi ?
  - La logique du code sera plus facile à comprendre.
  - Le code sera plus facile à tester.
  - Combiner du code asynchrone et du code synchrone est beaucoup plus simple.
  - Les concurrences critiques peuvent généralement être évitées.
  - Rendre le code dépendant des valeurs de retour facilite la coordination du code asynchrone.
  - En prime, le code fonctionne parfaitement avec l'injection de dépendances.

L'objectif recommandé est d'atteindre une transparence référentielle complète ou quasi-complète dans votre code. Votre base de code sera alors prévisible, testable et facile à gérer.

Autres ressources

- Modèle de programmation asynchrone de tâche (C#).

Asynchronous programming with async and await

Article • 02/13/2023

The Task asynchronous programming model (TAP) provides an abstraction over asynchronous code. You write code as a sequence of statements, just like always. You can read that code as though each statement completes before the next begins. The compiler performs many transformations because some of those statements may start work and return a Task that represents the ongoing work.

That's the goal of this syntax: enable code that reads like a sequence of statements, but executes in a much more complicated order based on external resource allocation and when tasks are complete. It's analogous to how people give instructions for processes that include asynchronous tasks. Throughout this article, you'll use an example of instructions for making breakfast to see how the `async` and `await` keywords make it easier to reason about code that includes a series of asynchronous instructions. You'd write the instructions something like the following list to explain how to make a breakfast:

1. Pour a cup of coffee.
2. Heat a pan, then fry two eggs.
3. Fry three slices of bacon.
4. Toast two pieces of bread.
5. Add butter and jam to the toast.
6. Pour a glass of orange juice.

If you have experience with cooking, you'd execute those instructions **asynchronously**. You'd start warming the pan for eggs, then start the bacon. You'd put the bread in the toaster, then start the eggs. At each step of the process, you'd start a task, then turn your attention to tasks that are ready for your attention.

Cooking breakfast is a good example of asynchronous work that isn't parallel. One person (or thread) can handle all these tasks. Continuing the breakfast analogy, one person can make breakfast asynchronously by starting the next task before the first task completes. The cooking progresses whether or not someone is watching it. As soon as you start warming the pan for the eggs, you can begin frying the bacon. Once the bacon starts, you can put the bread into the toaster.

For a parallel algorithm, you'd need multiple cooks (or threads). One would make the eggs, one the bacon, and so on. Each one would be focused on just that one task. Each

cook (or thread) would be blocked synchronously waiting for the bacon to be ready to flip, or the toast to pop.

Now, consider those same instructions written as C# statements:

C#

```
using System;
using System.Threading.Tasks;

namespace AsyncBreakfast
{
    // These classes are intentionally empty for the purpose of this
example. They are simply marker classes for the purpose of demonstration,
contain no properties, and serve no other purpose.
    internal class Bacon { }
    internal class Coffee { }
    internal class Egg { }
    internal class Juice { }
    internal class Toast { }

    class Program
    {
        static void Main(string[] args)
        {
            Coffee cup = PourCoffee();
            Console.WriteLine("coffee is ready");

            Egg eggs = FryEggs(2);
            Console.WriteLine("eggs are ready");

            Bacon bacon = FryBacon(3);
            Console.WriteLine("bacon is ready");

            Toast toast = ToastBread(2);
            ApplyButter(toast);
            ApplyJam(toast);
            Console.WriteLine("toast is ready");

            Juice oj = PourOJ();
            Console.WriteLine("oj is ready");
            Console.WriteLine("Breakfast is ready!");
        }

        private static Juice PourOJ()
        {
            Console.WriteLine("Pouring orange juice");
            return new Juice();
        }

        private static void ApplyJam(Toast toast) =>
            Console.WriteLine("Putting jam on the toast");
```

```
        private static void ApplyButter(Toast toast) =>
            Console.WriteLine("Putting butter on the toast");

        private static Toast ToastBread(int slices)
        {
            for (int slice = 0; slice < slices; slice++)
            {
                Console.WriteLine("Putting a slice of bread in the
toaster");
            }
            Console.WriteLine("Start toasting...");
            Task.Delay(3000).Wait();
            Console.WriteLine("Remove toast from toaster");

            return new Toast();
        }

        private static Bacon FryBacon(int slices)
        {
            Console.WriteLine($"putting {slices} slices of bacon in the
pan");
            Console.WriteLine("cooking first side of bacon...");
            Task.Delay(3000).Wait();
            for (int slice = 0; slice < slices; slice++)
            {
                Console.WriteLine("flipping a slice of bacon");
            }
            Console.WriteLine("cooking the second side of bacon...");
            Task.Delay(3000).Wait();
            Console.WriteLine("Put bacon on plate");

            return new Bacon();
        }

        private static Egg FryEggs(int howMany)
        {
            Console.WriteLine("Warming the egg pan...");
            Task.Delay(3000).Wait();
            Console.WriteLine($"cracking {howMany} eggs");
            Console.WriteLine("cooking the eggs ...");
            Task.Delay(3000).Wait();
            Console.WriteLine("Put eggs on plate");

            return new Egg();
        }

        private static Coffee PourCoffee()
        {
            Console.WriteLine("Pouring coffee");
            return new Coffee();
        }
    }
}
```

The synchronously prepared breakfast took roughly 30 minutes because the total is the sum of each task.

Computers don't interpret those instructions the same way people do. The computer will block on each statement until the work is complete before moving on to the next statement. That creates an unsatisfying breakfast. The later tasks wouldn't be started until the earlier tasks had been completed. It would take much longer to create the breakfast, and some items would have gotten cold before being served.

If you want the computer to execute the above instructions asynchronously, you must write asynchronous code.

These concerns are important for the programs you write today. When you write client programs, you want the UI to be responsive to user input. Your application shouldn't make a phone appear frozen while it's downloading data from the web. When you write server programs, you don't want threads blocked. Those threads could be serving other requests. Using synchronous code when asynchronous alternatives exist hurts your ability to scale out less expensively. You pay for those blocked threads.

Successful modern applications require asynchronous code. Without language support, writing asynchronous code required callbacks, completion events, or other means that obscured the original intent of the code. The advantage of the synchronous code is that

its step-by-step actions make it easy to scan and understand. Traditional asynchronous models forced you to focus on the asynchronous nature of the code, not on the fundamental actions of the code.

Don't block, await instead

The preceding code demonstrates a bad practice: constructing synchronous code to perform asynchronous operations. As written, this code blocks the thread executing it from doing any other work. It won't be interrupted while any of the tasks are in progress. It would be as though you stared at the toaster after putting the bread in. You'd ignore anyone talking to you until the toast popped.

Let's start by updating this code so that the thread doesn't block while tasks are running. The `await` keyword provides a non-blocking way to start a task, then continue execution when that task completes. A simple asynchronous version of the make a breakfast code would look like the following snippet:

C#

```
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");

    Egg eggs = await FryEggsAsync(2);
    Console.WriteLine("eggs are ready");

    Bacon bacon = await FryBaconAsync(3);
    Console.WriteLine("bacon is ready");

    Toast toast = await ToastBreadAsync(2);
    ApplyButter(toast);
    ApplyJam(toast);
    Console.WriteLine("toast is ready");

    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");
    Console.WriteLine("Breakfast is ready!");
}
```

> **ⓘ Important**
>
> The total elapsed time is roughly the same as the initial synchronous version. The code has yet to take advantage of some of the key features of asynchronous programming.

> **Tip**
>
> The method bodies of the `FryEggsAsync`, `FryBaconAsync`, and `ToastBreadAsync` have all been updated to return `Task<Egg>`, `Task<Bacon>`, and `Task<Toast>` respectively. The methods are renamed from their original version to include the "Async" suffix. Their implementations are shown as part of the **final version** later in this article.

> **Note**
>
> The `Main` method returns `Task`, despite not having a `return` expression—this is by design. For more information, see **Evaluation of a void-returning async function**.

This code doesn't block while the eggs or the bacon are cooking. This code won't start any other tasks though. You'd still put the toast in the toaster and stare at it until it pops. But at least, you'd respond to anyone that wanted your attention. In a restaurant where multiple orders are placed, the cook could start another breakfast while the first is cooking.

Now, the thread working on the breakfast isn't blocked while awaiting any started task that hasn't yet finished. For some applications, this change is all that's needed. A GUI application still responds to the user with just this change. However, for this scenario, you want more. You don't want each of the component tasks to be executed sequentially. It's better to start each of the component tasks before awaiting the previous task's completion.

Start tasks concurrently

In many scenarios, you want to start several independent tasks immediately. Then, as each task finishes, you can continue other work that's ready. In the breakfast analogy, that's how you get breakfast done more quickly. You also get everything done close to the same time. You'll get a hot breakfast.

The System.Threading.Tasks.Task and related types are classes you can use to reason about tasks that are in progress. That enables you to write code that more closely resembles the way you'd create breakfast. You'd start cooking the eggs, bacon, and toast at the same time. As each requires action, you'd turn your attention to that task, take care of the next action, then wait for something else that requires your attention.

You start a task and hold on to the Task object that represents the work. You'll `await` each task before working with its result.

Let's make these changes to the breakfast code. The first step is to store the tasks for operations when they start, rather than awaiting them:

C#

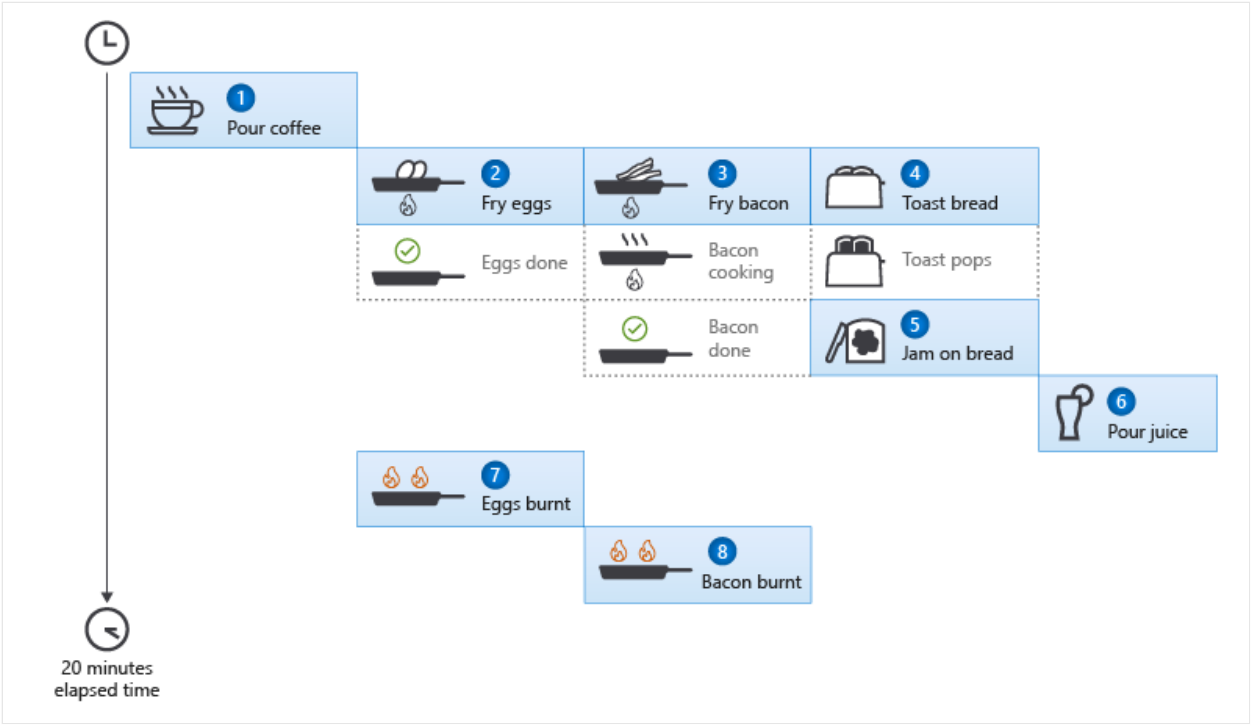
```
Coffee cup = PourCoffee();
Console.WriteLine("Coffee is ready");

Task<Egg> eggsTask = FryEggsAsync(2);
Egg eggs = await eggsTask;
Console.WriteLine("Eggs are ready");

Task<Bacon> baconTask = FryBaconAsync(3);
Bacon bacon = await baconTask;
Console.WriteLine("Bacon is ready");

Task<Toast> toastTask = ToastBreadAsync(2);
Toast toast = await toastTask;
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("Toast is ready");

Juice oj = PourOJ();
Console.WriteLine("Oj is ready");
Console.WriteLine("Breakfast is ready!");
```

The preceding code won't get your breakfast ready any faster. The tasks are all `await`ed as soon as they are started. Next, you can move the `await` statements for the bacon and eggs to the end of the method, before serving breakfast:

C#

```
Coffee cup = PourCoffee();
Console.WriteLine("Coffee is ready");

Task<Egg> eggsTask = FryEggsAsync(2);
Task<Bacon> baconTask = FryBaconAsync(3);
Task<Toast> toastTask = ToastBreadAsync(2);

Toast toast = await toastTask;
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("Toast is ready");
Juice oj = PourOJ();
Console.WriteLine("Oj is ready");

Egg eggs = await eggsTask;
Console.WriteLine("Eggs are ready");
Bacon bacon = await baconTask;
Console.WriteLine("Bacon is ready");
```

```
Console.WriteLine("Breakfast is ready!");
```



The asynchronously prepared breakfast took roughly 20 minutes, this time savings is because some tasks ran concurrently.

The preceding code works better. You start all the asynchronous tasks at once. You await each task only when you need the results. The preceding code may be similar to code in a web application that makes requests to different microservices, then combines the results into a single page. You'll make all the requests immediately, then `await` all those tasks and compose the web page.

Composition with tasks

You have everything ready for breakfast at the same time except the toast. Making the toast is the composition of an asynchronous operation (toasting the bread), and synchronous operations (adding the butter and the jam). Updating this code illustrates an important concept:

> **ⓘ Important**
>
> The composition of an asynchronous operation followed by synchronous work is an asynchronous operation. Stated another way, if any portion of an operation is asynchronous, the entire operation is asynchronous.

The preceding code showed you that you can use Task or Task<TResult> objects to hold running tasks. You `await` each task before using its result. The next step is to create methods that represent the combination of other work. Before serving breakfast, you want to await the task that represents toasting the bread before adding butter and jam. You can represent that work with the following code:

C#

```
static async Task<Toast> MakeToastWithButterAndJamAsync(int number)
{
    var toast = await ToastBreadAsync(number);
    ApplyButter(toast);
    ApplyJam(toast);

    return toast;
}
```

The preceding method has the `async` modifier in its signature. That signals to the compiler that this method contains an `await` statement; it contains asynchronous operations. This method represents the task that toasts the bread, then adds butter and jam. This method returns a Task<TResult> that represents the composition of those three operations. The main block of code now becomes:

C#

```
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");

    var eggsTask = FryEggsAsync(2);
    var baconTask = FryBaconAsync(3);
    var toastTask = MakeToastWithButterAndJamAsync(2);

    var eggs = await eggsTask;
    Console.WriteLine("eggs are ready");

    var bacon = await baconTask;
    Console.WriteLine("bacon is ready");

    var toast = await toastTask;
    Console.WriteLine("toast is ready");

    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");
    Console.WriteLine("Breakfast is ready!");
}
```

The previous change illustrated an important technique for working with asynchronous code. You compose tasks by separating the operations into a new method that returns a task. You can choose when to await that task. You can start other tasks concurrently.

Asynchronous exceptions

Up to this point, you've implicitly assumed that all these tasks complete successfully. Asynchronous methods throw exceptions, just like their synchronous counterparts. Asynchronous support for exceptions and error handling strives for the same goals as asynchronous support in general: You should write code that reads like a series of synchronous statements. Tasks throw exceptions when they can't complete successfully. The client code can catch those exceptions when a started task is `awaited`. For example, let's assume that the toaster catches fire while making the toast. You can simulate that by modifying the `ToastBreadAsync` method to match the following code:

C#

```
private static async Task<Toast> ToastBreadAsync(int slices)
{
    for (int slice = 0; slice < slices; slice++)
    {
        Console.WriteLine("Putting a slice of bread in the toaster");
    }
    Console.WriteLine("Start toasting...");
    await Task.Delay(2000);
    Console.WriteLine("Fire! Toast is ruined!");
    throw new InvalidOperationException("The toaster is on fire");
    await Task.Delay(1000);
    Console.WriteLine("Remove toast from toaster");

    return new Toast();
}
```

> **Note**
>
> You'll get a warning when you compile the preceding code regarding unreachable code. That's intentional, because once the toaster catches fire, operations won't proceed normally.

Run the application after making these changes, and you'll output similar to the following text:

Console

```
Pouring coffee
Coffee is ready
Warming the egg pan...
putting 3 slices of bacon in the pan
Cooking first side of bacon...
Putting a slice of bread in the toaster
Putting a slice of bread in the toaster
Start toasting...
Fire! Toast is ruined!
Flipping a slice of bacon
Flipping a slice of bacon
Flipping a slice of bacon
Cooking the second side of bacon...
Cracking 2 eggs
Cooking the eggs ...
Put bacon on plate
Put eggs on plate
Eggs are ready
Bacon is ready
Unhandled exception. System.InvalidOperationException: The toaster is on
fire
   at AsyncBreakfast.Program.ToastBreadAsync(Int32 slices) in
Program.cs:line 65
   at AsyncBreakfast.Program.MakeToastWithButterAndJamAsync(Int32 number) in
Program.cs:line 36
   at AsyncBreakfast.Program.Main(String[] args) in Program.cs:line 24
   at AsyncBreakfast.Program.<Main>(String[] args)
```

You'll notice quite a few tasks are completed between when the toaster catches fire and the exception is observed. When a task that runs asynchronously throws an exception, that Task is ***faulted***. The Task object holds the exception thrown in the Task.Exception property. Faulted tasks throw an exception when they're awaited.

There are two important mechanisms to understand: how an exception is stored in a faulted task, and how an exception is unpackaged and rethrown when code awaits a faulted task.

When code running asynchronously throws an exception, that exception is stored in the `Task`. The Task.Exception property is a System.AggregateException because more than one exception may be thrown during asynchronous work. Any exception thrown is added to the AggregateException.InnerExceptions collection. If that `Exception` property is null, a new `AggregateException` is created and the thrown exception is the first item in the collection.

The most common scenario for a faulted task is that the `Exception` property contains exactly one exception. When code `awaits` a faulted task, the first exception in the AggregateException.InnerExceptions collection is rethrown. That's why the output from

this example shows an `InvalidOperationException` instead of an `AggregateException`. Extracting the first inner exception makes working with asynchronous methods as similar as possible to working with their synchronous counterparts. You can examine the `Exception` property in your code when your scenario may generate multiple exceptions.

Before going on, comment out these two lines in your `ToastBreadAsync` method. You don't want to start another fire:

C#

```
Console.WriteLine("Fire! Toast is ruined!");
throw new InvalidOperationException("The toaster is on fire");
```

Await tasks efficiently

The series of `await` statements at the end of the preceding code can be improved by using methods of the `Task` class. One of those APIs is WhenAll, which returns a Task that completes when all the tasks in its argument list have completed, as shown in the following code:

C#

```
await Task.WhenAll(eggsTask, baconTask, toastTask);
Console.WriteLine("Eggs are ready");
Console.WriteLine("Bacon is ready");
Console.WriteLine("Toast is ready");
Console.WriteLine("Breakfast is ready!");
```

Another option is to use WhenAny, which returns a `Task<Task>` that completes when any of its arguments complete. You can await the returned task, knowing that it has already finished. The following code shows how you could use WhenAny to await the first task to finish and then process its result. After processing the result from the completed task, you remove that completed task from the list of tasks passed to `WhenAny`.

C#

```
var breakfastTasks = new List<Task> { eggsTask, baconTask, toastTask };
while (breakfastTasks.Count > 0)
{
    Task finishedTask = await Task.WhenAny(breakfastTasks);
    if (finishedTask == eggsTask)
    {
        Console.WriteLine("Eggs are ready");
```

```
    }
    else if (finishedTask == baconTask)
    {
        Console.WriteLine("Bacon is ready");
    }
    else if (finishedTask == toastTask)
    {
        Console.WriteLine("Toast is ready");
    }
    await finishedTask;
    breakfastTasks.Remove(finishedTask);
}
```

Near the end, you see the line `await finishedTask;`. The line `await Task.WhenAny` doesn't await the finished task. It `await`s the `Task` returned by `Task.WhenAny`. The result of `Task.WhenAny` is the task that has completed (or faulted). You should `await` that task again, even though you know it's finished running. That's how you retrieve its result, or ensure that the exception causing it to fault gets thrown.

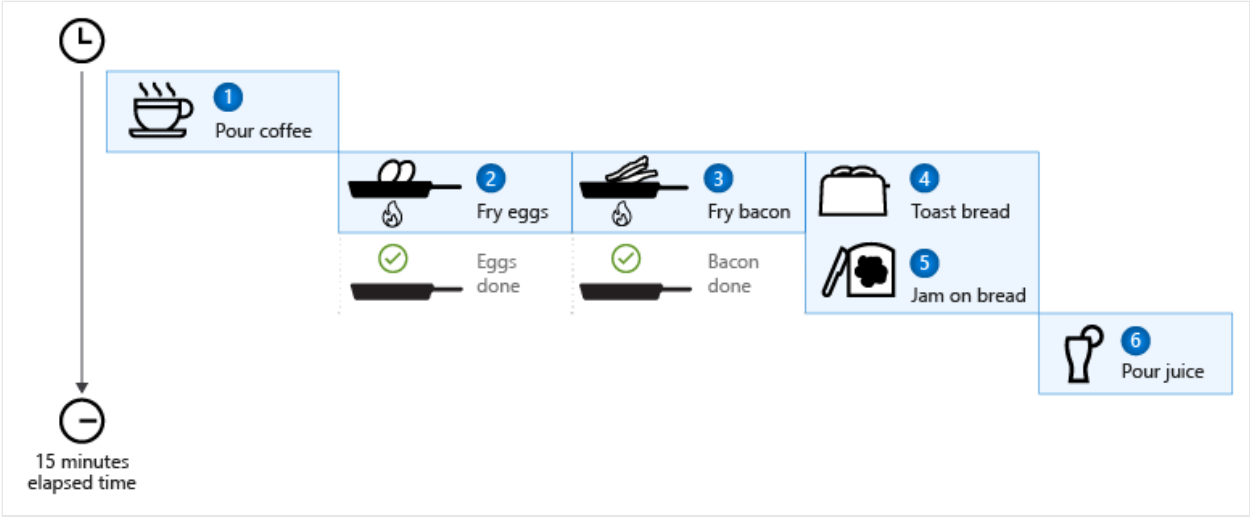
After all those changes, the final version of the code looks like this:

C#

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace AsyncBreakfast
{
    // These classes are intentionally empty for the purpose of this
example. They are simply marker classes for the purpose of demonstration,
contain no properties, and serve no other purpose.
    internal class Bacon { }
    internal class Coffee { }
    internal class Egg { }
    internal class Juice { }
    internal class Toast { }

    class Program
    {
```

```
        static async Task Main(string[] args)
        {
            Coffee cup = PourCoffee();
            Console.WriteLine("coffee is ready");

            var eggsTask = FryEggsAsync(2);
            var baconTask = FryBaconAsync(3);
            var toastTask = MakeToastWithButterAndJamAsync(2);

            var breakfastTasks = new List<Task> { eggsTask, baconTask,
toastTask };
            while (breakfastTasks.Count > 0)
            {
                Task finishedTask = await Task.WhenAny(breakfastTasks);
                if (finishedTask == eggsTask)
                {
                    Console.WriteLine("eggs are ready");
                }
                else if (finishedTask == baconTask)
                {
                    Console.WriteLine("bacon is ready");
                }
                else if (finishedTask == toastTask)
                {
                    Console.WriteLine("toast is ready");
                }
                await finishedTask;
                breakfastTasks.Remove(finishedTask);
            }

            Juice oj = PourOJ();
            Console.WriteLine("oj is ready");
            Console.WriteLine("Breakfast is ready!");
        }

        static async Task<Toast> MakeToastWithButterAndJamAsync(int number)
        {
            var toast = await ToastBreadAsync(number);
            ApplyButter(toast);
            ApplyJam(toast);

            return toast;
        }

        private static Juice PourOJ()
        {
            Console.WriteLine("Pouring orange juice");
            return new Juice();
        }

        private static void ApplyJam(Toast toast) =>
            Console.WriteLine("Putting jam on the toast");

        private static void ApplyButter(Toast toast) =>
            Console.WriteLine("Putting butter on the toast");
```

```
        private static async Task<Toast> ToastBreadAsync(int slices)
        {
            for (int slice = 0; slice < slices; slice++)
            {
                Console.WriteLine("Putting a slice of bread in the
toaster");
            }
            Console.WriteLine("Start toasting...");
            await Task.Delay(3000);
            Console.WriteLine("Remove toast from toaster");

            return new Toast();
        }

        private static async Task<Bacon> FryBaconAsync(int slices)
        {
            Console.WriteLine($"putting {slices} slices of bacon in the
pan");
            Console.WriteLine("cooking first side of bacon...");
            await Task.Delay(3000);
            for (int slice = 0; slice < slices; slice++)
            {
                Console.WriteLine("flipping a slice of bacon");
            }
            Console.WriteLine("cooking the second side of bacon...");
            await Task.Delay(3000);
            Console.WriteLine("Put bacon on plate");

            return new Bacon();
        }

        private static async Task<Egg> FryEggsAsync(int howMany)
        {
            Console.WriteLine("Warming the egg pan...");
            await Task.Delay(3000);
            Console.WriteLine($"cracking {howMany} eggs");
            Console.WriteLine("cooking the eggs ...");
            await Task.Delay(3000);
            Console.WriteLine("Put eggs on plate");

            return new Egg();
        }

        private static Coffee PourCoffee()
        {
            Console.WriteLine("Pouring coffee");
            return new Coffee();
        }
    }
}
```



The final version of the asynchronously prepared breakfast took roughly 6 minutes because some tasks ran concurrently, and the code monitored multiple tasks at once and only took action when it was needed.

This final code is asynchronous. It more accurately reflects how a person would cook a breakfast. Compare the preceding code with the first code sample in this article. The core actions are still clear from reading the code. You can read this code the same way you'd read those instructions for making a breakfast at the beginning of this article. The language features for `async` and `await` provide the translation every person makes to follow those written instructions: start tasks as you can and don't block waiting for tasks to complete.

Next steps

Explore real world scenarios for asynchronous programs

Modèle de programmation asynchrone des tâches

Article • 28/03/2023

Vous pouvez éviter des goulots d'étranglement de performance et améliorer la réactivité globale de votre application à l'aide de la programmation asynchrone. Toutefois, les techniques traditionnelles pour écrire des applications asynchrones peuvent être complexes et rendre ces applications difficiles à écrire, déboguer et mettre à jour.

C# introduit une approche simplifiée, la programmation asynchrone, qui tire parti de la prise en charge asynchrone dans le runtime .NET. Le compilateur effectue le travail difficile dont se chargeait le développeur jusqu'à maintenant. En outre, votre application conserve une structure logique qui ressemble à du code synchrone. Par conséquent, vous obtenez tous les avantages de la programmation asynchrone avec peu d'effort.

Cette rubrique fournit une vue d'ensemble sur quand et comment utiliser la programmation asynchrone, et inclut des liens vers des rubriques du support, qui contiennent des informations et des exemples.

Le comportement asynchrone améliore la réactivité.

Le comportement asynchrone est essentiel pour les activités qui sont potentiellement bloquantes, par exemple l'accès au web. L'accès à une ressource Web est parfois lent ou différé. Si cette activité est bloquée dans un processus synchrone, toute l'application doit attendre. Dans un processus asynchrone, l'application peut poursuivre une autre tâche qui ne dépend pas de la ressource Web jusqu'à ce que la tâche potentiellement bloquante soit terminée.

Le tableau suivant indique les zones classiques où la programmation asynchrone améliore la réactivité. Les API répertoriées de .NET et de Windows Runtime contiennent des méthodes qui prennent en charge la programmation asynchrone.

Domaine d'application	Types .NET avec les méthodes async	Types Windows Runtime avec les méthodes async
Accès Web	HttpClient	Windows.Web.Http.HttpClient SyndicationClient
Utilisation de fichiers	JsonSerializer StreamReader	StorageFile

Domaine d'application	Types .NET avec les méthodes async	Types Windows Runtime avec les méthodes async
	StreamWriter XmlReader XmlWriter	
Utilisation des images		MediaCapture BitmapEncoder BitmapDecoder
Programmation WCF	Opérations synchrones et asynchrones	

Le comportement asynchrone est particulièrement utile pour les applications qui accèdent au thread d'interface utilisateur, car toute activité liée à l'interface utilisateur partage généralement un thread. Si un processus est bloqué dans une application synchrone, tous les processus sont bloqués. Votre application ne répond plus et vous pouvez conclure qu'elle a rencontré une défaillance, alors qu'elle attend simplement.

Lorsque vous utilisez des méthodes asynchrones, l'application continue à répondre à l'interface utilisateur. Vous pouvez redimensionner ou réduire une fenêtre, par exemple, ou vous pouvez fermer l'application si vous ne souhaitez pas attendre qu'elle se termine.

L'approche basée sur async ajoute l'équivalent d'une transmission automatique à la liste d'options dont vous disposez pour concevoir des opérations asynchrones. En d'autres termes, vous obtenez tous les avantages de la programmation asynchrone classique mais avec beaucoup moins d'efforts du point de vue du développeur.

Les méthodes asynchrones sont faciles à écrire

Les mots clés async et await en C# sont au cœur de la programmation async. Avec ces deux mots clés, vous pouvez utiliser des ressources dans .NET Framework, .NET Core ou Windows Runtime pour créer une méthode asynchrone presque aussi facilement qu'une méthode synchrone. Les méthodes asynchrones définies avec mot clé `async` sont appelées *méthodes asynchrones*.

L'exemple suivant illustre une méthode async. Presque tous les éléments du code doivent vous sembler familiers.

Vous trouverez un exemple complet de Windows Presentation Foundation (WPF) disponible en téléchargement à partir de Programmation asynchrone avec async et await en C#.

```
C#
```

```
public async Task<int> GetUrlContentLengthAsync()
{
    var client = new HttpClient();

    Task<string> getStringTask =
        client.GetStringAsync("https://learn.microsoft.com/dotnet");

    DoIndependentWork();

    string contents = await getStringTask;

    return contents.Length;
}

void DoIndependentWork()
{
    Console.WriteLine("Working...");
}
```

Vous pouvez tirer plusieurs enseignements à partir de l'exemple précédent. Démarrez par la signature de la méthode. Elle inclut le modificateur `async`. Le type de retour est `Task<int>` (consultez la section « Types de retour » pour découvrir d'autres options). Le nom de la méthode se termine dans `Async`. Dans le corps de la méthode, `GetStringAsync` retourne un `Task<string>`. Cela signifie que quand vous attendez (`await`) la tâche, vous obtenez un `string` (`contents`). Avant d'attendre la tâche, vous pouvez effectuer tout travail qui ne repose pas sur le `string` de `GetStringAsync`.

Faites particulièrement attention à l'opérateur `await`. Il suspend `GetUrlContentLengthAsync` :

- `GetUrlContentLengthAsync` ne peut pas continuer tant que `getStringTask` n'est pas terminé.
- Pendant ce temps, le contrôle retourne à l'appelant de `GetUrlContentLengthAsync`.
- Le contrôle reprend ici quand `getStringTask` est terminé.
- L'opérateur `await` récupère alors le résultat `string` auprès de `getStringTask`.

L'instruction return spécifie un résultat entier. Toutes les méthodes qui attendent `GetUrlContentLengthAsync` récupèrent la valeur de longueur.

Si `GetUrlContentLengthAsync` n'a aucun travail qu'il peut effectuer entre l'appel de `GetStringAsync` et l'attente de son achèvement, vous pouvez simplifier votre code en appelant et attendant dans l'instruction unique suivante.

```
C#
```

```
string contents = await
client.GetStringAsync("https://learn.microsoft.com/dotnet");
```

Les caractéristiques suivantes résument ce qui fait de l'exemple précédent une méthode asynchrone :

- La signature de la méthode inclut un modificateur `async`.
- Le nom d'une méthode async, par convention, se termine par un suffixe « Async ».
- Le type de retour est l'un des types suivants :
  - Task<TResult> si votre méthode a une instruction return dans laquelle l'opérande est de type `TResult`.
  - Task si votre méthode n'a aucune instruction de retour, ou si elle a une instruction de retour sans opérande.
  - `void` si vous écrivez un gestionnaire d'événements async.
  - Tous les autres types ayant une méthode `GetAwaiter`.

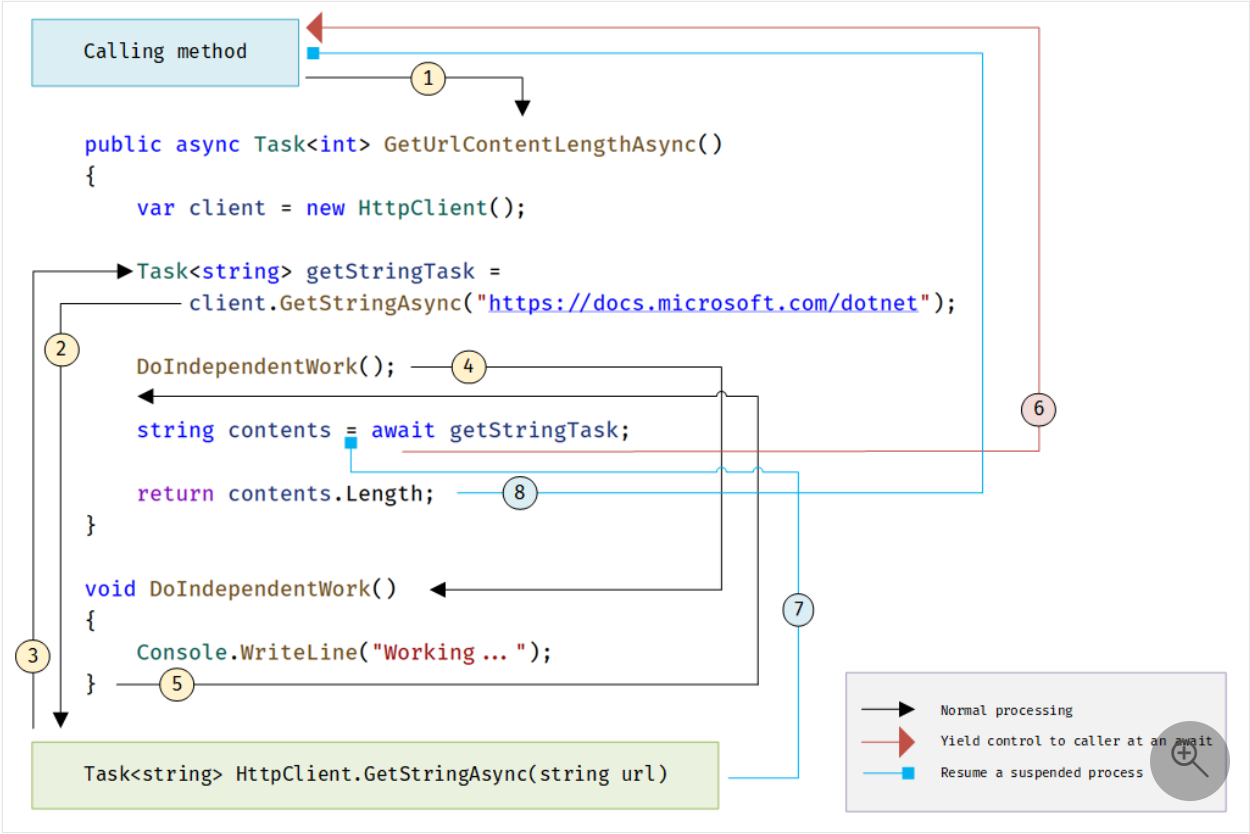
  Pour plus d'informations, consultez la section Types et paramètres de retour.
- La méthode inclut généralement au moins une expression `await`, qui marque le point au-delà duquel la méthode ne peut pas poursuivre son exécution tant que l'opération asynchrone attendue n'est pas terminée. Dans le même temps, la méthode est interrompue, et le contrôle retourne à l'appelant de la méthode. La section suivante de cette rubrique illustre ce qui se produit au point d'interruption.

Dans les méthodes async, vous utilisez les mots clés et les types fournis pour indiquer ce que vous souhaitez faire, et le compilateur effectue le reste, notamment le suivi de ce qui doit se produire lorsque le contrôle retourne à un point d'attente dans une méthode interrompue. Il peut être difficile de gérer des processus de routine, tels que les boucles et la gestion des exceptions, dans le code asynchrone traditionnel. Dans une méthode async, vous écrivez ces éléments comme vous l'auriez fait dans une solution synchrone, et le problème est résolu.

Pour plus d'informations sur le comportement asynchrone dans les versions antérieures de .NET Framework, consultezProgrammation asynchrone .NET Framework traditionnelle et TPL.

Ce qui se produit dans une méthode asynchrone

La chose la plus importante à comprendre en programmation asynchrone est le déplacement du flux de contrôle d'une méthode à l'autre. Le diagramme suivant présente le processus :



Les numéros du diagramme correspondent aux étapes suivantes, initiées lorsqu'une méthode appelante appelle la méthode asynchrone.

1. Une méthode appelante appelle et attend la méthode asynchrone `GetUrlContentLengthAsync`.

2. `GetUrlContentLengthAsync` crée une instance HttpClient et appelle la méthode asynchrone GetStringAsync pour télécharger le contenu d'un site Web comme une chaîne.

3. Quelque chose se produit dans `GetStringAsync` et suspend sa progression. Elle peut être obligée d'attendre la fin d'un téléchargement sur un site Web ou de toute autre activité bloquante. Pour éviter de bloquer les ressources, `GetStringAsync` cède le contrôle à son appelant, `GetUrlContentLengthAsync`.

   `GetStringAsync` retourne un Task<TResult>, où `TResult` est une chaîne, et `GetUrlContentLengthAsync` assigne la tâche à la variable `getStringTask`. La tâche représente le processus en cours de l'appel de `GetStringAsync`, avec l'engagement de produire une valeur de chaîne réelle lorsque le travail est terminé.

4. Étant donné que `getStringTask` n'a pas encore été attendu, `GetUrlContentLengthAsync` peut continuer avec un autre travail qui ne dépend pas du résultat final issu de `GetStringAsync`. Cette opération est représentée par un appel à la méthode synchrone `DoIndependentWork`.

5. `DoIndependentWork` est une méthode synchrone qui effectue son travail et retourne à son appelant.

6. `GetUrlContentLengthAsync` n'a plus de travail à exécuter sans un résultat de `getStringTask`. `GetUrlContentLengthAsync` veut ensuite calculer et retourner la longueur de la chaîne téléchargée, mais la méthode ne peut pas calculer cette valeur avant d'avoir la chaîne.

   Par conséquent, `GetUrlContentLengthAsync` utilise un opérateur await pour interrompre sa progression et pour céder le contrôle à la méthode ayant appelé `GetUrlContentLengthAsync`. `GetUrlContentLengthAsync` retourne un `Task<int>` à l'appelant. La tâche représente la promesse de produire un résultat entier qui est la longueur de la chaîne téléchargée.

   > ⓘ **Notes**
   >
   > Si `GetStringAsync` (et donc `getStringTask`) se termine avant que `GetUrlContentLengthAsync` ne l'attende, le contrôle reste dans `GetUrlContentLengthAsync`. Le fait de suspendre, puis de retourner à `GetUrlContentLengthAsync` ne sert à rien si le processus asynchrone appelé `getStringTask` s'est déjà effectué et si `GetUrlContentLengthAsync` n'a pas à attendre le résultat final.

   À l'intérieur de la méthode appelante, le modèle de traitement continue. L'appelant peut effectuer d'autres tâches qui ne dépendent pas du résultat de `GetUrlContentLengthAsync` avant d'attendre ce résultat, ou l'appelant peut attendre immédiatement. La méthode appelante attend `GetUrlContentLengthAsync`, et `GetUrlContentLengthAsync` attend `GetStringAsync`.

7. `GetStringAsync` se termine et génère un résultat de chaîne. Le résultat de chaîne n'est pas retourné par l'appel de `GetStringAsync` de la façon à laquelle vous pourriez vous attendre. (N'oubliez pas que la méthode a déjà retourné une tâche à l'étape 3.) Au lieu de cela, le résultat chaîne est stocké dans la tâche qui représente l'achèvement de la méthode, `getStringTask`. L'opérateur await récupère le résultat

de `getStringTask`. L'instruction d'assignation assigne le résultat récupéré à `contents`.

8. Lorsque `GetUrlContentLengthAsync` a le résultat de chaîne, la méthode peut calculer la longueur de la chaîne. Puis, le travail d'`GetUrlContentLengthAsync` est également interrompu, et le gestionnaire d'événements en attente peut reprendre. Dans l'exemple complet à la fin de la rubrique, vous pouvez vérifier que le gestionnaire d'événements récupère et imprime la valeur du résultat de la longueur. Si vous débutez en programmation asynchrone, prenez une minute pour déterminer la différence entre le comportement synchrone et le comportement asynchrone. Une méthode synchrone retourne une fois son travail terminé (étape 5), mais une méthode asynchrone retourne une valeur de tâche lorsque son travail est interrompu (étapes 3 et 6). Lorsque la méthode async termine finalement son travail, la tâche est marquée comme terminée et le résultat, le cas échéant, est stocké dans la tâche.

Méthodes asynchrones d'API

Vous pouvez vous demander où rechercher les méthodes telles que `GetStringAsync` qui prennent en charge la programmation async. .NET Framework 4.5 ou version ultérieure et .NET Core contiennent de nombreux membres qui fonctionnent avec `async` et `await`. Vous pouvez les identifier par le suffixe « Async » qui est ajouté au nom de membre, et par leur type de retour Task ou Task<TResult>. Par exemple, la classe `System.IO.Stream` contient des méthodes telles que CopyToAsync, ReadAsync et WriteAsync en même temps que les méthodes synchrones CopyTo, Read et Write.

Windows Runtime contient également de nombreuses méthodes utilisables avec `async` et `await` dans les applications Windows. Pour plus d'informations, consultez Programmation thread et asynchrone pour le développement UWP, et Programmation asynchrone (applications Windows Store) et Démarrage rapide : appel d'API asynchrones en C# ou Visual Basic si vous utilisez des versions antérieures de Windows Runtime.

Threads

Les méthodes Async sont conçues pour être des opérations non bloquantes. Une expression `await` dans une méthode asynchrone ne bloque pas le thread actuel lors de l'exécution de la tâche attendue. Au lieu de cela, l'expression inscrit le reste de la méthode comme continuation et retourne le contrôle à l'appelant de la méthode async.

Les mots clés `async` et `await` n'entraînent pas la création de threads supplémentaires. Les méthodes Async ne requièrent pas de multithreading, car une méthode async ne fonctionne pas sur son propre thread. La méthode s'exécute sur le contexte de synchronisation actuel et utilise du temps sur le thread uniquement lorsqu'elle est active. Vous pouvez utiliser Task.Run pour déplacer le travail lié au processeur vers un thread d'arrière-plan, mais un thread d'arrière-plan ne permet pas à un processus qui attend simplement les résultats de devenir disponible.

L'approche basée sur async en matière de programmation asynchrone est préférable aux approches existantes, dans presque tous les cas. En particulier, cette approche est préférable à la classe BackgroundWorker pour les opérations utilisant de nombreuses E/S, car le code est plus simple et il est inutile de vous protéger contre des conditions de concurrence. Combinée à la méthode Task.Run, la programmation asynchrone est plus appropriée que BackgroundWorker pour les opérations utilisant le processeur de manière intensive, car la programmation asynchrone sépare les détails de coordination de l'exécution de votre code à partir du travail que `Task.Run` transfère au pool de threads.

async et await

Si vous spécifiez qu'une méthode est une méthode async en utilisant le modificateur async, vous activez les deux fonctionnalités suivantes.

- La méthode async marquée peut utiliser await pour indiquer des points d'interruption. L'opérateur `await` indique au compilateur que la méthode asynchrone ne peut pas continuer au-delà de ce point, tant que le processus asynchrone attendu n'est pas terminé. Entre-temps, le contrôle retourne à l'appelant de la méthode async.

  L'interruption d'une méthode asynchrone dans une expression `await` ne constitue pas une sortie de la méthode et les blocs `finally` ne s'exécutent pas.

- La méthode async marquée peut elle-même être attendue par les méthodes qui l'appellent.

La méthode asynchrone contient généralement une ou plusieurs occurrences d'un opérateur `await`, mais l'absence des expressions `await` ne provoque pas d'erreur du compilateur. Si une méthode asynchrone n'utilise pas un opérateur `await` pour marquer un point de sélection, elle s'exécute comme le fait une méthode synchrone, en dépit du modificateur `async`. Le compilateur émet un avertissement pour ces méthodes.

`async` et `await` sont des mots clés contextuels. Pour plus d'informations et pour obtenir des exemples, consultez les rubriques suivantes :

- async
- await

Paramètres et types de retour

Une méthode async retourne généralement un Task ou un Task<TResult>. Dans une méthode async, un opérateur `await` est appliqué à une tâche retournée à partir d'un appel à une autre méthode async.

Vous spécifiez Task<TResult> comme type de retour si la méthode contient une instruction return qui spécifie un opérande de type `TResult`.

Vous utilisez Task comme type de retour si la méthode n'a aucune instruction return ou une instruction return qui ne retourne pas d'opérande.

Vous pouvez également spécifier n'importe quel autre type de retour, sous réserve que ce type inclue une méthode `GetAwaiter`. ValueTask<TResult> est un exemple d'un tel type. Il est disponible dans le package NuGet System.Threading.Tasks.Extension.

L'exemple suivant montre comment déclarer et appeler une méthode qui retourne Task<TResult> ou Task :

C#

```
async Task<int> GetTaskOfTResultAsync()
{
    int hours = 0;
    await Task.Delay(0);

    return hours;
}


Task<int> returnedTaskTResult = GetTaskOfTResultAsync();
int intResult = await returnedTaskTResult;
// Single line
// int intResult = await GetTaskOfTResultAsync();

async Task GetTaskAsync()
{
    await Task.Delay(0);
    // No return statement needed
}

Task returnedTask = GetTaskAsync();
```

```
await returnedTask;
// Single line
await GetTaskAsync();
```

Chaque tâche retournée représente le travail en cours. Une tâche encapsule des informations sur l'état du processus asynchrone et, éventuellement, le résultat final du processus ou l'exception que le processus déclenche s'il ne réussit pas.

Une méthode async peut également avoir un type de retour `void`. Ce type de retour est essentiellement utilisé pour définir les gestionnaires d'événements, où un type de retour `void` est obligatoire. Les gestionnaires d'événements asynchrones servent souvent de point de départ aux programmes asynchrones.

Une méthode asynchrone dont le type de retour est `void` ne peut pas être attendue, et l'appelant d'une méthode retournant void ne peut capturer aucune exception levée par la méthode.

Une méthode async ne peut pas déclarer de paramètres in, ref ou out, mais elle peut appeler des méthodes qui ont ces paramètres. De même, une méthode async ne peut pas retourner une valeur par référence, bien qu'elle puisse appeler des méthodes avec des valeurs de retour de référence.

Pour plus d'informations ainsi que des exemples, consultez la page Types de retour asynchrone (C#).

Les API asynchrones dans la programmation Windows Runtime ont l'un des types de retour suivants, qui sont semblables aux tâches :

- IAsyncOperation<TResult>, qui correspond à Task<TResult>
- IAsyncAction, qui correspond à Task
- IAsyncActionWithProgress<TProgress>
- IAsyncOperationWithProgress<TResult,TProgress>

Conventions d'affectation de noms

Par convention, les méthodes qui retournent généralement des types pouvant être attendus (par exemple, `Task`, `Task<T>`, `ValueTask`, `ValueTask<T>`) doivent avoir des noms qui se terminent par « Async ». Les méthodes qui démarrent une opération asynchrone, mais qui ne retournent pas un type awaitable ne doivent pas avoir un nom qui se termine par « Async ». Cependant, leur nom peut commencer par « Begin », « Start » ou un autre mot suggérant que cette méthode ne retourne pas ou ne lève pas le résultat de l'opération.

Vous pouvez ignorer la convention où un événement, une classe de base, ou un contrat d'interface suggère un nom différent. Par exemple, vous ne devez pas renommer les gestionnaires d'événements communs, tels que `OnButtonClick`.

Articles connexes (Visual Studio)

Titre	Description
Guide pratique pour effectuer plusieurs requêtes web en parallèle en utilisant async et await (C#)	Explique comment démarrer plusieurs tâches en même temps.
Types de retour async (C#)	Décrit les types que les méthodes async peuvent retourner et explique quand chaque type est approprié.
Annuler des tâches avec un jeton d'annulation comme mécanisme de signalisation.	Indique comment ajouter les fonctionnalités suivantes à votre solution async : - Annuler une liste de tâches (C#) - Annuler les tâches après un laps de temps (C#) - Traiter les tâches asynchrones terminées (C#)
Utiliser async pour l'accès aux fichiers (C#)	Répertorie et explique les avantages de l'utilisation d'async et d'await pour accéder aux fichiers.
Modèle asynchrone basé sur les tâches (TAP)	Décrit un modèle asynchrone, le modèle est basé sur les types Task et Task<TResult>.
Vidéos Async sur Channel 9	Fournit des liens vers diverses vidéos sur la programmation asynchrone.

Voir aussi

- Programmation asynchrone avec async et await
- async
- await

Types de retour asynchrone (C#)

Article • 10/05/2023

Les méthodes async peuvent avoir les types de retour suivants :

- Task, pour une méthode async qui effectue une opération mais ne retourne aucune valeur.
- Task<TResult>, pour une méthode async qui retourne une valeur.
- `void`, pour un gestionnaire d'événements.
- Tout type ayant une méthode `GetAwaiter` accessible. L'objet retourné par la méthode `GetAwaiter` doit implémenter l'interface System.Runtime.CompilerServices.ICriticalNotifyCompletion.
- IAsyncEnumerable<T>, pour une méthode asynchrone qui retourne un *flux asynchrone*.

Pour plus d'informations sur les méthodes async, consultez Programmation asynchrone avec async et await (C#).

Il existe plusieurs autres types qui sont spécifiques aux charges de travail Windows :

- DispatcherOperation, pour les opérations asynchrones limitées à Windows.
- IAsyncAction, pour les actions asynchrones dans UWP qui ne retournent pas de valeur.
- IAsyncActionWithProgress<TProgress>, pour les actions asynchrones dans UWP qui signalent la progression, mais ne retournent pas de valeur.
- IAsyncOperation<TResult>, pour les opérations asynchrones dans UWP qui retournent une valeur.
- IAsyncOperationWithProgress<TResult,TProgress>, pour les opérations asynchrones dans UWP qui signalent la progression et retournent une valeur.

Type de retour Task

Les méthodes async qui ne contiennent pas d'instruction `return` ou qui contiennent une instruction `return` ne retournant pas d'opérande ont généralement un type de retour Task. De telles méthodes retournent `void` si elles s'exécutent de façon synchrone. Si vous utilisez un type de retour Task pour une méthode async, une méthode d'appel peut utiliser un opérateur `await` pour suspendre l'achèvement de l'appelant jusqu'à ce que la méthode async soit terminée.

Dans l'exemple suivant, la méthode `WaitAndApologizeAsync` ne contient pas d'instruction `return`, de sorte que la méthode retourne un objet Task. Le retour de `Task` permet à `WaitAndApologizeAsync` d'être attendu. Le type Task n'inclut pas de propriété `Result`, car il n'a aucune valeur de retour.

C#

```
public static async Task DisplayCurrentInfoAsync()
{
    await WaitAndApologizeAsync();

    Console.WriteLine($"Today is {DateTime.Now:D}");
    Console.WriteLine($"The current time is {DateTime.Now.TimeOfDay:t}");
    Console.WriteLine("The current temperature is 76 degrees.");
}

static async Task WaitAndApologizeAsync()
{
    await Task.Delay(2000);

    Console.WriteLine("Sorry for the delay...\n");
}
// Example output:
//    Sorry for the delay...
//
// Today is Monday, August 17, 2020
// The current time is 12:59:24.2183304
// The current temperature is 76 degrees.
```

`WaitAndApologizeAsync` est attendue avec une instruction await au lieu d'une expression await, comme pour l'instruction d'appel d'une méthode synchrone retournant un type void. Dans ce cas, l'application d'un opérateur await ne génère pas de valeur. Lorsque l'opérande droit de `await` est Task<TResult>, l'expression `await` produit un résultat de `T`. Lorsque l'opérande droit de `await` est Task, le `await` et son opérande sont une instruction.

Vous pouvez séparer l'appel vers `WaitAndApologizeAsync` de l'application d'un opérateur await, comme le montre le code suivant. Cependant, n'oubliez pas qu'un type `Task` n'a pas de propriété `Result` et qu'aucune valeur n'est générée quand un opérateur await est appliqué à un type `Task`.

Le code suivant sépare l'appel à la méthode `WaitAndApologizeAsync` de l'attente de la tâche que retourne la méthode.

C#

```
Task waitAndApologizeTask = WaitAndApologizeAsync();

string output =
    $"Today is {DateTime.Now:D}\n" +
    $"The current time is {DateTime.Now.TimeOfDay:t}\n" +
    "The current temperature is 76 degrees.\n";

await waitAndApologizeTask;
Console.WriteLine(output);
```

Type de retour Task<TResult>

Le type de retour Task<TResult> est utilisé pour une méthode asynchrone qui contient une instruction return dans laquelle l'opérande est `TResult`.

Dans l'exemple suivant, la méthode `GetLeisureHoursAsync` contient une instruction `return` qui retourne un nombre entier. La déclaration de méthode doit spécifier un type de retour de `Task<int>`. La méthode asynchrone FromResult est un espace réservé pour une opération qui retourne DayOfWeek.

C#

```
public static async Task ShowTodaysInfoAsync()
{
    string message =
        $"Today is {DateTime.Today:D}\n" +
        "Today's hours of leisure: " +
        $"{await GetLeisureHoursAsync()}";

    Console.WriteLine(message);
}

static async Task<int> GetLeisureHoursAsync()
{
    DayOfWeek today = await Task.FromResult(DateTime.Now.DayOfWeek);

    int leisureHours =
        today is DayOfWeek.Saturday || today is DayOfWeek.Sunday
        ? 16 : 5;

    return leisureHours;
}
// Example output:
//    Today is Wednesday, May 24, 2017
//    Today's hours of leisure: 5
```

Quand la méthode `GetLeisureHoursAsync` est appelée à partir d'une expression await dans la méthode `ShowTodaysInfo`, cette expression récupère la valeur entière (la valeur de `leisureHours`) qui est stockée dans la tâche retournée par la méthode `GetLeisureHours`. Pour plus d'informations sur les expressions await, consultez await.

Vous pouvez mieux comprendre comment `await` récupère le résultat à partir de `Task<T>` en séparant l'appel vers `GetLeisureHoursAsync` de l'application de `await`, comme le montre le code suivant. Un appel à la méthode `GetLeisureHoursAsync` qui n'est pas immédiatement attendue retourne un type `Task<int>`, comme vous pourriez l'attendre de la déclaration de la méthode. La tâche est assignée à la variable `getLeisureHoursTask` dans l'exemple. Comme `getLeisureHoursTask` est un Task<TResult>, il contient une propriété Result de type `TResult`. Dans ce cas, `TResult` représente un type entier. Quand `await` est appliqué à `getLeisureHoursTask`, l'expression await prend pour la valeur le contenu de la propriété Result de `getLeisureHoursTask`. La valeur est assignée à la variable `ret`.

> ⓘ **Important**
>
> **Result** est une propriété bloquante. Si vous essayez d'y accéder avant la fin de sa tâche, le thread actif est bloqué tant que la tâche n'est pas terminée et que la valeur n'est pas disponible. Dans la plupart des cas, vous devez accéder à la valeur avec `await` au lieu d'accéder directement à la propriété.
>
> L'exemple précédent a récupéré la valeur de la propriété **Result** pour bloquer le thread principal afin que la méthode `Main` puisse imprimer `message` dans la console avant la fin de l'application.

C#

```
var getLeisureHoursTask = GetLeisureHoursAsync();

string message =
    $"Today is {DateTime.Today:D}\n" +
    "Today's hours of leisure: " +
    $"{await getLeisureHoursTask}";

Console.WriteLine(message);
```

Type de retour void

Vous utilisez le type de retour `void` dans les gestionnaires d'événements asynchrones, lesquels exigent un type de retour `void`. Pour les méthodes autres que les gestionnaires d'événements qui ne retournent pas de valeur, vous devez retourner Task à la place, car une méthode async qui retourne `void` ne peut pas être attendue. Tout appelant de cette méthode doit continuer jusqu'à la fin sans attendre la méthode asynchrone appelée. L'appelant doit être indépendant de toutes les valeurs ou exceptions générées par la méthode asynchrone.

L'appelant d'une méthode asynchrone de retour void ne peut pas intercepter les exceptions levées à partir de la méthode. Ces exceptions non gérées sont susceptibles d'entraîner l'échec de votre application. Si une méthode qui retourne Task ou Task<TResult> lève une exception, celle-ci est stockée dans la tâche retournée. L'exception est levée à nouveau lorsque la tâche est attendue. Assurez-vous que toute méthode asynchrone qui peut produire une exception a un type de retour de Task ou de Task<TResult> et que les appels à la méthode sont attendus.

L'exemple suivant illustre le comportement d'un gestionnaire d'événements asynchrones. Dans l'exemple de code, un gestionnaire d'événements asynchrones doit informer le thread principal lorsqu'il se termine. Le thread principal peut donc attendre la fin d'un gestionnaire d'événements asynchrones avant de quitter le programme.

C#

```
public class NaiveButton
{
    public event EventHandler? Clicked;

    public void Click()
    {
        Console.WriteLine("Somebody has clicked a button. Let's raise the
event...");
        Clicked?.Invoke(this, EventArgs.Empty);
        Console.WriteLine("All listeners are notified.");
    }
}

public class AsyncVoidExample
{
    static readonly TaskCompletionSource<bool> s_tcs = new
TaskCompletionSource<bool>();

    public static async Task MultipleEventHandlersAsync()
    {
        Task<bool> secondHandlerFinished = s_tcs.Task;

        var button = new NaiveButton();

        button.Clicked += OnButtonClicked1;
```

```
        button.Clicked += OnButtonClicked2Async;
        button.Clicked += OnButtonClicked3;

        Console.WriteLine("Before button.Click() is called...");
        button.Click();
        Console.WriteLine("After button.Click() is called...");

        await secondHandlerFinished;
    }

    private static void OnButtonClicked1(object? sender, EventArgs e)
    {
        Console.WriteLine("   Handler 1 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("   Handler 1 is done.");
    }

    private static async void OnButtonClicked2Async(object? sender,
EventArgs e)
    {
        Console.WriteLine("   Handler 2 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("   Handler 2 is about to go async...");
        await Task.Delay(500);
        Console.WriteLine("   Handler 2 is done.");
        s_tcs.SetResult(true);
    }

    private static void OnButtonClicked3(object? sender, EventArgs e)
    {
        Console.WriteLine("   Handler 3 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("   Handler 3 is done.");
    }
}
// Example output:
//
// Before button.Click() is called...
// Somebody has clicked a button. Let's raise the event...
//    Handler 1 is starting...
//    Handler 1 is done.
//    Handler 2 is starting...
//    Handler 2 is about to go async...
//    Handler 3 is starting...
//    Handler 3 is done.
// All listeners are notified.
// After button.Click() is called...
//    Handler 2 is done.
```

Types de retour async généralisés et ValueTask<TResult>

Une méthode asynchrone peut retourner n'importe quel type ayant une méthode `GetAwaiter` accessible qui retourne une instance d'un *type awaiter*. En outre, le type retourné à partir de la méthode `GetAwaiter` doit avoir l'attribut System.Runtime.CompilerServices.AsyncMethodBuilderAttribute. Vous pouvez en savoir plus dans l'article sur les Attributs lus par le compilateur ou la spécification C# pour le Modèle de générateur de types de tâches.

Cette fonctionnalité est le complément d'expressions await qui décrit les exigences de l'opérande de `await`. Les types de retour asynchrones généralisés permettent au compilateur de générer des méthodes `async` qui retournent différents types. Les types de retour asynchrones généralisés ont permis d'améliorer les performances dans les bibliothèques .NET. Étant donné que Task et Task<TResult> sont des types référence, l'allocation de mémoire dans des chemins critiques pour les performances, en particulier quand les allocations se produisent dans des boucles serrées, peuvent nuire aux performances. La prise en charge de types de retour généralisés signifie que vous pouvez retourner un type valeur léger au lieu d'un type référence pour éviter des allocations de mémoire supplémentaires.

Le .NET fournit la structure System.Threading.Tasks.ValueTask<TResult> comme implémentation légère d'une valeur retournant des tâches généralisées. L'exemple suivant utilise la structure ValueTask<TResult> pour récupérer la valeur de deux dés.

C#

```
class Program
{
    static readonly Random s_rnd = new Random();

    static async Task Main() =>
        Console.WriteLine($"You rolled {await GetDiceRollAsync()}");

    static async ValueTask<int> GetDiceRollAsync()
    {
        Console.WriteLine("Shaking dice...");

        int roll1 = await RollAsync();
        int roll2 = await RollAsync();

        return roll1 + roll2;
    }

    static async ValueTask<int> RollAsync()
    {
        await Task.Delay(500);

        int diceRoll = s_rnd.Next(1, 7);
        return diceRoll;
    }
```

```
}
// Example output:
//    Shaking dice...
//    You rolled 8
```

L'écriture d'un type de retour asynchrone généralisé est un scénario avancé et est ciblé pour une utilisation dans des environnements spécialisés. Envisagez d'utiliser les types `Task`, `Task<T>` et `ValueTask<T>`, qui couvrent la plupart des scénarios de code asynchrone.

En C# 10 et versions ultérieures, vous pouvez appliquer l'attribut `AsyncMethodBuilder` à une méthode asynchrone (au lieu de la déclaration de type de retour asynchrone) pour remplacer le générateur pour ce type. En règle générale, vous appliquez cet attribut pour utiliser un autre générateur fourni dans le runtime .NET.

Flux asynchrones avec IAsyncEnumerable<T>

Une méthode asynchrone peut retourner un *flux asynchrone*, représenté par IAsyncEnumerable<T>. Un flux asynchrone permet d'énumérer les éléments lus à partir d'un flux lorsque des éléments sont générés en blocs avec des appels asynchrones répétés. L'exemple suivant montre une méthode asynchrone qui génère un flux asynchrone :

C#

```
static async IAsyncEnumerable<string> ReadWordsFromStreamAsync()
{
    string data =
        @"This is a line of text.
              Here is the second line of text.
              And there is one more for good measure.
              Wait, that was the penultimate line.";

    using var readStream = new StringReader(data);

    string? line = await readStream.ReadLineAsync();
    while (line != null)
    {
        foreach (string word in line.Split(' ',
StringSplitOptions.RemoveEmptyEntries))
        {
            yield return word;
        }

        line = await readStream.ReadLineAsync();
    }
}
```

L'exemple précédent lit les lignes d'une chaîne de manière asynchrone. Une fois chaque ligne lue, le code énumère chaque mot de la chaîne. Les appelants énumèrent chaque mot à l'aide de l'instruction `await foreach`. La méthode attend quand elle doit lire de manière asynchrone la ligne suivante à partir de la chaîne source.

Voir aussi

- FromResult
- Traiter les tâches asynchrones terminées
- Programmation asynchrone avec async et await (C#)
- async
- await

Annuler une liste de tâches

Article • 28/03/2023

Vous pouvez annuler une application console asynchrone si vous ne souhaitez pas attendre qu'elle se termine. Les exemples de cette rubrique vous montrent comment ajouter une annulation à une application qui télécharge du contenu d'une liste de sites web. Vous pouvez annuler de nombreuses tâches en associant l'instance CancellationTokenSource à chaque tâche. Si vous sélectionnez la touche `Entrée`, vous annulez toutes les tâches qui ne sont pas encore terminées.

Ce didacticiel contient les sections suivantes :

- ✓ Création d'une application console .NET
- ✓ Écriture d'une application asynchrone prenant en charge l'annulation
- ✓ Démonstration de l'annulation de la signalisation

Prérequis

Ce didacticiel requiert les éléments suivants :

- Kit de développement logiciel (SDK) .NET 5 ou version ultérieure
- Environnement de développement intégré (IDE)
  - Nous vous recommandons Visual Studio, Visual Studio Code ou Visual Studio pour Mac.

Créer un exemple d'application

Créez une application console .NET Core. Vous pouvez en créer une à l'aide de la commande dotnet new console ou à partir de Visual Studio. Ouvrez le fichier *Program.cs* dans votre éditeur de code favori.

Remplacer les instructions using

Remplacez les instructions using existantes par ces déclarations :

C#

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Net.Http;
```

```
using System.Threading;
using System.Threading.Tasks;
```

Ajouter des champs

Dans la définition de classe `Program`, ajoutez ces trois champs :

C#

```
static readonly CancellationTokenSource s_cts = new
CancellationTokenSource();

static readonly HttpClient s_client = new HttpClient
{
    MaxResponseContentBufferSize = 1_000_000
};

static readonly IEnumerable<string> s_urlList = new string[]
{
    "https://learn.microsoft.com",
    "https://learn.microsoft.com/aspnet/core",
    "https://learn.microsoft.com/azure",
    "https://learn.microsoft.com/azure/devops",
    "https://learn.microsoft.com/dotnet",
    "https://learn.microsoft.com/dynamics365",
    "https://learn.microsoft.com/education",
    "https://learn.microsoft.com/enterprise-mobility-security",
    "https://learn.microsoft.com/gaming",
    "https://learn.microsoft.com/graph",
    "https://learn.microsoft.com/microsoft-365",
    "https://learn.microsoft.com/office",
    "https://learn.microsoft.com/powershell",
    "https://learn.microsoft.com/sql",
    "https://learn.microsoft.com/surface",
    "https://learn.microsoft.com/system-center",
    "https://learn.microsoft.com/visualstudio",
    "https://learn.microsoft.com/windows",
    "https://learn.microsoft.com/xamarin"
};
```

Le CancellationTokenSource est utilisé pour signaler une annulation demandée à un CancellationToken. Le `HttpClient` expose la possibilité d'envoyer des requêtes HTTP et de recevoir des réponses HTTP. Le `s_urlList` contient toutes les URL que l'application prévoit de traiter.

Mettre à jour le point d'entrée de l'application

Le point d'entrée principal dans l'application console est la méthode `Main`. Remplacez la méthode existante par ce qui suit :

C#

```
static async Task Main()
{
    Console.WriteLine("Application started.");
    Console.WriteLine("Press the ENTER key to cancel...\n");

    Task cancelTask = Task.Run(() =>
    {
        while (Console.ReadKey().Key != ConsoleKey.Enter)
        {
            Console.WriteLine("Press the ENTER key to cancel...");
        }

        Console.WriteLine("\nENTER key pressed: cancelling downloads.\n");
        s_cts.Cancel();
    });

    Task sumPageSizesTask = SumPageSizesAsync();

    Task finishedTask = await Task.WhenAny(new[] { cancelTask,
sumPageSizesTask });
    if (finishedTask == cancelTask)
    {
        // wait for the cancellation to take place:
        try
        {
            await sumPageSizesTask;
            Console.WriteLine("Download task completed before cancel request
was processed.");
        }
        catch (TaskCanceledException)
        {
            Console.WriteLine("Download task has been cancelled.");
        }
    }

    Console.WriteLine("Application ending.");
}
```

La méthode `Main` mise à jour est désormais considérée comme une méthode Async main, ce qui autorise un point d'entrée asynchrone dans l'exécutable. Il écrit quelques messages d'instruction dans la console, puis déclare une instance Task nommée `cancelTask`, qui lit les séquences de touches de la console. Si la touche `Entrée` est utilisée, un appel à CancellationTokenSource.Cancel() est effectué. Cela signale l'annulation. Ensuite, la variable `sumPageSizesTask` est affectée à partir de la méthode

`SumPageSizesAsync`. Les deux tâches sont ensuite passées à Task.WhenAny(Task[]), qui se poursuit une fois l'une des deux tâches terminée.

Le bloc de code suivant garantit que l'application ne se ferme pas tant que l'annulation n'a pas été traitée. Si la première tâche à effectuer est `cancelTask`, la tâche `sumPageSizeTask` est attendue. Si elle a été annulée, lorsqu'elle est attendue, cela lève une System.Threading.Tasks.TaskCanceledException. Le bloc intercepte cette exception et imprime un message.

Créer la méthode de taille de page de somme asynchrone

Sous la méthode `Main`, ajoutez la méthode `SumPageSizesAsync` :

C#

```
static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    int total = 0;
    foreach (string url in s_urlList)
    {
        int contentLength = await ProcessUrlAsync(url, s_client,
s_cts.Token);
        total += contentLength;
    }

    stopwatch.Stop();

    Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
    Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
}
```

La méthode commence par instancier et démarrer un Stopwatch. Elle effectue ensuite une boucle à travers chaque URL dans les appels `s_urlList` et appelle `ProcessUrlAsync`. Avec chaque itération, le `s_cts.Token` est passé dans la méthode `ProcessUrlAsync` et le code retourne un Task<TResult> où `TResult` est un entier :

C#

```
int total = 0;
foreach (string url in s_urlList)
{
    int contentLength = await ProcessUrlAsync(url, s_client, s_cts.Token);
```

```
    total += contentLength;
}
```

Ajouter une méthode de processus

Ajoutez la méthode suivante `ProcessUrlAsync` sous la méthode `SumPageSizesAsync` :

C#

```
static async Task<int> ProcessUrlAsync(string url, HttpClient client,
CancellationToken token)
{
    HttpResponseMessage response = await client.GetAsync(url, token);
    byte[] content = await response.Content.ReadAsByteArrayAsync(token);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}
```

Pour toute URL donnée, la méthode utilise l'instance `client` fournie pour obtenir la réponse en tant que `byte[]`. L'instance CancellationToken est passée dans les méthodes HttpClient.GetAsync(String, CancellationToken) et HttpContent.ReadAsByteArrayAsync(). `token` est utilisé pour s'inscrire à l'annulation demandée. La longueur est retournée après l'URL et la longueur est écrite vers la console.

Exemple de sortie d'application

Console

```
Application started.
Press the ENTER key to cancel...

https://learn.microsoft.com                                   37,357
https://learn.microsoft.com/aspnet/core                       85,589
https://learn.microsoft.com/azure                            398,939
https://learn.microsoft.com/azure/devops                      73,663
https://learn.microsoft.com/dotnet                            67,452
https://learn.microsoft.com/dynamics365                       48,582
https://learn.microsoft.com/education                         22,924

ENTER key pressed: cancelling downloads.

Application ending.
```

Exemple complet

Le code suivant est le texte complet du fichier *Program.cs* de l'exemple.

C#

```
using System.Diagnostics;

class Program
{
    static readonly CancellationTokenSource s_cts = new
CancellationTokenSource();

    static readonly HttpClient s_client = new HttpClient
    {
        MaxResponseContentBufferSize = 1_000_000
    };

    static readonly IEnumerable<string> s_urlList = new string[]
    {
            "https://learn.microsoft.com",
            "https://learn.microsoft.com/aspnet/core",
            "https://learn.microsoft.com/azure",
            "https://learn.microsoft.com/azure/devops",
            "https://learn.microsoft.com/dotnet",
            "https://learn.microsoft.com/dynamics365",
            "https://learn.microsoft.com/education",
            "https://learn.microsoft.com/enterprise-mobility-security",
            "https://learn.microsoft.com/gaming",
            "https://learn.microsoft.com/graph",
            "https://learn.microsoft.com/microsoft-365",
            "https://learn.microsoft.com/office",
            "https://learn.microsoft.com/powershell",
            "https://learn.microsoft.com/sql",
            "https://learn.microsoft.com/surface",
            "https://learn.microsoft.com/system-center",
            "https://learn.microsoft.com/visualstudio",
            "https://learn.microsoft.com/windows",
            "https://learn.microsoft.com/xamarin"
    };

    static async Task Main()
    {
        Console.WriteLine("Application started.");
        Console.WriteLine("Press the ENTER key to cancel...\n");

        Task cancelTask = Task.Run(() =>
        {
            while (Console.ReadKey().Key != ConsoleKey.Enter)
            {
                Console.WriteLine("Press the ENTER key to cancel...");
            }
```

```
            Console.WriteLine("\nENTER key pressed: cancelling
downloads.\n");
            s_cts.Cancel();
        });

        Task sumPageSizesTask = SumPageSizesAsync();

        Task finishedTask = await Task.WhenAny(new[] { cancelTask,
sumPageSizesTask });
        if (finishedTask == cancelTask)
        {
            // wait for the cancellation to take place:
            try
            {
                await sumPageSizesTask;
                Console.WriteLine("Download task completed before cancel
request was processed.");
            }
            catch (TaskCanceledException)
            {
                Console.WriteLine("Download task has been cancelled.");
            }
        }

        Console.WriteLine("Application ending.");
    }

    static async Task SumPageSizesAsync()
    {
        var stopwatch = Stopwatch.StartNew();

        int total = 0;
        foreach (string url in s_urlList)
        {
            int contentLength = await ProcessUrlAsync(url, s_client,
s_cts.Token);
            total += contentLength;
        }

        stopwatch.Stop();

        Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
        Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
    }

    static async Task<int> ProcessUrlAsync(string url, HttpClient client,
CancellationToken token)
    {
        HttpResponseMessage response = await client.GetAsync(url, token);
        byte[] content = await response.Content.ReadAsByteArrayAsync(token);
        Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

        return content.Length;
    }
}
```

Voir aussi

- CancellationToken
- CancellationTokenSource
- Programmation asynchrone avec async et await (C#)

Étapes suivantes

Annuler des tâches async après une période spécifique (C#)

Annuler des tâches asynch après une période spécifique

Article • 25/03/2023

Si vous ne souhaitez pas attendre la fin d'une opération asynchrone, vous pouvez l'annuler après une période spécifique à l'aide de la méthode CancellationTokenSource.CancelAfter. Cette méthode planifie l'annulation de toutes les tâches associées qui ne sont pas terminées au terme de la période indiquée par l'expression `CancelAfter`.

Cet exemple s'appuie sur le code développé dans Annuler une liste de tâches (C#) pour télécharger une liste de sites Web et afficher la longueur du contenu de chacun d'eux.

Ce didacticiel contient les sections suivantes :

- ✓ Mise à jour d'une application console .NET existante
- ✓ Planification d'une annulation

Prérequis

Ce didacticiel requiert les éléments suivants :

- Vous êtes censé avoir créé une application dans le didacticiel Annuler une liste de tâches (C#)
- Kit de développement logiciel (SDK) .NET 5 ou version ultérieure
- Environnement de développement intégré (IDE)
  - Nous vous recommandons Visual Studio, Visual Studio Code ou Visual Studio pour Mac.

Mettre à jour le point d'entrée de l'application

Remplacez la méthode existante `Main` par ce qui suit :

C#

```
static async Task Main()
{
    Console.WriteLine("Application started.");

    try
    {
        s_cts.CancelAfter(3500);
```

```
        await SumPageSizesAsync();
    }
    catch (OperationCanceledException)
    {
        Console.WriteLine("\nTasks cancelled: timed out.\n");
    }
    finally
    {
        s_cts.Dispose();
    }

    Console.WriteLine("Application ending.");
}
```

La méthode mise à jour `Main` écrit quelques messages d'instruction dans la console. Dans l'instruction try-catch, un appel à CancellationTokenSource.CancelAfter(Int32) planifie une annulation. Cela signale l'annulation après un certain temps.

Ensuite, la méthode `SumPageSizesAsync` est attendue. Si le traitement de toutes les URL se produit plus rapidement que l'annulation planifiée, l'application se termine. Toutefois, si l'annulation planifiée est déclenchée avant que toutes les URL soient traitées, une OperationCanceledException est levée.

Exemple de sortie d'application

Console

```
Application started.

https://learn.microsoft.com                                37,357
https://learn.microsoft.com/aspnet/core                    85,589
https://learn.microsoft.com/azure                         398,939
https://learn.microsoft.com/azure/devops                   73,663

Tasks cancelled: timed out.

Application ending.
```

Exemple complet

Le code suivant est le texte complet du fichier *Program.cs* de l'exemple.

C#

```
using System.Diagnostics;
```

```
class Program
{
    static readonly CancellationTokenSource s_cts = new
CancellationTokenSource();

    static readonly HttpClient s_client = new HttpClient
    {
        MaxResponseContentBufferSize = 1_000_000
    };

    static readonly IEnumerable<string> s_urlList = new string[]
    {
            "https://learn.microsoft.com",
            "https://learn.microsoft.com/aspnet/core",
            "https://learn.microsoft.com/azure",
            "https://learn.microsoft.com/azure/devops",
            "https://learn.microsoft.com/dotnet",
            "https://learn.microsoft.com/dynamics365",
            "https://learn.microsoft.com/education",
            "https://learn.microsoft.com/enterprise-mobility-security",
            "https://learn.microsoft.com/gaming",
            "https://learn.microsoft.com/graph",
            "https://learn.microsoft.com/microsoft-365",
            "https://learn.microsoft.com/office",
            "https://learn.microsoft.com/powershell",
            "https://learn.microsoft.com/sql",
            "https://learn.microsoft.com/surface",
            "https://learn.microsoft.com/system-center",
            "https://learn.microsoft.com/visualstudio",
            "https://learn.microsoft.com/windows",
            "https://learn.microsoft.com/xamarin"
    };

    static async Task Main()
    {
        Console.WriteLine("Application started.");

        try
        {
            s_cts.CancelAfter(3500);

            await SumPageSizesAsync();
        }
        catch (OperationCanceledException)
        {
            Console.WriteLine("\nTasks cancelled: timed out.\n");
        }
        finally
        {
            s_cts.Dispose();
        }

        Console.WriteLine("Application ending.");
    }
```

```
    static async Task SumPageSizesAsync()
    {
        var stopwatch = Stopwatch.StartNew();

        int total = 0;
        foreach (string url in s_urlList)
        {
            int contentLength = await ProcessUrlAsync(url, s_client,
s_cts.Token);
            total += contentLength;
        }

        stopwatch.Stop();

        Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
        Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
    }

    static async Task<int> ProcessUrlAsync(string url, HttpClient client,
CancellationToken token)
    {
        HttpResponseMessage response = await client.GetAsync(url, token);
        byte[] content = await response.Content.ReadAsByteArrayAsync(token);
        Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

        return content.Length;
    }
}
```

Voir aussi

- CancellationToken
- CancellationTokenSource
- Programmation asynchrone avec async et await (C#)
- Annuler une liste de tâches (C#)

Traiter les tâches asynchrones terminées (C#)

Article • 23/05/2023

En utilisant Task.WhenAny, vous pouvez démarrer plusieurs tâches en même temps et les traiter une par une, une fois qu'elles sont terminées, au lieu de les traiter dans l'ordre dans lequel elles ont démarré.

L'exemple suivant utilise une requête pour créer une collection de tâches. Chaque tâche télécharge le contenu d'un site web spécifié. À chaque itération d'une boucle while, un appel attendu à WhenAny retourne la tâche de la collection de tâches dont le téléchargement se termine en premier. Cette tâche est supprimée de la collection et traitée. La boucle se répète jusqu'à ce que la collection ne contienne plus aucune tâche.

Prérequis

Vous pouvez suivre ce didacticiel en utilisant l'une des options suivantes :

- Visual Studio 2022 avec la charge de travail **Développement .NET Desktop** installée. Le kit SDK .NET est automatiquement installé lorsque vous sélectionnez cette charge de travail.
- Le kit de développement logiciel (SDK) .NET avec un éditeur de code de votre choix, tel que Visual Studio Code.

Créer un exemple d'application

Créez une application console .NET Core. Vous pouvez en créer une à l'aide de la commande dotnet new console ou à partir de Visual Studio.

Ouvrez le fichier *Program.cs* dans votre éditeur de code, puis remplacez le code existant par ce code :

C#

```
using System.Diagnostics;

namespace ProcessTasksAsTheyFinish;

class Program
{
    static void Main(string[] args)
    {
```

```
        Console.WriteLine("Hello World!");
    }
}
```

Ajouter des champs

Dans la définition de la classe `Program`, ajoutez les deux champs suivants :

C#

```
static readonly HttpClient s_client = new HttpClient
{
    MaxResponseContentBufferSize = 1_000_000
};

static readonly IEnumerable<string> s_urlList = new string[]
{
    "https://learn.microsoft.com",
    "https://learn.microsoft.com/aspnet/core",
    "https://learn.microsoft.com/azure",
    "https://learn.microsoft.com/azure/devops",
    "https://learn.microsoft.com/dotnet",
    "https://learn.microsoft.com/dynamics365",
    "https://learn.microsoft.com/education",
    "https://learn.microsoft.com/enterprise-mobility-security",
    "https://learn.microsoft.com/gaming",
    "https://learn.microsoft.com/graph",
    "https://learn.microsoft.com/microsoft-365",
    "https://learn.microsoft.com/office",
    "https://learn.microsoft.com/powershell",
    "https://learn.microsoft.com/sql",
    "https://learn.microsoft.com/surface",
    "https://learn.microsoft.com/system-center",
    "https://learn.microsoft.com/visualstudio",
    "https://learn.microsoft.com/windows",
    "https://learn.microsoft.com/xamarin"
};
```

Le `HttpClient` expose la possibilité d'envoyer des requêtes HTTP et de recevoir des réponses HTTP. Le `s_urlList` contient toutes les URL que l'application prévoit de traiter.

Mettre à jour le point d'entrée de l'application

Le point d'entrée principal dans l'application console est la méthode `Main`. Remplacez la méthode existante par ce qui suit :

C#

```
static Task Main() => SumPageSizesAsync();
```

La méthode `Main` mise à jour est désormais considérée comme une méthode Async main, ce qui autorise un point d'entrée asynchrone dans l'exécutable. Il est exprimé sous la forme d'un appel à `SumPageSizesAsync`.

Créer la méthode de taille de page de somme asynchrone

Sous la méthode `Main`, ajoutez la méthode `SumPageSizesAsync` :

C#

```
static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    IEnumerable<Task<int>> downloadTasksQuery =
        from url in s_urlList
        select ProcessUrlAsync(url, s_client);

    List<Task<int>> downloadTasks = downloadTasksQuery.ToList();

    int total = 0;
    while (downloadTasks.Any())
    {
        Task<int> finishedTask = await Task.WhenAny(downloadTasks);
        downloadTasks.Remove(finishedTask);
        total += await finishedTask;
    }

    stopwatch.Stop();

    Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
    Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
}
```

La boucle `while` supprime l'une des tâches de chaque itération. Une fois chaque tâche terminée, la boucle se termine. La méthode commence par instancier et démarrer un Stopwatch. Elle comprend ensuite une requête qui, lorsqu'elle est exécutée, crée une collection de tâches. Chaque appel à `ProcessUrlAsync` dans le code suivant retourne un Task<TResult>, où `TResult` est un entier :

C#

```
IEnumerable<Task<int>> downloadTasksQuery =
    from url in s_urlList
    select ProcessUrlAsync(url, s_client);
```

En raison d'une exécution différée avec LINQ, vous appelez Enumerable.ToList pour démarrer chaque tâche.

C#

```
List<Task<int>> downloadTasks = downloadTasksQuery.ToList();
```

La boucle `while` effectue les étapes suivantes pour chaque tâche dans la collection :

1. Elle attend un appel à `WhenAny` pour identifier la première tâche dans la collection qui a terminé son téléchargement.

   C#

   ```
   Task<int> finishedTask = await Task.WhenAny(downloadTasks);
   ```

2. Elle supprime cette tâche de la collection.

   C#

   ```
   downloadTasks.Remove(finishedTask);
   ```

3. Elle attend `finishedTask`, qui est retourné par un appel à `ProcessUrlAsync`. La variable `finishedTask` est un Task<TResult> où `TResult` est un entier. La tâche est déjà terminée, mais vous l'attendez pour récupérer la longueur du site web téléchargé, comme le montre l'exemple suivant. Si la tâche est défaillante, `await` lève la première exception enfant stockée dans `AggregateException`, contrairement à la lecture de la propriété Task<TResult>.Result, qui lève le `AggregateException`.

   C#

   ```
   total += await finishedTask;
   ```

Ajouter une méthode de processus

Ajoutez la méthode suivante `ProcessUrlAsync` sous la méthode `SumPageSizesAsync` :

C#

```
static async Task<int> ProcessUrlAsync(string url, HttpClient client)
{
    byte[] content = await client.GetByteArrayAsync(url);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}
```

Pour toute URL donnée, la méthode utilise l'instance `client` fournie pour obtenir la réponse en tant que `byte[]`. La longueur est retournée après l'URL et la longueur est écrite vers la console.

Exécutez le programme plusieurs fois pour vérifier que les longueurs téléchargées n'apparaissent pas toujours dans le même ordre.

> ⊗ **Attention**
>
> Vous pouvez utiliser `WhenAny` dans une boucle, comme décrit dans l'exemple, pour résoudre les problèmes qui impliquent un petit nombre de tâches. Cependant, d'autres approches sont plus efficaces si vous avez un grand nombre de tâches à traiter. Pour plus d'informations et d'exemples, consultez **Traitement des tâches une fois terminées**.

Exemple complet

Le code suivant est le texte complet du fichier *Program.cs* de l'exemple.

C#

```
using System.Diagnostics;

HttpClient s_client = new()
{
    MaxResponseContentBufferSize = 1_000_000
};

IEnumerable<string> s_urlList = new string[]
{
    "https://learn.microsoft.com",
    "https://learn.microsoft.com/aspnet/core",
    "https://learn.microsoft.com/azure",
    "https://learn.microsoft.com/azure/devops",
    "https://learn.microsoft.com/dotnet",
    "https://learn.microsoft.com/dynamics365",
```

```
    "https://learn.microsoft.com/education",
    "https://learn.microsoft.com/enterprise-mobility-security",
    "https://learn.microsoft.com/gaming",
    "https://learn.microsoft.com/graph",
    "https://learn.microsoft.com/microsoft-365",
    "https://learn.microsoft.com/office",
    "https://learn.microsoft.com/powershell",
    "https://learn.microsoft.com/sql",
    "https://learn.microsoft.com/surface",
    "https://learn.microsoft.com/system-center",
    "https://learn.microsoft.com/visualstudio",
    "https://learn.microsoft.com/windows",
    "https://learn.microsoft.com/xamarin"
};

await SumPageSizesAsync();

async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    IEnumerable<Task<int>> downloadTasksQuery =
        from url in s_urlList
        select ProcessUrlAsync(url, s_client);

    List<Task<int>> downloadTasks = downloadTasksQuery.ToList();

    int total = 0;
    while (downloadTasks.Any())
    {
        Task<int> finishedTask = await Task.WhenAny(downloadTasks);
        downloadTasks.Remove(finishedTask);
        total += await finishedTask;
    }

    stopwatch.Stop();

    Console.WriteLine($"\nTotal bytes returned:    {total:#,#}");
    Console.WriteLine($"Elapsed time:              {stopwatch.Elapsed}\n");
}

static async Task<int> ProcessUrlAsync(string url, HttpClient client)
{
    byte[] content = await client.GetByteArrayAsync(url);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}

// Example output:
// https://learn.microsoft.com                                   132,517
// https://learn.microsoft.com/powershell                         57,375
// https://learn.microsoft.com/gaming                             33,549
// https://learn.microsoft.com/aspnet/core                        88,714
// https://learn.microsoft.com/surface                            39,840
```

```
// https://learn.microsoft.com/enterprise-mobility-security          30,903
// https://learn.microsoft.com/microsoft-365                         67,867
// https://learn.microsoft.com/windows                               26,816
// https://learn.microsoft.com/xamarin                               57,958
// https://learn.microsoft.com/dotnet                                78,706
// https://learn.microsoft.com/graph                                 48,277
// https://learn.microsoft.com/dynamics365                           49,042
// https://learn.microsoft.com/office                                67,867
// https://learn.microsoft.com/system-center                         42,887
// https://learn.microsoft.com/education                             38,636
// https://learn.microsoft.com/azure                                421,663
// https://learn.microsoft.com/visualstudio                          30,925
// https://learn.microsoft.com/sql                                   54,608
// https://learn.microsoft.com/azure/devops                          86,034

// Total bytes returned:    1,454,184
// Elapsed time:            00:00:01.1290403
```

Voir aussi

- WhenAny
- Programmation asynchrone avec async et await (C#)

Accès asynchrone aux fichiers (C#)

Article • 10/05/2023

Vous pouvez utiliser la fonctionnalité Async pour accéder à des fichiers. La fonctionnalité async vous permet d'appeler des méthodes asynchrones sans utiliser de rappels ni fractionner votre code entre plusieurs méthodes ou expressions lambda. Pour rendre le code synchrone asynchrone, il vous suffit d'appeler une méthode asynchrone au lieu d'une méthode synchrone, puis d'ajouter quelques mots clés au code.

Vous pouvez considérer les raisons suivantes pour ajouter de l'asynchrone aux appels d'accès aux fichiers :

- ✓ L'asynchronisme rend les applications d'interface utilisateur plus réactives parce que le thread d'interface utilisateur qui lance l'exécution peut effectuer d'autres tâches. Si le thread d'interface utilisateur doit exécuter du code qui prend du temps (par exemple, plus de 50 millisecondes), l'interface utilisateur peut figer jusqu'à ce que l'E/S soit terminée et que le thread d'interface utilisateur puisse traiter à nouveau des entrées au clavier et à la souris et d'autres événements.
- ✓ L'asynchronisme améliore l'extensibilité d'ASP.NET et d'autres applications serveur en réduisant le besoin de threads. Si l'application utilise un thread dédié par réponse et que mille demandes sont traitées simultanément, mille threads sont nécessaires. Les opérations asynchrones n'ont souvent pas besoin d'utiliser un thread pendant l'attente. Elles utilisent le thread de terminaison d'E/S existant brièvement à la fin.
- ✓ La latence d'une opération d'accès à un fichier peut être très basse sous les conditions actuelles, mais la latence peut considérablement augmenter à l'avenir. Par exemple, un fichier peut être déplacé vers un serveur qui se trouve à travers le monde.
- ✓ La charge mémoire supplémentaire pour l'utilisation de la fonctionnalité Async est faible.
- ✓ Les tâches asynchrones peuvent facilement être exécutées en parallèle.

Utiliser les classes appropriées

Les exemples simples de cette rubrique illustrent File.WriteAllTextAsync et File.ReadAllTextAsync. Pour un contrôle précis des opérations d'E/S de fichier, utilisez la classe FileStream, qui a une option qui entraîne l'exécution d'E/S asynchrones au niveau du système d'exploitation. En utilisant cette option, vous pouvez éviter de bloquer un thread de pool de threads dans de nombreux cas. Pour l'activer, spécifiez l'argument `useAsync=true` ou `options=FileOptions.Asynchronous` dans l'appel de constructeur.

Vous ne pouvez pas utiliser cette option avec StreamReader et StreamWriter si vous les ouvrez directement en spécifiant un chemin d'accès au fichier. Toutefois, vous pouvez utiliser cette option si vous leur fournissez un Stream ouvert par la classe FileStream. Les appels asynchrones sont plus rapides dans les applications d'interface utilisateur même si un thread de pool de threads est bloqué, car le thread d'interface utilisateur n'est pas bloqué pendant l'attente.

Écrire du texte

Les exemples suivants écrivent du texte dans un fichier. À chaque instruction await, la méthode s'arrête immédiatement. Quand l'E/S de fichier est terminée, la méthode reprend à l'instruction qui suit l'instruction await. Le modificateur asynchrone se trouve dans la définition de méthodes qui utilisent l'instruction await.

Exemple simple

C#

```
public async Task SimpleWriteAsync()
{
    string filePath = "simple.txt";
    string text = $"Hello World";

    await File.WriteAllTextAsync(filePath, text);
}
```

Exemple de contrôle fini

C#

```
public async Task ProcessWriteAsync()
{
    string filePath = "temp.txt";
    string text = $"Hello World{Environment.NewLine}";

    await WriteTextAsync(filePath, text);
}

async Task WriteTextAsync(string filePath, string text)
{
    byte[] encodedText = Encoding.Unicode.GetBytes(text);

    using var sourceStream =
        new FileStream(
            filePath,
```

```
            FileMode.Create, FileAccess.Write, FileShare.None,
            bufferSize: 4096, useAsync: true);

    await sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
}
```

L'exemple d'origine comprend l'instruction `await sourceStream.WriteAsync(encodedText, 0, encodedText.Length);`, qui est une contraction des deux instructions suivantes :

C#

```
Task theTask = sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
await theTask;
```

La première instruction retourne une tâche et provoque le début du traitement du fichier. La deuxième instruction avec await provoque la fin immédiate de la méthode et retourne une tâche différente. Quand le traitement du fichier se termine plus loin, l'exécution retourne à l'instruction qui suit l'attente.

Lire le texte

Les exemples suivants lisent du texte dans un fichier.

Exemple simple

C#

```
public async Task SimpleReadAsync()
{
    string filePath = "simple.txt";
    string text = await File.ReadAllTextAsync(filePath);

    Console.WriteLine(text);
}
```

Exemple de contrôle fini

Le texte est mis en mémoire tampon et, dans cet exemple, est placé dans un StringBuilder. Contrairement à l'exemple précédent, l'évaluation de l'instruction await génère une valeur. La méthode ReadAsync retourne un Task<Int32>, de sorte que l'évaluation de l'expression await génère une valeur `Int32` `numRead` une fois l'opération effectuée. Pour plus d'informations, consultez Types de retour Async (C#).

C#

```
public async Task ProcessReadAsync()
{
    try
    {
        string filePath = "temp.txt";
        if (File.Exists(filePath) != false)
        {
            string text = await ReadTextAsync(filePath);
            Console.WriteLine(text);
        }
        else
        {
            Console.WriteLine($"file not found: {filePath}");
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}

async Task<string> ReadTextAsync(string filePath)
{
    using var sourceStream =
        new FileStream(
            filePath,
            FileMode.Open, FileAccess.Read, FileShare.Read,
            bufferSize: 4096, useAsync: true);

    var sb = new StringBuilder();

    byte[] buffer = new byte[0x1000];
    int numRead;
    while ((numRead = await sourceStream.ReadAsync(buffer, 0,
buffer.Length)) != 0)
    {
        string text = Encoding.Unicode.GetString(buffer, 0, numRead);
        sb.Append(text);
    }

    return sb.ToString();
}
```

E/S asynchrones en parallèle

Les exemples suivants illustrent le traitement parallèle en écrivant 10 fichiers texte.

Exemple simple

C#

```
public async Task SimpleParallelWriteAsync()
{
    string folder = Directory.CreateDirectory("tempfolder").Name;
    IList<Task> writeTaskList = new List<Task>();

    for (int index = 11; index <= 20; ++ index)
    {
        string fileName = $"file-{index:00}.txt";
        string filePath = $"{folder}/{fileName}";
        string text = $"In file {index}{Environment.NewLine}";

        writeTaskList.Add(File.WriteAllTextAsync(filePath, text));
    }

    await Task.WhenAll(writeTaskList);
}
```

Exemple de contrôle fini

Pour chaque fichier, la méthode WriteAsync retourne une tâche qui est ensuite ajoutée à une liste des tâches. L'instruction `await Task.WhenAll(tasks);` quitte la méthode et reprend dans cette dernière quand le traitement du fichier est terminé pour toutes les tâches.

L'exemple ferme toutes les instances de FileStream dans un bloc `finally` une fois les tâches effectuées. Si chaque `FileStream` était plutôt créé dans une instruction `using`, `FileStream` pourrait être libéré avant que la tâche ne soit terminée.

Toute amélioration des performances provient presque entièrement du traitement parallèle et non du traitement asynchrone. Les avantages du mode asynchrone sont qu'il n'attache pas plusieurs threads et qu'il ne bloque pas le thread d'interface utilisateur.

C#

```
public async Task ProcessMultipleWritesAsync()
{
    IList<FileStream> sourceStreams = new List<FileStream>();

    try
    {
        string folder = Directory.CreateDirectory("tempfolder").Name;
        IList<Task> writeTaskList = new List<Task>();

        for (int index = 1; index <= 10; ++ index)
        {
            string fileName = $"file-{index:00}.txt";
```

```
            string filePath = $"{folder}/{fileName}";

            string text = $"In file {index}{Environment.NewLine}";
            byte[] encodedText = Encoding.Unicode.GetBytes(text);

            var sourceStream =
                new FileStream(
                    filePath,
                    FileMode.Create, FileAccess.Write, FileShare.None,
                    bufferSize: 4096, useAsync: true);

            Task writeTask = sourceStream.WriteAsync(encodedText, 0,
encodedText.Length);
            sourceStreams.Add(sourceStream);

            writeTaskList.Add(writeTask);
        }

        await Task.WhenAll(writeTaskList);
    }
    finally
    {
        foreach (FileStream sourceStream in sourceStreams)
        {
            sourceStream.Close();
        }
    }
}
```

Quand vous utilisez les méthodes WriteAsync et ReadAsync, vous pouvez spécifier un CancellationToken, qui vous permet d'annuler l'opération en cours de route. Pour plus d'informations, consultez Annulation dans les threads managés.

Voir aussi

- Programmation asynchrone avec async et await (C#)
- Types de retour async (C#)

Attributes

Article • 03/15/2023

Attributes provide a powerful method of associating metadata, or declarative information, with code (assemblies, types, methods, properties, and so forth). After an attribute is associated with a program entity, the attribute can be queried at run time by using a technique called *reflection*.

Attributes have the following properties:

- Attributes add metadata to your program. *Metadata* is information about the types defined in a program. All .NET assemblies contain a specified set of metadata that describes the types and type members defined in the assembly. You can add custom attributes to specify any additional information that is required.
- You can apply one or more attributes to entire assemblies, modules, or smaller program elements such as classes and properties.
- Attributes can accept arguments in the same way as methods and properties.
- Your program can examine its own metadata or the metadata in other programs by using reflection.

Reflection provides objects (of type Type) that describe assemblies, modules, and types. You can use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties. If you're using attributes in your code, reflection enables you to access them. For more information, see Attributes.

Here's a simple example of reflection using the GetType() method - inherited by all types from the `Object` base class - to obtain the type of a variable:

> **Note**
>
> Make sure you add `using System;` and `using System.Reflection;` at the top of your *.cs* file.

C#

```
// Using GetType to obtain type information:
int i = 42;
Type type = i.GetType();
Console.WriteLine(type);
```

The output is: `System.Int32`.

The following example uses reflection to obtain the full name of the loaded assembly.

C#

```
// Using Reflection to get information of an Assembly:
Assembly info = typeof(int).Assembly;
Console.WriteLine(info);
```

The output is something like: `System.Private.CoreLib, Version=7.0.0.0, Culture=neutral, PublicKeyToken=7cec85d7bea7798e`.

> **Note**
>
> The C# keywords `protected` and `internal` have no meaning in Intermediate Language (IL) and are not used in the reflection APIs. The corresponding terms in IL are *Family* and *Assembly*. To identify an `internal` method using reflection, use the **IsAssembly** property. To identify a `protected internal` method, use the **IsFamilyOrAssembly**.

Using attributes

Attributes can be placed on almost any declaration, though a specific attribute might restrict the types of declarations on which it's valid. In C#, you specify an attribute by placing the name of the attribute enclosed in square brackets (`[]`) above the declaration of the entity to which it applies.

In this example, the SerializableAttribute attribute is used to apply a specific characteristic to a class:

C#

```
[Serializable]
public class SampleClass
{
    // Objects of this type can be serialized.
}
```

A method with the attribute DllImportAttribute is declared like the following example:

C#

```
[System.Runtime.InteropServices.DllImport("user32.dll")]
extern static void SampleMethod();
```

More than one attribute can be placed on a declaration as the following example shows:

C#

```
void MethodA([In][Out] ref double x) { }
void MethodB([Out][In] ref double x) { }
void MethodC([In, Out] ref double x) { }
```

Some attributes can be specified more than once for a given entity. An example of such a multiuse attribute is ConditionalAttribute:

C#

```
[Conditional("DEBUG"), Conditional("TEST1")]
void TraceMethod()
{
    // ...
}
```

> ⓘ Note
>
> By convention, all attribute names end with the word "Attribute" to distinguish them from other items in the .NET libraries. However, you do not need to specify the attribute suffix when using attributes in code. For example, `[DllImport]` is equivalent to `[DllImportAttribute]`, but `DllImportAttribute` is the attribute's actual name in the .NET Class Library.

Attribute parameters

Many attributes have parameters, which can be positional, unnamed, or named. Any positional parameters must be specified in a certain order and can't be omitted. Named parameters are optional and can be specified in any order. Positional parameters are specified first. For example, these three attributes are equivalent:

C#

```
[DllImport("user32.dll")]
[DllImport("user32.dll", SetLastError=false, ExactSpelling=false)]
[DllImport("user32.dll", ExactSpelling=false, SetLastError=false)]
```

The first parameter, the DLL name, is positional and always comes first; the others are named. In this case, both named parameters default to false, so they can be omitted. Positional parameters correspond to the parameters of the attribute constructor. Named or optional parameters correspond to either properties or fields of the attribute. Refer to the individual attribute's documentation for information on default parameter values.

For more information on allowed parameter types, see the Attributes section of the C# language specification

Attribute targets

The *target* of an attribute is the entity that the attribute applies to. For example, an attribute may apply to a class, a particular method, or an entire assembly. By default, an attribute applies to the element that follows it. But you can also explicitly identify, for example, whether an attribute is applied to a method, or to its parameter, or to its return value.

To explicitly identify an attribute target, use the following syntax:

```
C#

[target : attribute-list]
```

The list of possible `target` values is shown in the following table.

Target value	Applies to
`assembly`	Entire assembly
`module`	Current assembly module
`field`	Field in a class or a struct
`event`	Event
`method`	Method or `get` and `set` property accessors
`param`	Method parameters or `set` property accessor parameters
`property`	Property
`return`	Return value of a method, property indexer, or `get` property accessor
`type`	Struct, class, interface, enum, or delegate

You would specify the `field` target value to apply an attribute to the backing field created for an auto-implemented property.

The following example shows how to apply attributes to assemblies and modules. For more information, see Common Attributes (C#).

C#

```
using System;
using System.Reflection;
[assembly: AssemblyTitleAttribute("Production assembly 4")]
[module: CLSCompliant(true)]
```

The following example shows how to apply attributes to methods, method parameters, and method return values in C#.

C#

```
// default: applies to method
[ValidatedContract]
int Method1() { return 0; }

// applies to method
[method: ValidatedContract]
int Method2() { return 0; }

// applies to parameter
int Method3([ValidatedContract] string contract) { return 0; }

// applies to return value
[return: ValidatedContract]
int Method4() { return 0; }
```

> ⓘ **Note**
>
> Regardless of the targets on which `ValidatedContract` is defined to be valid, the `return` target has to be specified, even if `ValidatedContract` were defined to apply only to return values. In other words, the compiler will not use `AttributeUsage` information to resolve ambiguous attribute targets. For more information, see **AttributeUsage**.

Common uses for attributes

The following list includes a few of the common uses of attributes in code:

- Marking methods using the `WebMethod` attribute in Web services to indicate that the method should be callable over the SOAP protocol. For more information, see WebMethodAttribute.
- Describing how to marshal method parameters when interoperating with native code. For more information, see MarshalAsAttribute.
- Describing the COM properties for classes, methods, and interfaces.
- Calling unmanaged code using the DllImportAttribute class.
- Describing your assembly in terms of title, version, description, or trademark.
- Describing which members of a class to serialize for persistence.
- Describing how to map between class members and XML nodes for XML serialization.
- Describing the security requirements for methods.
- Specifying characteristics used to enforce security.
- Controlling optimizations by the just-in-time (JIT) compiler so the code remains easy to debug.
- Obtaining information about the caller to a method.

Reflection overview

Reflection is useful in the following situations:

- When you have to access attributes in your program's metadata. For more information, see Retrieving Information Stored in Attributes.
- For examining and instantiating types in an assembly.
- For building new types at run time. Use classes in System.Reflection.Emit.
- For performing late binding, accessing methods on types created at run time. See the article Dynamically Loading and Using Types.

Related sections

For more information:

- Common Attributes (C#)
- Caller Information (C#)
- Attributes
- Reflection
- Viewing Type Information
- Reflection and Generic Types
- System.Reflection.Emit
- Retrieving Information Stored in Attributes

Créer des attributs personnalisés

Article • 20/03/2023

Vous pouvez créer vos propres attributs personnalisés en définissant une classe d'attributs. Cette classe dérive directement ou indirectement d'Attribute, ce qui permet d'identifier rapidement et facilement des définitions d'attributs dans des métadonnées. Supposons que vous souhaitiez baliser des types avec le nom du programmeur qui les a écrits. Vous pouvez définir une classe d'attributs `Author` personnalisés :

C#

```
[System.AttributeUsage(System.AttributeTargets.Class |
                       System.AttributeTargets.Struct)
]
public class AuthorAttribute : System.Attribute
{
    private string Name;
    public double Version;

    public AuthorAttribute(string name)
    {
        Name = name;
        Version = 1.0;
    }
}
```

Le nom de la classe `AuthorAttribute` est le nom de l'attribut, `Author` plus le sufixe `Attribute`. Étant dérivée de `System.Attribute`, il s'agit donc d'une classe d'attributs personnalisés. Les paramètres du constructeur sont les paramètres positionnels de l'attribut personnalisé. Dans cet exemple, `name` est un paramètre positionnel. Les propriétés ou champs de type public accessibles en lecture/écriture sont des paramètres nommés. Dans le cas présent, `version` est le seul paramètre nommé. Notez l'utilisation de l'attribut `AttributeUsage` pour rendre l'attribut `Author` valide uniquement sur les déclarations de classe et de `struct`.

Vous pouvez utiliser ce nouvel attribut de la manière suivante :

C#

```
[Author("P. Ackerman", Version = 1.1)]
class SampleClass
{
    // P. Ackerman's code goes here...
}
```

`AttributeUsage` a un paramètre nommé, `AllowMultiple`, qui permet de déclarer un attribut personnalisé comme étant à usage unique ou multiple. Dans l'exemple de code suivant, un attribut à usage multiple est créé.

C#

```
[System.AttributeUsage(System.AttributeTargets.Class |
                       System.AttributeTargets.Struct,
                       AllowMultiple = true)  // Multiuse attribute.
]
public class AuthorAttribute : System.Attribute
{
    string Name;
    public double Version;

    public AuthorAttribute(string name)
    {
        Name = name;

        // Default value.
        Version = 1.0;
    }

    public string GetName() => Name;
}
```

Dans l'exemple de code suivant, plusieurs attributs du même type sont appliqués à une classe.

C#

```
[Author("P. Ackerman"), Author("R. Koch", Version = 2.0)]
public class ThirdClass
{
    // ...
}
```

Voir aussi

- System.Reflection
- Écriture des attributs personnalisés
- AttributeUsage (C#)

Accéder à des attributs à l'aide de la réflexion

Article • 20/03/2023

La définition d'attributs personnalisés et leur ajout à votre code source présentent peu d'intérêt si vous ne pouvez pas ensuite récupérer et manipuler ces informations. La réflexion vous permet de récupérer les informations qui ont été définies à l'aide d'attributs personnalisés. La méthode clé est `GetCustomAttributes`. Elle retourne un tableau d'objets qui sont les équivalents des attributs du code source au moment de l'exécution. Cette méthode a plusieurs versions surchargées. Pour plus d'informations, consultez Attribute.

Par exemple, cette spécification d'attribut :

C#

```
[Author("P. Ackerman", Version = 1.1)]
class SampleClass { }
```

équivaut fonctionnellement au code suivant :

C#

```
var anonymousAuthorObject = new Author("P. Ackerman")
{
    Version = 1.1
};
```

Toutefois, le code n'est pas exécuté tant qu'une requête n'est pas effectuée sur `SampleClass` pour obtenir les attributs. L'appel de `GetCustomAttributes` sur `SampleClass` entraîne la construction et l'initialisation d'un objet `Author`. Si la classe a d'autres attributs, les autres objets attribut sont construits de la même façon. `GetCustomAttributes` retourne ensuite l'objet `Author` et les autres objets attribut dans un tableau. Vous pouvez ensuite itérer sur ce tableau, déterminer quels attributs ont été appliqués en fonction du type de chaque élément du tableau et extraire des informations des objets attribut.

Voici un exemple complet. Un attribut personnalisé est défini, appliqué à plusieurs entités, puis récupéré par réflexion.

C#

```
// Multiuse attribute.
[System.AttributeUsage(System.AttributeTargets.Class |
                       System.AttributeTargets.Struct,
                       AllowMultiple = true)  // Multiuse attribute.
]
public class AuthorAttribute : System.Attribute
{
    string Name;
    public double Version;

    public AuthorAttribute(string name)
    {
        Name = name;

        // Default value.
        Version = 1.0;
    }

    public string GetName() => Name;
}

// Class with the Author attribute.
[Author("P. Ackerman")]
public class FirstClass
{
    // ...
}

// Class without the Author attribute.
public class SecondClass
{
    // ...
}

// Class with multiple Author attributes.
[Author("P. Ackerman"), Author("R. Koch", Version = 2.0)]
public class ThirdClass
{
    // ...
}

class TestAuthorAttribute
{
    public static void Test()
    {
        PrintAuthorInfo(typeof(FirstClass));
        PrintAuthorInfo(typeof(SecondClass));
        PrintAuthorInfo(typeof(ThirdClass));
    }

    private static void PrintAuthorInfo(System.Type t)
    {
        System.Console.WriteLine($"Author information for {t}");
```

```
        // Using reflection.
        System.Attribute[] attrs = System.Attribute.GetCustomAttributes(t);
// Reflection.

        // Displaying output.
        foreach (System.Attribute attr in attrs)
        {
            if (attr is AuthorAttribute a)
            {
                System.Console.WriteLine($"   {a.GetName()}, version
{a.Version:f}");
            }
        }
    }
}
/* Output:
    Author information for FirstClass
       P. Ackerman, version 1.00
    Author information for SecondClass
    Author information for ThirdClass
       R. Koch, version 2.00
       P. Ackerman, version 1.00
*/
```

Voir aussi

- System.Reflection
- Attribute
- Récupération des informations stockées dans les attributs

Guide pratique pour créer une union C/C++ à l'aide d'attributs en C#

Article • 20/03/2023

Vous pouvez personnaliser la disposition des structs en mémoire à l'aide d'attributs. Par exemple, vous pouvez créer ce qu'on appelle une union en C/C++ à l'aide des attributs `StructLayout(LayoutKind.Explicit)` et `FieldOffset`.

Dans ce segment de code, tous les champs de `TestUnion` débutent au même emplacement en mémoire.

C#

```
[System.Runtime.InteropServices.StructLayout(LayoutKind.Explicit)]
struct TestUnion
{
    [System.Runtime.InteropServices.FieldOffset(0)]
    public int i;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public double d;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public char c;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public byte b;
}
```

Voici un autre exemple de code où les champs débutent à différents emplacements définis explicitement.

C#

```
[System.Runtime.InteropServices.StructLayout(LayoutKind.Explicit)]
struct TestExplicit
{
    [System.Runtime.InteropServices.FieldOffset(0)]
    public long lg;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public int i1;

    [System.Runtime.InteropServices.FieldOffset(4)]
    public int i2;

    [System.Runtime.InteropServices.FieldOffset(8)]
```

```
    public double d;

    [System.Runtime.InteropServices.FieldOffset(12)]
    public char c;

    [System.Runtime.InteropServices.FieldOffset(14)]
    public byte b;
}
```

Les deux champs entiers, `i1` et `i2` combinés, partagent leurs emplacements de mémoire avec `lg`. Soit `lg` utilise les 8 premiers octets, soit `i1` utilise les 4 premiers octets et `i2` les 4 octets suivants. Ce type de contrôle sur la disposition des structs est utile quand vous utilisez des appels de code non managé.

Voir aussi

- System.Reflection
- Attribute
- Attributs

Collections

Article • 04/09/2023

Le runtime .NET fournit de nombreux types de collection qui stockent et gèrent des groupes d'objets associés. Certains types de collection, tels que System.Array, System.Span<T>et System.Memory<T> sont reconnus dans le langage C#. En outre, les interfaces telles que System.Collections.Generic.IEnumerable<T> sont reconnues dans le langage pour énumérer les éléments d'une collection.

Les collections offrent une façon flexible quand il s'agit d'utiliser des groupes d'objets. Vous pouvez classifier différentes collections selon ces caractéristiques :

- **Accès aux éléments** : chaque collection peut être énumérée pour accéder à chaque élément dans l'ordre. Certaines collections accèdent à des éléments par *index*, la position de l'élément dans une collection ordonnée. L'exemple le plus courant est System.Collections.Generic.List<T>. D'autres collections accèdent à des éléments par *clé*, où une *valeur* est associée à une *clé* unique. L'exemple le plus courant est System.Collections.Generic.Dictionary<TKey,TValue>. Vous choisissez entre ces types de collection en fonction de la façon dont votre application accède aux éléments.
- **Profil de niveau de performance** : chaque collection a des profils de niveau de performance différents pour les actions telles que l'ajout d'un élément, la recherche d'un élément ou la suppression d'un élément. Vous pouvez choisir un type de collection en fonction des opérations utilisées le plus dans votre application.
- **Développer et réduire dynamiquement** : la plupart des collections prenant en charge l'ajout ou la suppression d'éléments dynamiquement. Notamment, Array, System.Span<T>et System.Memory<T> ne le font pas.

En plus de ces caractéristiques, le runtime fournit des collections spécialisées qui empêchent d'ajouter ou de supprimer des éléments ou de modifier les éléments de la collection. D'autres collections spécialisées assurent la sécurité de l'accès simultané dans les applications multithreads.

Vous trouverez tous les types de collection dans la référence de l'API .NET . Pour plus d'informations, voir Types de collection couramment utilisés et Sélection d'une classe de collections.

> ⓘ **Notes**

Pour les exemples de cet article, vous devrez peut-être ajouter des **directives d'utilisation** pour les `System.Collections.Generic` et `System.Linq` espaces de noms.

Groupes sont représentées par System.Array et prennent en charge la syntaxe dans le langage C#. Cette syntaxe fournit des déclarations plus concises pour les variables de groupes.

System.Span<T> est un type ref struct qui fournit une capture instantanée sur une séquence d'éléments sans copier ces éléments. Le compilateur applique des règles de sécurité pour s'assurer que les `Span` ne sont plus accessibles une fois que la séquence qu'il référence n'est plus dans l'étendue. Il est utilisé dans de nombreuses API .NET pour améliorer le niveau de performance. Memory<T> fournit un comportement similaire lorsque vous ne pouvez pas utiliser de type `ref struct`.

À compter de C# 12, tous les types de collection peuvent être initialisés à l'aide d'une expression de collection.

Collections indexables

Une *collection indexable* est une collection où vous pouvez accéder à chaque élément à l'aide de son index. Son *index* est le nombre d'éléments avant celui-ci dans la séquence. Par conséquent, la référence d'élément par index `0` est le premier élément, l'index `1` est le deuxième, et ainsi de suite. Ces exemples utilisent la classe List<T>. Il s'agit de la collection indexable la plus courante.

L'exemple suivant crée et initialise une liste de chaînes, supprime un élément et ajoute un élément à la fin de la liste. Après chaque modification, itère au sein des chaînes à l'aide d'une instruction foreach foreach ou d'une boucle `for` :

C#

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook coho pink sockeye

// Remove an element from the list by specifying
// the object.
salmons.Remove("coho");
```

```
// Iterate using the index:
for (var index = 0; index < salmons.Count; index++)
{
    Console.Write(salmons[index] + " ");
}
// Output: chinook pink sockeye

// Add the removed element
salmons.Add("coho");
// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook pink sockeye coho
```

L'exemple suivant supprime les éléments d'une liste par index. À la place d'une instruction `foreach`, il utilise une instruction `for` qui itère dans l'ordre décroissant. La méthode RemoveAt, les éléments après l'élément supprimé ont une valeur d'index moins élevée.

C#

```
var numbers = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// Remove odd numbers.
for (var index = numbers.Count - 1; index >= 0; index--)
{
    if (numbers[index] % 2 == 1)
    {
        // Remove the element by specifying
        // the zero-based index in the list.
        numbers.RemoveAt(index);
    }
}

// Iterate through the list.
// A lambda expression is placed in the ForEach method
// of the List(T) object.
numbers.ForEach(
    number => Console.Write(number + " "));
// Output: 0 2 4 6 8
```

Pour le type d'éléments de List<T>, vous pouvez également définir votre propre classe. Dans l'exemple suivant, la classe `Galaxy` qui est utilisée par List<T> est définie dans le code.

C#

```
private static void IterateThroughList()
{
    var theGalaxies = new List<Galaxy>
    {
        new (){ Name="Tadpole", MegaLightYears=400},
        new (){ Name="Pinwheel", MegaLightYears=25},
        new (){ Name="Milky Way", MegaLightYears=0},
        new (){ Name="Andromeda", MegaLightYears=3}
    };

    foreach (Galaxy theGalaxy in theGalaxies)
    {
        Console.WriteLine(theGalaxy.Name + "  " + theGalaxy.MegaLightYears);
    }

    // Output:
    //  Tadpole  400
    //  Pinwheel  25
    //  Milky Way  0
    //  Andromeda  3
}

public class Galaxy
{
    public string Name { get; set; }
    public int MegaLightYears { get; set; }
}
```

Collections de paires clé/valeur

Ces exemples utilisent la classe Dictionary<TKey,TValue>. Il s'agit de la collection de dictionnaire la plus courante. Une collection de dictionnaires vous permet d'accéder aux éléments de la collection en utilisant la clé de chaque élément. Chaque ajout au dictionnaire se compose d'une valeur et de sa clé associée.

L'exemple suivant crée une collection `Dictionary` et itère au sein du dictionnaire à l'aide d'une instruction `foreach`.

C#

```
private static void IterateThruDictionary()
{
    Dictionary<string, Element> elements = BuildDictionary();

    foreach (KeyValuePair<string, Element> kvp in elements)
    {
        Element theElement = kvp.Value;

        Console.WriteLine("key: " + kvp.Key);
```

```
        Console.WriteLine("values: " + theElement.Symbol + " " +
            theElement.Name + " " + theElement.AtomicNumber);
    }
}

public class Element
{
    public required string Symbol { get; init; }
    public required string Name { get; init; }
    public required int AtomicNumber { get; init; }
}

private static Dictionary<string, Element> BuildDictionary() =>
    new ()
    {
        {"K",
            new (){ Symbol="K", Name="Potassium", AtomicNumber=19}},
        {"Ca",
            new (){ Symbol="Ca", Name="Calcium", AtomicNumber=20}},
        {"Sc",
            new (){ Symbol="Sc", Name="Scandium", AtomicNumber=21}},
        {"Ti",
            new (){ Symbol="Ti", Name="Titanium", AtomicNumber=22}}
    };
```

L'exemple suivant utilise la méthode ContainsKey et la propriété Item[] de `Dictionary` pour rechercher rapidement un élément par clé. En C#, la propriété `Item` vous permet d'accéder à un élément de la collection `elements` à l'aide d'`elements[symbol]`.

C#

```
if (elements.ContainsKey(symbol) == false)
{
    Console.WriteLine(symbol + " not found");
}
else
{
    Element theElement = elements[symbol];
    Console.WriteLine("found: " + theElement.Name);
}
```

L'exemple suivant utilise à la place la méthode TryGetValue pour rechercher rapidement un élément par clé.

C#

```
if (elements.TryGetValue(symbol, out Element? theElement) == false)
    Console.WriteLine(symbol + " not found");
```

```
else
    Console.WriteLine("found: " + theElement.Name);
```

Itérateurs

Un *itérateur* est utilisé pour exécuter une itération personnalisée sur une collection. Un itérateur peut être une méthode ou un accesseur `get`. Un itérateur utilise une instruction yield return pour retourner chaque élément de la collection un par un.

Vous appelez un itérateur en utilisant une instruction foreach. Chaque itération de la boucle `foreach` appelle l'itérateur. Quand une instruction `yield return` est atteinte dans l'itérateur, une expression est retournée et la localisation actuelle dans le code est retenue. L'exécution est redémarrée à partir de cet emplacement la prochaine fois que l'itérateur est appelé.

Pour plus d'informations, consultez Itérateurs (C#).

L'exemple suivant utilise une méthode Iterator. La méthode Iterator a une instruction `yield return` qui se trouve à l'intérieur d'une boucle `for`. Dans la méthode `ListEvenNumbers`, chaque itération du corps de l'instruction `foreach` crée un appel à la méthode Iterator, qui continue sur l'instruction `yield return` suivante.

C#

```
private static void ListEvenNumbers()
{
    foreach (int number in EvenSequence(5, 18))
    {
        Console.Write(number.ToString() + " ");
    }
    Console.WriteLine();
    // Output: 6 8 10 12 14 16 18
}

private static IEnumerable<int> EvenSequence(
    int firstNumber, int lastNumber)
{
    // Yield even numbers in the range.
    for (var number = firstNumber; number <= lastNumber; number++)
    {
        if (number % 2 == 0)
        {
            yield return number;
        }
    }
}
```

Language Integrated Query (LINQ) et collections

Language Integrated Query (LINQ) peut être utilisées pour accéder aux collections. Les requêtes LINQ fournissent des fonctionnalités de filtrage, de classement et de regroupement. Pour plus d'informations, consultez Bien démarrer avec LINQ en C#.

L'exemple suivant exécute une requête LINQ sur un `List` générique. La requête LINQ retourne une autre collection qui contient les résultats.

C#

```
private static void ShowLINQ()
{
    List<Element> elements = BuildList();

    // LINQ Query.
    var subset = from theElement in elements
                 where theElement.AtomicNumber < 22
                 orderby theElement.Name
                 select theElement;

    foreach (Element theElement in subset)
    {
        Console.WriteLine(theElement.Name + " " + theElement.AtomicNumber);
    }

    // Output:
    //  Calcium 20
    //  Potassium 19
    //  Scandium 21
}

private static List<Element> BuildList() => new()
    {
        { new(){ Symbol="K", Name="Potassium", AtomicNumber=19}},
        { new(){ Symbol="Ca", Name="Calcium", AtomicNumber=20}},
        { new(){ Symbol="Sc", Name="Scandium", AtomicNumber=21}},
        { new(){ Symbol="Ti", Name="Titanium", AtomicNumber=22}}
    };
```

Covariance et contravariance (C#)

Article • 25/02/2023

En C#, la covariance et la contravariance permettent la conversion de références implicite pour les types tableau, les types délégués et les arguments de type générique. La covariance conserve la compatibilité d'assignation et la contravariance l'inverse.

Le code suivant illustre la différence entre la compatibilité d'assignation, la covariance et la contravariance.

C#

```
// Assignment compatibility.
string str = "test";
// An object of a more derived type is assigned to an object of a less
derived type.
object obj = str;

// Covariance.
IEnumerable<string> strings = new List<string>();
// An object that is instantiated with a more derived type argument
// is assigned to an object instantiated with a less derived type argument.
// Assignment compatibility is preserved.
IEnumerable<object> objects = strings;

// Contravariance.
// Assume that the following method is in the class:
static void SetObject(object o) { }
Action<object> actObject = SetObject;
// An object that is instantiated with a less derived type argument
// is assigned to an object instantiated with a more derived type argument.
// Assignment compatibility is reversed.
Action<string> actString = actObject;
```

La covariance pour les tableaux permet la conversion implicite d'un tableau d'un type plus dérivé en un tableau d'un type moins dérivé. Cette opération n'est cependant pas sécurisée au niveau des types, comme le montre l'exemple de code suivant.

C#

```
object[] array = new String[10];
// The following statement produces a run-time exception.
// array[0] = 10;
```

La prise en charge de la covariance et la contravariance pour les groupes de méthodes permet la correspondance des signatures de méthode avec des types délégués. Ceci

vous permet d'affecter aux délégués non seulement les méthodes ayant des signatures correspondantes, mais aussi des méthodes qui retournent des types plus dérivés (covariance) ou qui acceptent des paramètres ayant des types moins dérivés (contravariance) que ceux spécifiés par le type délégué. Pour plus d'informations, consultez Variance dans les délégués (C#) et Utilisation de la variance dans les délégués (C#).

L'exemple de code suivant montre la prise en charge de la covariance et de la contravariance pour les groupes de méthode.

C#

```
static object GetObject() { return null; }
static void SetObject(object obj) { }

static string GetString() { return ""; }
static void SetString(string str) { }

static void Test()
{
    // Covariance. A delegate specifies a return type as object,
    // but you can assign a method that returns a string.
    Func<object> del = GetString;

    // Contravariance. A delegate specifies a parameter type as string,
    // but you can assign a method that takes an object.
    Action<string> del2 = SetObject;
}
```

Dans le NET Framework 4 ou les versions ultérieures, C# prend en charge la covariance et la contravariance dans les interfaces et les délégués génériques, et permet la conversion implicite de paramètres de type générique. Pour plus d'informations, consultez Variance dans les interfaces génériques (C#) et Variance dans les délégués (C#).

L'exemple de code suivant illustre la conversion de références implicite pour les interfaces génériques.

C#

```
IEnumerable<String> strings = new List<String>();
IEnumerable<Object> objects = strings;
```

Une interface ou un délégué générique est dit de type *variant* si ses paramètres génériques sont déclarés covariants ou contravariants. C# vous permet de créer vos

propres interfaces et délégués de type variant. Pour plus d'informations, consultez Création d'interfaces génériques de type variant (C#) et Variance dans les délégués (C#).

Rubriques connexes

Titre	Description
Variance dans les interfaces génériques (C#)	Décrit la covariance et la contravariance dans les interfaces génériques et fournit la liste des interfaces génériques de type variant dans .NET.
Création d'interfaces génériques de type variant (C#)	Montre comment créer des interfaces de type variant personnalisées.
Utilisation de la variance dans les interfaces pour les collections génériques (C#)	Montre comment la prise en charge de la covariance et de la contravariance dans les interfaces IEnumerable<T> et IComparable<T> peut vous aider à réutiliser du code.
Variance dans les délégués (C#)	Présente la covariance et la contravariance dans les délégués génériques et non génériques, et fournit une liste des délégués génériques de type variant dans .NET.
Utilisation de la variance dans les délégués (C#)	Montre comment utiliser la prise en charge de la covariance et de la contravariance dans les délégués non génériques pour faire correspondre des signatures de méthode avec des types délégués.
Utilisation de la variance pour les délégués génériques Func et Action (C#)	Montre comment la prise en charge de la covariance et de la contravariance dans les délégués `Func` et `Action` peut vous aider à réutiliser du code.

Variance dans les interfaces génériques (C#)

Article • 10/05/2023

.NET Framework 4 a introduit la prise en charge de la variance pour plusieurs interfaces génériques existantes. La prise en charge de la variance permet la conversion implicite des classes qui implémentent ces interfaces.

À partir de .NET Framework 4, les interfaces suivantes sont des variantes :

- IEnumerable<T> (T est covariant)
- IEnumerator<T> (T est covariant)
- IQueryable<T> (T est covariant)
- IGrouping<TKey,TElement> (`TKey` et `TElement` sont covariants)
- IComparer<T> (T est contravariant)
- IEqualityComparer<T> (T est contravariant)
- IComparable<T> (T est contravariant)

À compter de .NET Framework 4.5, les interfaces suivantes sont des variants :

- IReadOnlyList<T> (T est covariant)
- IReadOnlyCollection<T> (T est covariant)

La covariance permet à une méthode d'avoir un type de retour plus dérivé que celui défini par le paramètre de type générique de l'interface. Pour illustrer la fonctionnalité de covariance, considérez ces interfaces génériques : `IEnumerable<Object>` et `IEnumerable<String>`. L'interface `IEnumerable<String>` n'hérite pas de l'interface `IEnumerable<Object>`. Toutefois, le type `String` hérite du type `Object` et, dans certains cas, vous pouvez assigner des objets de ces interfaces de l'un à l'autre. Ce cas de figure est présenté dans l'exemple de code suivant :

C#

```
IEnumerable<String> strings = new List<String>();
IEnumerable<Object> objects = strings;
```

Dans les versions antérieures de .NET Framework, ce code provoque une erreur de compilation en C# et, si `Option Strict` est activé, en Visual Basic. Cependant, vous pouvez désormais utiliser `strings` au lieu d'`objects`, comme illustré dans l'exemple précédent, parce que l'interface IEnumerable<T> est covariante.

La contravariance permet à une méthode d'avoir des types d'argument moins dérivés que ceux spécifiés par le paramètre générique de l'interface. Pour illustrer la contravariance, supposez que vous avez créé une classe `BaseComparer` pour comparer des instances de la classe `BaseClass`. La classe `BaseComparer` implémente l'interface `IEqualityComparer<BaseClass>`. Étant donné que l'interface IEqualityComparer<T> est maintenant contravariante, vous pouvez utiliser `BaseComparer` pour comparer des instances des classes qui héritent de la classe `BaseClass`. Ce cas de figure est présenté dans l'exemple de code suivant :

C#

```
// Simple hierarchy of classes.
class BaseClass { }
class DerivedClass : BaseClass { }

// Comparer class.
class BaseComparer : IEqualityComparer<BaseClass>
{
    public int GetHashCode(BaseClass baseInstance)
    {
        return baseInstance.GetHashCode();
    }
    public bool Equals(BaseClass x, BaseClass y)
    {
        return x == y;
    }
}
class Program
{
    static void Test()
    {
        IEqualityComparer<BaseClass> baseComparer = new BaseComparer();

        // Implicit conversion of IEqualityComparer<BaseClass> to
        // IEqualityComparer<DerivedClass>.
        IEqualityComparer<DerivedClass> childComparer = baseComparer;
    }
}
```

Pour obtenir d'autres exemples, consultez Utilisation de la variance dans les interfaces pour les collections génériques (C#).

La variance dans les interfaces génériques est prise en charge uniquement pour les types référence. Les types valeur ne prennent pas en charge la variance. Par exemple, `IEnumerable<int>` ne peut pas être converti implicitement en `IEnumerable<object>`, car les entiers sont représentés par un type valeur.

C#

```
IEnumerable<int> integers = new List<int>();
// The following statement generates a compiler error,
// because int is a value type.
// IEnumerable<Object> objects = integers;
```

Il est également important de se souvenir que les classes qui implémentent des interfaces variantes sont toujours invariantes. Par exemple, même si List<T> implémente l'interface covariante IEnumerable<T>, vous ne pouvez pas convertir implicitement `List<String>` en `List<Object>`. En voici une illustration dans l'exemple de code suivant.

C#

```
// The following line generates a compiler error
// because classes are invariant.
// List<Object> list = new List<String>();

// You can use the interface object instead.
IEnumerable<Object> listObjects = new List<String>();
```

Voir aussi

- Utilisation de la variance dans les interfaces pour les collections génériques (C#)
- Création d'interfaces génériques de type variant (C#)
- Interfaces génériques
- Variance dans les délégués (C#)

Création d'interfaces génériques de type variant (C#)

Article • 10/05/2023

Vous pouvez déclarer des paramètres de type générique dans les interfaces comme covariant ou contravariant. La *covariance* permet aux méthodes d'interface d'avoir des types de retour plus dérivés que ceux définis par les paramètres de type générique. La *contravariance* permet aux méthodes d'interface d'avoir des types d'argument moins dérivés que ceux spécifiés par les paramètres génériques. Une interface générique ayant des paramètres de type générique covariant ou contravariant est appelée *variante*.

> ⓘ **Notes**
>
> .NET Framework 4 a introduit la prise en charge de la variance pour plusieurs interfaces génériques existantes. Pour obtenir la liste des interfaces de type variant dans .NET, consultez **Variance dans les interfaces génériques (C#)**.

Déclaration d'interfaces génériques de type variant

Vous pouvez déclarer des interfaces génériques de type variant à l'aide des mots clés `in` et `out` pour les paramètres de type générique.

> ⓘ **Important**
>
> En C#, les paramètres `ref`, `in` et `out` ne peuvent pas être de type variant. Les types valeur ne prennent pas non plus en charge la variance.

Vous pouvez déclarer un covariant de paramètre de type générique à l'aide du mot clé `out`. Le type covariant doit remplir les conditions suivantes :

- Le type est utilisé uniquement comme un type de retour de méthodes d'interface, et non pas comme un type d'arguments de méthode. L'exemple suivant en fournit une illustration dans laquelle le type `R` est déclaré covariant.

  ```
  C#
  ```

```
interface ICovariant<out R>
{
    R GetSomething();
    // The following statement generates a compiler error.
    // void SetSomething(R sampleArg);

}
```

Il existe une exception à cette règle. Si vous avez un délégué générique contravariant comme paramètre de méthode, vous pouvez utiliser le type comme paramètre de type générique pour le délégué. L'exemple suivant du type `R` illustre ce principe. Pour plus d'informations, consultez Variance dans les délégués (C#) et Utilisation de la variance pour les délégués génériques Func et Action (C#).

C#

```
interface ICovariant<out R>
{
    void DoSomething(Action<R> callback);
}
```

- Le type n'est pas utilisé comme contrainte générique pour les méthodes d'interface. C'est ce que montre le code suivant.

C#

```
interface ICovariant<out R>
{
    // The following statement generates a compiler error
    // because you can use only contravariant or invariant types
    // in generic constraints.
    // void DoSomething<T>() where T : R;
}
```

Vous pouvez déclarer un contravariant de paramètre de type générique à l'aide du mot clé `in`. Le type contravariant peut être utilisé uniquement comme type d'arguments de méthode, et non comme type de retour de méthodes d'interface. Le type contravariant peut également être utilisé pour les contraintes génériques. Le code suivant montre comment déclarer une interface de type contravariant et utiliser une contrainte générique pour l'une de ses méthodes.

C#

```
interface IContravariant<in A>
{
```

```
    void SetSomething(A sampleArg);
    void DoSomething<T>() where T : A;
    // The following statement generates a compiler error.
    // A GetSomething();
}
```

Il est également possible de prendre en charge à la fois la covariance et la contravariance dans la même interface, mais pour des paramètres de type différent, comme indiqué dans l'exemple de code suivant.

C#

```
interface IVariant<out R, in A>
{
    R GetSomething();
    void SetSomething(A sampleArg);
    R GetSetSomethings(A sampleArg);
}
```

Implémentation d'interfaces génériques de type variant

Vous implémentez des interfaces génériques de type variant dans les classes en utilisant la même syntaxe que celle utilisée pour les interfaces invariantes. L'exemple de code suivant montre comment implémenter une interface de type covariant dans une classe générique.

C#

```
interface ICovariant<out R>
{
    R GetSomething();
}
class SampleImplementation<R> : ICovariant<R>
{
    public R GetSomething()
    {
        // Some code.
        return default(R);
    }
}
```

Les classes qui implémentent des interfaces de type variant sont invariantes. Par exemple, prenons le code suivant.

```
```

C#

```
// The interface is covariant.
ICovariant<Button> ibutton = new SampleImplementation<Button>();
ICovariant<Object> iobj = ibutton;

// The class is invariant.
SampleImplementation<Button> button = new SampleImplementation<Button>();
// The following statement generates a compiler error
// because classes are invariant.
// SampleImplementation<Object> obj = button;
```

Extension d'interfaces génériques de type variant

Quand vous étendez une interface générique de type variant, vous devez utiliser les mots clés `in` et `out` pour spécifier explicitement si l'interface dérivée prend en charge la variance. Le compilateur ne déduit pas la variance de l'interface étendue. Observons, par exemple, les interfaces suivantes.

C#

```
interface ICovariant<out T> { }
interface IInvariant<T> : ICovariant<T> { }
interface IExtCovariant<out T> : ICovariant<T> { }
```

Dans l'interface `IInvariant<T>`, le paramètre de type générique `T` est invariant, alors que dans `IExtCovariant<out T>`, le paramètre de type est covariant, bien que les deux interfaces étendent la même interface. La même règle est appliquée aux paramètres de type générique contravariant.

Vous pouvez créer une interface qui étend à la fois l'interface dans laquelle le paramètre de type générique `T` est covariant et l'interface dans laquelle il est contravariant si, dans l'interface étendue, le paramètre de type générique `T` est invariant. En voici une illustration dans l'exemple de code suivant.

C#

```
interface ICovariant<out T> { }
interface IContravariant<in T> { }
interface IInvariant<T> : ICovariant<T>, IContravariant<T> { }
```

Toutefois, si un paramètre de type générique `T` est déclaré covariant dans une interface, vous ne pouvez pas le déclarer contravariant dans l'interface étendue, ou vice versa. En voici une illustration dans l'exemple de code suivant.

C#

```
interface ICovariant<out T> { }
// The following statement generates a compiler error.
// interface ICoContraVariant<in T> : ICovariant<T> { }
```

Éviter l'ambiguïté

Quand vous implémentez des interfaces génériques de type variant, la variance peut parfois mener à une certaine ambiguïté. Il convient d'éviter cette ambiguïté.

Par exemple, si vous implémentez explicitement la même interface générique de type variant avec des paramètres de type générique différents dans une classe, cela peut être source d'ambiguïté. Le compilateur ne génère pas d'erreur dans ce cas, mais l'implémentation d'interface qui sera choisie lors de l'exécution n'est pas spécifiée. Cette ambiguïté peut entraîner des bogues subtils dans votre code. Considérez l'exemple de code suivant.

C#

```
// Simple class hierarchy.
class Animal { }
class Cat : Animal { }
class Dog : Animal { }

// This class introduces ambiguity
// because IEnumerable<out T> is covariant.
class Pets : IEnumerable<Cat>, IEnumerable<Dog>
{
    IEnumerator<Cat> IEnumerable<Cat>.GetEnumerator()
    {
        Console.WriteLine("Cat");
        // Some code.
        return null;
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        // Some code.
        return null;
    }

    IEnumerator<Dog> IEnumerable<Dog>.GetEnumerator()
    {
```

```
        Console.WriteLine("Dog");
        // Some code.
        return null;
    }
}
class Program
{
    public static void Test()
    {
        IEnumerable<Animal> pets = new Pets();
        pets.GetEnumerator();
    }
}
```

Dans cet exemple, la façon dont la méthode `pets.GetEnumerator` choisit entre `Cat` et `Dog` n'est pas spécifiée. Cela peut provoquer des problèmes dans votre code.

Voir aussi

- Variance dans les interfaces génériques (C#)
- Utilisation de la variance pour les délégués génériques Func et Action (C#)

Utilisation de la variance dans les interfaces pour les collections génériques (C#)

Article • 10/05/2023

Une interface covariante permet à ses méthodes de retourner des types plus dérivés que ceux spécifiés dans l'interface. Une interface contravariante permet à ses méthodes d'accepter des paramètres de types moins dérivés que ceux spécifiés dans l'interface.

Dans .NET Framework 4, plusieurs interfaces existantes sont devenues covariantes et contravariantes. Celles-ci comprennent IEnumerable<T> et IComparable<T>. Cela vous permet de réutiliser des méthodes qui fonctionnent avec les collections génériques de types de base pour les collections de types dérivés.

Pour obtenir la liste des interfaces de type variant dans .NET, consultez Variance dans les interfaces génériques (C#).

Conversion de collections génériques

L'exemple suivant illustre les avantages de la prise en charge de la covariance dans l'interface IEnumerable<T>. La méthode `PrintFullName` accepte une collection de type `IEnumerable<Person>` comme paramètre. Toutefois, vous pouvez la réutiliser pour une collection de type `IEnumerable<Employee>`, car `Employee` hérite de `Person`.

C#

```
// Simple hierarchy of classes.
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Employee : Person { }

class Program
{
    // The method has a parameter of the IEnumerable<Person> type.
    public static void PrintFullName(IEnumerable<Person> persons)
    {
        foreach (Person person in persons)
        {
            Console.WriteLine("Name: {0} {1}",
```

```
            person.FirstName, person.LastName);
        }
    }

    public static void Test()
    {
        IEnumerable<Employee> employees = new List<Employee>();

        // You can pass IEnumerable<Employee>,
        // although the method expects IEnumerable<Person>.

        PrintFullName(employees);

    }
}
```

Comparaison de collections génériques

L'exemple suivant illustre les avantages de la prise en charge de la contravariance dans l'interface IEqualityComparer<T>. La classe `PersonComparer` implémente l'interface `IEqualityComparer<Person>`. Toutefois, vous pouvez réutiliser cette classe pour comparer une séquence d'objets de type `Employee`, car `Employee` hérite de `Person`.

C#

```
// Simple hierarchy of classes.
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Employee : Person { }

// The custom comparer for the Person type
// with standard implementations of Equals()
// and GetHashCode() methods.
class PersonComparer : IEqualityComparer<Person>
{
    public bool Equals(Person x, Person y)
    {
        if (Object.ReferenceEquals(x, y)) return true;
        if (Object.ReferenceEquals(x, null) ||
            Object.ReferenceEquals(y, null))
            return false;
        return x.FirstName == y.FirstName && x.LastName == y.LastName;
    }
    public int GetHashCode(Person person)
    {
        if (Object.ReferenceEquals(person, null)) return 0;
```

```
        int hashFirstName = person.FirstName == null
            ? 0 : person.FirstName.GetHashCode();
        int hashLastName = person.LastName.GetHashCode();
        return hashFirstName ^ hashLastName;
    }
}

class Program
{

    public static void Test()
    {
        List<Employee> employees = new List<Employee> {
               new Employee() {FirstName = "Michael", LastName =
"Alexander"},
               new Employee() {FirstName = "Jeff", LastName = "Price"}
            };

        // You can pass PersonComparer,
        // which implements IEqualityComparer<Person>,
        // although the method expects IEqualityComparer<Employee>.

        IEnumerable<Employee> noduplicates =
            employees.Distinct<Employee>(new PersonComparer());

        foreach (var employee in noduplicates)
            Console.WriteLine(employee.FirstName + " " + employee.LastName);
    }
}
```

Voir aussi

- Variance dans les interfaces génériques (C#)

Variance dans les délégués (C#)

Article • 09/05/2023

.NET Framework 3.5 a introduit la prise en charge de la variance pour faire correspondre les signatures de méthode aux types délégués pour tous les délégués dans C#. Cela signifie que vous pouvez assigner aux délégués non seulement les méthodes ayant des signatures correspondantes, mais également des méthodes qui retournent des types plus dérivés (covariance) ou qui acceptent des paramètres ayant des types moins dérivés (contravariance) que ceux spécifiés par le type délégué. Cela inclut à la fois des délégués génériques et non génériques.

Par exemple, considérez le code suivant, qui a deux classes et deux délégués : génériques et non génériques.

C#

```
public class First { }
public class Second : First { }
public delegate First SampleDelegate(Second a);
public delegate R SampleGenericDelegate<A, R>(A a);
```

Quand vous créez des délégués des types `SampleDelegate` ou `SampleGenericDelegate<A, R>`, vous pouvez assigner l'une des méthodes suivantes à ces délégués.

C#

```
// Matching signature.
public static First ASecondRFirst(Second second)
{ return new First(); }

// The return type is more derived.
public static Second ASecondRSecond(Second second)
{ return new Second(); }

// The argument type is less derived.
public static First AFirstRFirst(First first)
{ return new First(); }

// The return type is more derived
// and the argument type is less derived.
public static Second AFirstRSecond(First first)
{ return new Second(); }
```

L'exemple de code suivant illustre la conversion implicite entre la signature de méthode et le type délégué.

C#

```
// Assigning a method with a matching signature
// to a non-generic delegate. No conversion is necessary.
SampleDelegate dNonGeneric = ASecondRFirst;
// Assigning a method with a more derived return type
// and less derived argument type to a non-generic delegate.
// The implicit conversion is used.
SampleDelegate dNonGenericConversion = AFirstRSecond;

// Assigning a method with a matching signature to a generic delegate.
// No conversion is necessary.
SampleGenericDelegate<Second, First> dGeneric = ASecondRFirst;
// Assigning a method with a more derived return type
// and less derived argument type to a generic delegate.
// The implicit conversion is used.
SampleGenericDelegate<Second, First> dGenericConversion = AFirstRSecond;
```

Pour obtenir d'autres exemples, consultez Utilisation de la variance dans les délégués (C#) et Utilisation de la variance pour les délégués génériques Func et Action (C#).

Variance dans les paramètres de type générique

Dans .NET Framework 4 ou version ultérieure, vous pouvez activer la conversion implicite entre les délégués afin que les délégués génériques ayant des types différents spécifiés par les paramètres de type générique puissent être assignés les uns aux autres, si les types sont hérités les uns des autres comme requis par la variance.

Pour activer la conversion implicite, vous devez déclarer explicitement les paramètres génériques dans un délégué comme covariant ou contravariant à l'aide du mot clé `in` ou `out`.

L'exemple de code suivant indique comment vous pouvez créer un délégué ayant un paramètre de type générique covariant.

C#

```
// Type T is declared covariant by using the out keyword.
public delegate T SampleGenericDelegate <out T>();

public static void Test()
{
    SampleGenericDelegate <String> dString = () => " ";

    // You can assign delegates to each other,
    // because the type T is declared covariant.
```

```
    SampleGenericDelegate <Object> dObject = dString;
}
```

Si vous utilisez uniquement la prise en charge de la variance pour faire correspondre les signatures de méthode aux types délégués et que vous n'utilisez pas les mots clés `in` et `out`, vous pouvez réaliser qu'il est parfois possible d'instancier des délégués avec des expressions ou méthodes lambda identiques, mais que vous ne pouvez pas assigner un délégué à un autre.

Dans l'exemple de code suivant, `SampleGenericDelegate<String>` ne peut pas être converti explicitement en `SampleGenericDelegate<Object>`, même si `String` hérite de `Object`. Vous pouvez résoudre ce problème en marquant le paramètre générique `T` avec le mot clé `out`.

C#

```
public delegate T SampleGenericDelegate<T>();

public static void Test()
{
    SampleGenericDelegate<String> dString = () => " ";

    // You can assign the dObject delegate
    // to the same lambda expression as dString delegate
    // because of the variance support for
    // matching method signatures with delegate types.
    SampleGenericDelegate<Object> dObject = () => " ";

    // The following statement generates a compiler error
    // because the generic type T is not marked as covariant.
    // SampleGenericDelegate <Object> dObject = dString;

}
```

Délégués génériques ayant des paramètres de type variant dans .NET

.NET Framework 4 a introduit la prise en charge de la variance pour les paramètres de type générique dans plusieurs délégués génériques existants :

- Les délégués `Action` de l'espace de noms System, par exemple, Action<T> et Action<T1,T2>
- Les délégués `Func` de l'espace de noms System, par exemple, Func<TResult> et Func<T,TResult>

- Le délégué Predicate<T>
- Le délégué Comparison<T>
- Le délégué Converter<TInput,TOutput>

Pour obtenir plus d'informations et d'exemples, consultez Utilisation de la variance pour les délégués génériques Func et Action (C#).

Déclaration de paramètres de type variant dans les délégués génériques

Si un délégué générique possède des paramètres de type générique covariant ou contravariant, il peut être désigné sous le nom de *délégué générique variant*.

Vous pouvez déclarer un paramètre de type générique covariant dans un délégué générique à l'aide du mot clé `out`. Le type covariant peut être utilisé uniquement comme type de retour de méthode et non comme type d'arguments de méthode. L'exemple de code suivant indique comment déclarer un délégué générique covariant.

```
C#

public delegate R DCovariant<out R>();
```

Vous pouvez déclarer un paramètre de type générique contravariant dans un délégué générique à l'aide du mot clé `in`. Le type contravariant peut être utilisé uniquement comme type d'arguments de méthode et non comme type de retour de méthode. L'exemple de code suivant indique comment déclarer un délégué générique contravariant.

```
C#

public delegate void DContravariant<in A>(A a);
```

> **ⓘ Important**
>
> Les paramètres `ref`, `in` et `out` en C# ne peuvent pas être marqués comme étant variants.

Il est également possible de prendre en charge à la fois la variance et la covariance dans le même délégué, mais pour des paramètres de type différents. Cela est illustré par l'exemple suivant.

C#

```
public delegate R DVariant<in A, out R>(A a);
```

Instanciation et appel de délégués génériques variants

Vous pouvez instancier et appeler des délégués variants de la même façon que vous instanciez et appelez des délégués invariants. Dans l'exemple suivant, le délégué est instancié par une expression lambda.

C#

```
DVariant<String, String> dvariant = (String str) => str + " ";
dvariant("test");
```

Combinaison des délégués génériques variants

Ne combinez pas de délégués de type variant. La méthode Combine ne prend pas en charge la conversion des délégués variants et nécessite le même type pour tous les délégués. Il peut s'ensuivre une exception runtime quand vous combinez les délégués à l'aide de la méthode Combine ou de l'opérateur `+`, comme dans l'exemple de code suivant.

C#

```
Action<object> actObj = x => Console.WriteLine("object: {0}", x);
Action<string> actStr = x => Console.WriteLine("string: {0}", x);
// All of the following statements throw exceptions at run time.
// Action<string> actCombine = actStr + actObj;
// actStr += actObj;
// Delegate.Combine(actStr, actObj);
```

Variance dans les paramètres de type générique pour les types valeur et référence

La variance pour les paramètres de type générique est prise en charge uniquement pour les types référence. Par exemple, `DVariant<int>` ne peut pas être converti implicitement en `DVariant<Object>` ni `DVariant<long>` parce que l'entier est un type valeur.

L'exemple suivant montre que la variance dans les paramètres de type générique n'est pas prise en charge pour les types valeur.

C#

```
// The type T is covariant.
public delegate T DVariant<out T>();

// The type T is invariant.
public delegate T DInvariant<T>();

public static void Test()
{
    int i = 0;
    DInvariant<int> dInt = () => i;
    DVariant<int> dVariantInt = () => i;

    // All of the following statements generate a compiler error
    // because type variance in generic parameters is not supported
    // for value types, even if generic type parameters are declared
variant.
    // DInvariant<Object> dObject = dInt;
    // DInvariant<long> dLong = dInt;
    // DVariant<Object> dVariantObject = dVariantInt;
    // DVariant<long> dVariantLong = dVariantInt;
}
```

Voir aussi

- Génériques
- Utilisation de la variance pour les délégués génériques Func et Action (C#)
- Comment combiner des délégués (délégués multicast)

Utilisation de la variance dans les délégués (C#)

Article • 07/04/2023

Quand vous assignez une méthode à un délégué, la *covariance* et la *contravariance* offrent une grande flexibilité pour la mise en correspondance d'un type délégué avec une signature de méthode. La covariance permet à une méthode d'avoir un type de retour qui est plus dérivé que celui défini dans le délégué. La contravariance autorise une méthode qui a des types de paramètres moins dérivés que ceux du type délégué.

Exemple 1 : Covariance

Description

Cet exemple montre comment vous pouvez utiliser des délégués avec des méthodes ayant des types de retour dérivés du type de retour dans la signature du délégué. Le type de données retourné par `DogsHandler` est `Dogs`, qui dérive du type `Mammals` défini dans le délégué.

Code

C#

```
class Mammals {}
class Dogs : Mammals {}

class Program
{
    // Define the delegate.
    public delegate Mammals HandlerMethod();

    public static Mammals MammalsHandler()
    {
        return null;
    }

    public static Dogs DogsHandler()
    {
        return null;
    }

    static void Test()
    {
```

```
        HandlerMethod handlerMammals = MammalsHandler;

        // Covariance enables this assignment.
        HandlerMethod handlerDogs = DogsHandler;
    }
}
```

Exemple 2 : Contravariance

Description

Cet exemple montre comment vous pouvez utiliser des délégués avec des méthodes ayant des paramètres dont les types sont des types de base du type de paramètre de signature de délégué. Avec la contravariance, vous pouvez maintenant utiliser un gestionnaire d'événements plutôt que des gestionnaires distincts. L'exemple suivant utilise deux délégués :

- Un délégué KeyEventHandler qui définit la signature de l'événement Button.KeyDown. Sa signature est :

  C#

  ```
  public delegate void KeyEventHandler(object sender, KeyEventArgs e)
  ```

- Un délégué MouseEventHandler qui définit la signature de l'événement Button.MouseClick. Sa signature est :

  C#

  ```
  public delegate void MouseEventHandler(object sender, MouseEventArgs e)
  ```

L'exemple définit un gestionnaire d'événements avec un paramètre EventArgs et s'en sert pour gérer les événements `Button.KeyDown` et `Button.MouseClick`. C'est possible parce que EventArgs est un type de base de KeyEventArgs et MouseEventArgs.

Code

C#

```
// Event handler that accepts a parameter of the EventArgs type.
private void MultiHandler(object sender, System.EventArgs e)
{
    label1.Text = System.DateTime.Now.ToString();
```

```
}

public Form1()
{
    InitializeComponent();

    // You can use a method that has an EventArgs parameter,
    // although the event expects the KeyEventArgs parameter.
    this.button1.KeyDown += this.MultiHandler;

    // You can use the same method
    // for an event that expects the MouseEventArgs parameter.
    this.button1.MouseClick += this.MultiHandler;

}
```

Voir aussi

- Variance dans les délégués (C#)
- Utilisation de la variance pour les délégués génériques Func et Action (C#)

Utilisation de la variance pour les délégués génériques Func et Action (C#)

Article • 07/04/2023

Ces exemples montrent comment utiliser la covariance et la contravariance dans les délégués génériques `Func` et `Action` pour permettre de réutiliser des méthodes et fournir davantage de flexibilité dans votre code.

Pour plus d'informations sur la covariance et la contravariance, consultez Variance dans les délégués (C#).

Utilisation des délégués avec les paramètres de type covariant

L'exemple suivant illustre les avantages de la prise en charge de la covariance dans les délégués génériques `Func`. La méthode `FindByTitle` prend un paramètre de type `String` et retourne un objet de type `Employee`. Toutefois, vous pouvez assigner cette méthode au délégué `Func<String, Person>`, car `Employee` hérite de `Person`.

C#

```
// Simple hierarchy of classes.
public class Person { }
public class Employee : Person { }
class Program
{
    static Employee FindByTitle(String title)
    {
        // This is a stub for a method that returns
        // an employee that has the specified title.
        return new Employee();
    }

    static void Test()
    {
        // Create an instance of the delegate without using variance.
        Func<String, Employee> findEmployee = FindByTitle;

        // The delegate expects a method to return Person,
        // but you can assign it a method that returns Employee.
        Func<String, Person> findPerson = FindByTitle;

        // You can also assign a delegate
        // that returns a more derived type
        // to a delegate that returns a less derived type.
```

```
        findPerson = findEmployee;

    }
}
```

Utilisation des délégués avec les paramètres de type contravariant

L'exemple suivant illustre les avantages de la prise en charge de la contravariance dans les délégués génériques `Action`. La méthode `AddToContacts` prend un paramètre de type `Person`. Toutefois, vous pouvez assigner cette méthode au délégué `Action<Employee>`, car `Employee` hérite de `Person`.

C#

```
public class Person { }
public class Employee : Person { }
class Program
{
    static void AddToContacts(Person person)
    {
        // This method adds a Person object
        // to a contact list.
    }

    static void Test()
    {
        // Create an instance of the delegate without using variance.
        Action<Person> addPersonToContacts = AddToContacts;

        // The Action delegate expects
        // a method that has an Employee parameter,
        // but you can assign it a method that has a Person parameter
        // because Employee derives from Person.
        Action<Employee> addEmployeeToContacts = AddToContacts;

        // You can also assign a delegate
        // that accepts a less derived parameter to a delegate
        // that accepts a more derived parameter.
        addEmployeeToContacts = addPersonToContacts;
    }
}
```

Voir aussi

- Covariance et contravariance (C#)

- Génériques

Expression Trees

Article • 03/09/2023

Expression trees represent code in a tree-like data structure, where each node is an expression, for example, a method call or a binary operation such as `x < y`.

If you have used LINQ, you have experience with a rich library where the `Func` types are part of the API set. (If you aren't familiar with LINQ, you probably want to read the LINQ tutorial and the article about lambda expressions before this one.) Expression Trees provide richer interaction with the arguments that are functions.

You write function arguments, typically using Lambda Expressions, when you create LINQ queries. In a typical LINQ query, those function arguments are transformed into a delegate the compiler creates.

You've likely already written code that uses Expression trees. Entity Framework's LINQ APIs accept Expression trees as the arguments for the LINQ Query Expression Pattern. That enables Entity Framework to translate the query you wrote in C# into SQL that executes in the database engine. Another example is Moq, which is a popular mocking framework for .NET.

When you want to have a richer interaction, you need to use *Expression Trees*. Expression Trees represent code as a structure that you examine, modify, or execute. These tools give you the power to manipulate code during run time. You write code that examines running algorithms, or injects new capabilities. In more advanced scenarios, you modify running algorithms and even translate C# expressions into another form for execution in another environment.

You compile and run code represented by expression trees. Building and running expression trees enables dynamic modification of executable code, the execution of LINQ queries in various databases, and the creation of dynamic queries. For more information about expression trees in LINQ, see How to use expression trees to build dynamic queries.

Expression trees are also used in the dynamic language runtime (DLR) to provide interoperability between dynamic languages and .NET and to enable compiler writers to emit expression trees instead of Microsoft intermediate language (MSIL). For more information about the DLR, see Dynamic Language Runtime Overview.

You can have the C# or Visual Basic compiler create an expression tree for you based on an anonymous lambda expression, or you can create expression trees manually by using the System.Linq.Expressions namespace.

When a lambda expression is assigned to a variable of type Expression<TDelegate>, the compiler emits code to build an expression tree that represents the lambda expression.

The C# compiler generates expression trees only from expression lambdas (or single-line lambdas). It can't parse statement lambdas (or multi-line lambdas). For more information about lambda expressions in C#, see Lambda Expressions.

The following code examples demonstrate how to have the C# compiler create an expression tree that represents the lambda expression `num => num < 5`.

C#

```
Expression<Func<int, bool>> lambda = num => num < 5;
```

You create expression trees in your code. You build the tree by creating each node and attaching the nodes into a tree structure. You learn how to create expressions in the article on building expression trees.

Expression trees are immutable. If you want to modify an expression tree, you must construct a new expression tree by copying the existing one and replacing nodes in it. You use an expression tree visitor to traverse the existing expression tree. For more information, see the article on translating expression trees.

Once you build an expression tree, you execute the code represented by the expression tree.

Limitations

There are some newer C# language elements that don't translate well into expression trees. Expression trees can't contain `await` expressions, or `async` lambda expressions. Many of the features added in C# 6 and later don't appear exactly as written in expression trees. Instead, newer features are exposed in expression trees in the equivalent, earlier syntax, where possible. Other constructs aren't available. It means that code that interprets expression trees works the same when new language features are introduced. However, even with these limitations, expression trees do enable you to create dynamic algorithms that rely on interpreting and modifying code that is represented as a data structure. It enables rich libraries such as Entity Framework to accomplish what they do.

Expression trees won't support new expression node types. It would be a breaking change for all libraries interpreting expression trees to introduce new node types. The following list includes most C# language elements that can't be used:

- Conditional methods that have been removed
- base access
- Method group expressions, including *address-of* (&) a method group, and anonymous method expressions
- References to local functions
- Statements, including assignment (`=`) and statement bodied expressions
- Partial methods with only a defining declaration
- Unsafe pointer operations
- dynamic operations
- Coalescing operators with null or default literal left side, null coalescing assignment, and the null propagating operator (?.)
- Multi-dimensional array initializers, indexed properties, and dictionary initializers
- throw expressions
- Accessing static virtual or abstract interface members
- Lambda expressions that have attributes
- Interpolated strings
- UTF-8 string conversions or UTF-8 string literals
- Method invocations using variable arguments, named arguments or optional arguments
- Expressions using System.Index or System.Range, index "from end" (^) operator or range expressions (..)
- async lambda expressions or await expressions, including await foreach and await using
- Tuple literals, tuple conversions, tuple == or !=, or with expressions
- Discards (_), deconstructing assignment, pattern matching is operator or the pattern matching switch expression
- COM call with `ref` omitted on the arguments
- ref, in or out parameters, `ref` return values, `out` arguments, or any values of ref struct type

Exécuter des arborescences d'expressions

Article • 13/03/2023

Une *arborescence d'expressions* est une structure de données qui représente du code. Il ne s'agit pas de code compilé et exécutable. Si vous souhaitez exécuter le code .NET représenté par une arborescence d'expressions, vous devez le convertir en instructions de langage intermédiaire exécutables. L'exécution d'une arborescence d'expressions peut retourner une valeur, ou elle peut simplement effectuer une action telle que l'appel d'une méthode.

Seules les arborescences d'expressions qui représentent des expressions lambda peuvent être exécutées. Les arborescences d'expressions qui représentent des expressions lambda peuvent être de type LambdaExpression ou Expression<TDelegate>. Pour exécuter ces arborescences d'expressions, appelez la méthode Compile pour créer un délégué exécutable, puis appelez le délégué.

> ⓘ **Notes**
>
> Si le type du délégué n'est pas connu, autrement dit si l'expression lambda est de type **LambdaExpression** et non **Expression<TDelegate>**, appelez la méthode **DynamicInvoke** sur le délégué au lieu de l'appeler directement.

Si une arborescence d'expressions ne représente pas une expression lambda, vous pouvez créer une expression lambda ayant l'arborescence d'expressions d'origine comme corps, en appelant la méthode Lambda<TDelegate>(Expression, IEnumerable<ParameterExpression>). Ensuite, vous pouvez exécuter l'expression lambda comme décrit plus haut dans cette section.

Des expressions lambda aux fonctions

Vous pouvez convertir n'importe quelle LambdaExpression ou n'importe quel type dérivé de LambdaExpression en langage intermédiaire exécutable. Les autres types d'expressions ne peuvent pas être convertis directement en code. Cette restriction a peu d'effet dans la pratique. Les expressions lambda sont les seuls types d'expressions que vous pourriez souhaiter exécuter par l'intermédiaire d'une conversion en langage intermédiaire exécutable. (Réfléchissez à ce que signifie l'exécution directe d'une System.Linq.Expressions.ConstantExpression. Est-ce que cela veut dire quelque chose d'utile ?) Toute arborescence de l'expression qui est une

System.Linq.Expressions.LambdaExpression ou un type dérivé de `LambdaExpression` peut être convertie en langage intermédiaire (IL). Le type d'expression System.Linq.Expressions.Expression<TDelegate> est le seul exemple concret dans les bibliothèques .NET Core. Il sert à représenter une expression mappée à un type délégué quelconque. Ce type étant mappé à un type délégué, .NET peut examiner l'expression et générer le langage intermédiaire pour un délégué approprié qui correspond à la signature de l'expression lambda. Le type délégué est basé sur le type d'expression. Si vous souhaitez utiliser l'objet délégué d'une manière fortement typée, vous devez connaître le type de retour et la liste d'arguments. La méthode `LambdaExpression.Compile()` retourne le type `Delegate`. Vous devez effectuer un cast vers le type délégué approprié pour que les outils de compilation puissent vérifier la liste d'arguments de type de retour.

Dans la plupart des cas, un mappage simple entre une expression et son délégué correspondant existe. Par exemple, une arborescence d'expressions représentée par `Expression<Func<int>>` serait convertie en un délégué du type `Func<int>`. Pour une expression lambda avec tout type de retour et liste d'arguments, il existe un type délégué qui est le type cible pour le code exécutable représenté par cette expression lambda.

Le type System.Linq.Expressions.LambdaExpression contient des membres LambdaExpression.Compile et LambdaExpression.CompileToMethod que vous utiliseriez pour convertir une arborescence d'expressions en code exécutable. La méthode `Compile` crée un délégué. La méthode `CompileToMethod` met à jour un objet System.Reflection.Emit.MethodBuilder avec le langage intermédiaire qui représente la sortie compilée de l'arborescence d'expressions.

> **ⓘ Important**
>
> `CompileToMethod` est disponible uniquement dans .NET Framework, et non dans .NET Core ou .NET 5 et versions ultérieures.

Si vous le souhaitez, vous pouvez également fournir un System.Runtime.CompilerServices.DebugInfoGenerator qui reçoit les informations de débogage de symbole pour l'objet délégué généré. `DebugInfoGenerator` fournit des informations de débogage complètes sur le délégué généré.

Vous pouvez convertir une expression en délégué à l'aide du code suivant :

```
C#
```

```
Expression<Func<int>> add = () => 1 + 2;
var func = add.Compile(); // Create Delegate
var answer = func(); // Invoke Delegate
Console.WriteLine(answer);
```

L'exemple de code suivant illustre les types concrets utilisés lorsque vous compilez et exécutez une arborescence d'expression.

C#

```
Expression<Func<int, bool>> expr = num => num < 5;

// Compiling the expression tree into a delegate.
Func<int, bool> result = expr.Compile();

// Invoking the delegate and writing the result to the console.
Console.WriteLine(result(4));

// Prints True.

// You can also use simplified syntax
// to compile and run an expression tree.
// The following line can replace two previous statements.
Console.WriteLine(expr.Compile()(4));

// Also prints True.
```

L'exemple de code suivant montre comment exécuter une arborescence d'expressions qui représente l'élévation d'un nombre à une puissance en créant une expression lambda et en l'exécutant. Le résultat, qui représente le nombre élevé à la puissance, est affiché.

C#

```
// The expression tree to execute.
BinaryExpression be = Expression.Power(Expression.Constant(2d),
Expression.Constant(3d));

// Create a lambda expression.
Expression<Func<double>> le = Expression.Lambda<Func<double>>(be);

// Compile the lambda expression.
Func<double> compiledExpression = le.Compile();

// Execute the lambda expression.
double result = compiledExpression();

// Display the result.
Console.WriteLine(result);
```

```
// This code produces the following output:
// 8
```

Exécution et durées de vie

Vous exécutez le code en appelant le délégué créé quand vous avez appelé `LambdaExpression.Compile()`. Le code précédent, `add.Compile()`, retourne un délégué. Vous appelez ce délégué en appelant `func()`, qui exécute le code.

Ce délégué représente le code dans l'arborescence d'expressions. Vous pouvez conserver le handle de ce délégué et l'appeler ultérieurement. Vous n'avez pas besoin de compiler l'arborescence d'expressions chaque fois que vous souhaitez exécuter le code qu'elle représente. (N'oubliez pas que les arborescences d'expressions sont immuables ; compiler la même arborescence d'expressions ultérieurement crée un délégué qui exécute le même code.)

> **⊗ Attention**
>
> Évitez de créer des mécanismes de mise en cache plus complexes pour améliorer les performances en évitant les appels de compilation inutiles. Comparer deux arborescences d'expressions arbitraires pour déterminer si elles représentent le même algorithme est une opération qui prend du temps. Le temps de calcul économisé en évitant tout appel supplémentaire à `LambdaExpression.Compile()` sera probablement largement contrebalancé par le temps passé à exécuter le code qui détermine si deux arborescences d'expressions génèrent le même code exécutable.

Mises en garde

Compiler une expression lambda en un délégué et appeler ce délégué sont l'une des opérations les plus simples que vous puissiez effectuer avec une arborescence d'expressions. Toutefois, même avec cette simple opération, il existe des pièges dont vous devez être conscient.

Les expressions lambda créent des fermetures sur toutes les variables locales qui sont référencées dans l'expression. Vous devez vérifier que toutes les variables qui font partie du délégué sont utilisables à l'emplacement où vous appelez `Compile` et quand vous exécutez le délégué résultant. Le compilateur garantit que les variables sont dans l'étendue. Toutefois, si votre expression accède à une variable qui implémente

`IDisposable`, il est possible que votre code supprime l'objet alors qu'il est toujours détenu par l'arborescence d'expressions.

Par exemple, ce code fonctionne bien, car `int` n'implémente pas `IDisposable` :

C#

```
private static Func<int, int> CreateBoundFunc()
{
    var constant = 5; // constant is captured by the expression tree
    Expression<Func<int, int>> expression = (b) => constant + b;
    var rVal = expression.Compile();
    return rVal;
}
```

Le délégué a capturé une référence à la variable locale `constant`. Cette variable est accessible à tout moment ultérieurement, quand la fonction retournée par `CreateBoundFunc` s'exécute.

Maintenant, examinez la classe qui suit, qui implémente System.IDisposable :

C#

```
public class Resource : IDisposable
{
    private bool _isDisposed = false;
    public int Argument
    {
        get
        {
            if (!_isDisposed)
                return 5;
            else throw new ObjectDisposedException("Resource");
        }
    }

    public void Dispose()
    {
        _isDisposed = true;
    }
}
```

Si vous l'utilisez dans une expression comme illustré dans le code suivant, vous obtenez une System.ObjectDisposedException quand vous exécutez le code référencé par la propriété `Resource.Argument` :

C#

```
private static Func<int, int> CreateBoundResource()
{
    using (var constant = new Resource()) // constant is captured by the
expression tree
    {
        Expression<Func<int, int>> expression = (b) => constant.Argument +
b;
        var rVal = expression.Compile();
        return rVal;
    }
}
```

Le délégué retourné par cette méthode a fermé l'objet `constant`, qui a été supprimé. (Il a été supprimé car il a été déclaré dans une instruction `using`.)

Désormais, quand vous exécuterez le délégué retourné par cette méthode, une `ObjectDisposedException` est levée au moment de l'exécution.

Cela semble étrange d'avoir une erreur d'exécution qui représente une construction de compilation, mais cela fait partie des éventualités quand vous travaillez avec des arborescences d'expressions.

Comme il existe de nombreuses permutations de ce problème, il est difficile de proposer des conseils généraux pour l'éviter. Faites attention si vous accédez à des variables locales quand vous définissez des expressions et faites attention si vous accédez à l'état de l'objet actif (représenté par `this`) quand vous créez une arborescence d'expressions qui peut être retournée par une API publique.

Le code dans votre expression peut référencer des méthodes ou des propriétés dans d'autres assemblys. Ces assemblys doivent être accessibles quand l'expression est définie, quand elle est compilée et quand le délégué résultant est appelé. S'il n'est pas présent, vous recevrez une `ReferencedAssemblyNotFoundException`.

Résumé

Les arborescences d'expressions qui représentent des expressions lambda peuvent être compilées pour créer un délégué exécutable. Les arborescences d'expression fournissent un mécanisme pour exécuter le code représenté par une arborescence d'expressions.

L'arborescence d'expressions représente le code qui s'exécuterait pour toute construction donnée que vous créez. Tant que l'environnement dans lequel vous compilez et exécutez le code correspond à celui dans lequel vous créez une expression, tout fonctionne comme prévu. Si ce n'est pas le cas, les erreurs sont prévisibles et sont

interceptées lors de vos premiers tests de n'importe quel code utilisant les arborescences d'expressions.

Traduire des arborescences d'expressions

Article • 13/03/2023

Dans cet article, découvrez comment visiter chaque nœud dans une arborescence d'expressions lors de la génération d'une copie modifiée de cette arborescence d'expressions. Vous traduisez les arborescences d'expressions pour comprendre les algorithmes exprimés pour pouvoir traduire dans un autre environnement. Vous modifiez l'algorithme qui a été créé. Vous pouvez ajouter la journalisation, intercepter des appels de méthode et les suivre, entre autres.

Le code que vous générez pour traduire une arborescence d'expressions est une extension de ce que vous avez déjà vu pour visiter tous les nœuds dans une arborescence. Quand vous traduisez une arborescence d'expressions, vous visitez tous les nœuds et, ce faisant, vous générez la nouvelle arborescence. Celle-ci peut contenir des références aux nœuds d'origine ou de nouveaux nœuds que vous avez placés dans l'arborescence.

Nous allons visiter une arborescence d'expressions et en créant une nouvelle arborescence avec certains nœuds de substitution. Dans cet exemple, nous allons remplacer une constante par une autre 10 fois plus grande. Pour le reste, vous laisserez l'arborescence d'expressions intacte. Au lieu de lire la valeur de la constante et de la remplacer par une nouvelle constante, vous allez remplacer le nœud de constante par un nouveau nœud qui effectue la multiplication.

Ici, une fois que vous avez trouvé un nœud de constante, vous créez un nœud de multiplication dont les enfants sont la constante d'origine, et la constante `10` :

C#

```
private static Expression ReplaceNodes(Expression original)
{
    if (original.NodeType == ExpressionType.Constant)
    {
        return Expression.Multiply(original, Expression.Constant(10));
    }
    else if (original.NodeType == ExpressionType.Add)
    {
        var binaryExpression = (BinaryExpression)original;
        return Expression.Add(
            ReplaceNodes(binaryExpression.Left),
            ReplaceNodes(binaryExpression.Right));
    }
```

```
    return original;
}
```

Créez une arborescence en remplaçant le nœud d'origine par le substitut. Vous pouvez vérifier les changements en compilant et en exécutant l'arborescence remplacée.

C#

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var addition = Expression.Add(one, two);
var sum = ReplaceNodes(addition);
var executableFunc = Expression.Lambda(sum);

var func = (Func<int>)executableFunc.Compile();
var answer = func();
Console.WriteLine(answer);
```

Pour générer une nouvelle arborescence, il faut visiter les nœuds dans l'arborescence existante, créer de nouveaux nœuds et les insérer dans l'arborescence. L'exemple précédent montre l'importance du caractère immuable des arborescences d'expressions. Notez que la nouvelle arborescence créée dans le code précédent contient une combinaison de nœuds nouvellement créés et de nœuds de l'arborescence existante. Les nœuds peuvent être utilisés dans les deux arborescences, car les nœuds de l'arborescence existante ne peuvent pas être modifiés. La réutilisation des nœuds entraîne une amélioration significative de l'efficacité de la mémoire. Les mêmes nœuds sont utilisables dans toute une arborescence, ou dans plusieurs arborescences d'expressions. Les nœuds ne pouvant pas être modifiés, le même nœud peut être réutilisé chaque fois qu'il est nécessaire.

Parcours et exécution d'une addition

Nous allons vérifier cette nouvelle arborescence en générant un deuxième visiteur qui parcourt l'arborescence de nœuds d'addition et calcule le résultat. Apportez quelques modifications au visiteur que nous avons vu jusqu'ici. Dans cette nouvelle version, le visiteur retourne la somme partielle de l'opération d'addition jusqu'à ce point. Pour une expression constante, il s'agit simplement de la valeur de l'expression constante. Pour une expression d'addition, le résultat est la somme des opérandes gauche et droit, une fois ces arborescences parcourues.

C#

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
```

```
var three = Expression.Constant(3, typeof(int));
var four = Expression.Constant(4, typeof(int));
var addition = Expression.Add(one, two);
var add2 = Expression.Add(three, four);
var sum = Expression.Add(addition, add2);

// Declare the delegate, so you can call it
// from itself recursively:
Func<Expression, int> aggregate = null!;
// Aggregate, return constants, or the sum of the left and right operand.
// Major simplification: Assume every binary expression is an addition.
aggregate = (exp) =>
    exp.NodeType == ExpressionType.Constant ?
    (int)((ConstantExpression)exp).Value :
    aggregate(((BinaryExpression)exp).Left) +
aggregate(((BinaryExpression)exp).Right);

var theSum = aggregate(sum);
Console.WriteLine(theSum);
```

Il y a pas mal de code ici, mais les concepts sont abordables. Ce code visite les enfants avec une recherche en profondeur. Quand il rencontre un nœud de constante, le visiteur retourne la valeur de la constante. Une fois que le visiteur a visité les deux enfants, ceux-ci ont calculé la somme calculée pour cette sous-arborescence. Le nœud d'addition peut maintenant calculer sa somme. Une fois que tous les nœuds dans l'arborescence d'expressions ont été visités, la somme a été calculée. Vous pouvez suivre l'exécution en exécutant l'exemple dans le débogueur.

Nous allons simplifier le suivi de l'analyse des nœuds et du calcul de la somme en parcourant l'arborescence. Voici une version mise à jour de la méthode Aggregate qui comprend des informations de suivi :

C#

```
private static int Aggregate(Expression exp)
{
    if (exp.NodeType == ExpressionType.Constant)
    {
        var constantExp = (ConstantExpression)exp;
        Console.Error.WriteLine($"Found Constant: {constantExp.Value}");
        if (constantExp.Value is int value)
        {
            return value;
        }
        else
        {
            return 0;
        }
    }
    else if (exp.NodeType == ExpressionType.Add)
```

```
    {
        var addExp = (BinaryExpression)exp;
        Console.Error.WriteLine("Found Addition Expression");
        Console.Error.WriteLine("Computing Left node");
        var leftOperand = Aggregate(addExp.Left);
        Console.Error.WriteLine($"Left is: {leftOperand}");
        Console.Error.WriteLine("Computing Right node");
        var rightOperand = Aggregate(addExp.Right);
        Console.Error.WriteLine($"Right is: {rightOperand}");
        var sum = leftOperand + rightOperand;
        Console.Error.WriteLine($"Computed sum: {sum}");
        return sum;
    }
    else throw new NotSupportedException("Haven't written this yet");
}
```

Son exécution sur l'expression `sum` génère la sortie suivante :

Sortie

```
10
Found Addition Expression
Computing Left node
Found Addition Expression
Computing Left node
Found Constant: 1
Left is: 1
Computing Right node
Found Constant: 2
Right is: 2
Computed sum: 3
Left is: 3
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 3
Left is: 3
Computing Right node
Found Constant: 4
Right is: 4
Computed sum: 7
Right is: 7
Computed sum: 10
10
```

Suivez la sortie et suivez le code précédent. Vous devriez pouvoir déterminer comment le code visite chaque nœud et calcule la somme à mesure qu'il parcourt l'arborescence et trouve la somme.

À présent, jetons un œil à une autre exécution, avec l'expression donnée par `sum1` :

C#

```
Expression<Func<int>> sum1 = () => 1 + (2 + (3 + 4));
```

Voici la sortie de l'examen de cette expression :

Sortie

```
Found Addition Expression
Computing Left node
Found Constant: 1
Left is: 1
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 2
Left is: 2
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 3
Left is: 3
Computing Right node
Found Constant: 4
Right is: 4
Computed sum: 7
Right is: 7
Computed sum: 9
Right is: 9
Computed sum: 10
10
```

Bien que la réponse finale soit la même, le parcours d'arborescence est différent. Les nœuds sont parcourus dans un ordre différent, car l'arborescence a été construite avec différentes opérations survenant en premier.

Créer une copie modifiée

Créez un projet **Application console**. Ajoutez une directive `using` au fichier pour l'espace de noms `System.Linq.Expressions`. Ajoutez la classe `AndAlsoModifier` à votre projet.

C#

```
public class AndAlsoModifier : ExpressionVisitor
{
    public Expression Modify(Expression expression)
    {
```

```
        return Visit(expression);
    }

    protected override Expression VisitBinary(BinaryExpression b)
    {
        if (b.NodeType == ExpressionType.AndAlso)
        {
            Expression left = this.Visit(b.Left);
            Expression right = this.Visit(b.Right);

            // Make this binary expression an OrElse operation instead of an
AndAlso operation.
            return Expression.MakeBinary(ExpressionType.OrElse, left, right,
b.IsLiftedToNull, b.Method);
        }

        return base.VisitBinary(b);
    }
}
```

Cette classe hérite de la classe `AND` et est spécialisée pour modifier les expressions qui représentent des opérations ExpressionVisitor conditionnelles. Elle convertit ces opérations d'un `AND` conditionnel en un `OR` conditionnel. La classe substitue la méthode VisitBinary du type de base, car les expressions `AND` conditionnelles sont représentées sous forme d'expressions binaires. Dans la méthode `VisitBinary`, si l'expression passée représente une opération `AND` conditionnelle, le code construit une nouvelle expression qui contient l'opérateur `OR` conditionnel au lieu de l'opérateur `AND` conditionnel. Si l'expression passée à `VisitBinary` ne représente pas une opération `AND` conditionnelle, la méthode emploie l'implémentation de classe de base. Les méthodes de classe de base construisent des nœuds qui sont comme les arborescences d'expressions passées, mais les nœuds voient leurs sous-arborescences remplacées par les arborescences d'expressions produites de manière récursive par le visiteur.

Ajoutez une directive `using` au fichier pour l'espace de noms `System.Linq.Expressions`. Ajoutez le code à la méthode `Main` dans le fichier Program.cs pour créer une arborescence d'expressions, et passez-le à la méthode qui le modifie.

C#

```
Expression<Func<string, bool>> expr = name => name.Length > 10 &&
name.StartsWith("G");
Console.WriteLine(expr);

AndAlsoModifier treeModifier = new AndAlsoModifier();
Expression modifiedExpr = treeModifier.Modify((Expression)expr);

Console.WriteLine(modifiedExpr);
```

```
/*  This code produces the following output:

    name => ((name.Length > 10) && name.StartsWith("G"))
    name => ((name.Length > 10) || name.StartsWith("G"))
*/
```

Le code crée une expression qui contient une opération `AND` conditionnelle. Il crée ensuite une instance de la classe `AndAlsoModifier` et passe l'expression à la méthode `Modify` de cette classe. Les arborescences d'expressions d'origine et modifiée sont toutes deux générées pour montrer la modification. Compilez et exécutez l'application.

En savoir plus

Cet exemple montre un petit sous-ensemble du code que vous créeriez pour parcourir et interpréter les algorithmes représentés par une arborescence d'expressions. Pour des informations sur la génération d'une bibliothèque à usage général qui traduit des arborescences d'expressions dans un autre langage, consultez cette série de Matt Warren. Elle décrit en détail comment traduire le code que vous pourriez rencontrer dans une arborescence d'expressions.

Cet article aura mis en évidence toute la puissance des arborescences d'expressions. Vous examinez un ensemble de code, y apportez les modifications souhaitées et exécutez la version modifiée. Les arborescences d'expressions étant immuables, vous créez des arborescences à l'aide de composants d'arborescences existantes. La réutilisation des nœuds réduit la quantité de mémoire nécessaire pour créer des arborescences d'expressions.

Syntaxe de DebugView

Article • 13/03/2023

La propriété **DebugView** (disponible uniquement lors du débogage) fournit un rendu de la chaîne d'arborescences d'expression. La plupart de la syntaxe est assez simple à comprendre ; les cas spéciaux sont décrits dans les sections suivantes.

Chaque exemple est suivi d'un commentaire en bloc contenant **DebugView**.

ParameterExpression

Les noms de variables ParameterExpression s'affichent avec le symbole `$` au début.

Si un paramètre n'a pas de nom, un nom généré automatiquement lui est assigné, tel que `$var1` ou `$var2`.

C#

```
ParameterExpression numParam =  Expression.Parameter(typeof(int), "num");
/*
    $num
*/

ParameterExpression numParam =  Expression.Parameter(typeof(int));
/*
    $var1
*/
```

ConstantExpression

Pour les objets ConstantExpression qui représentent des valeurs entières, des chaînes et des valeurs `null`, la valeur de la constante est affichée.

Pour les types numériques comportant des suffixes standard comme littéraux C#, le suffixe est ajouté à la valeur. Le tableau suivant indique les suffixes associés aux différents types numériques.

Type	Mot clé	Suffixe
System.UInt32	uint	U
System.Int64	long	L

Type	Mot clé	Suffixe
System.UInt64	ulong	UL
System.Double	double	D
System.Single	float	F
System.Decimal	decimal	M

C#

```
int num = 10;
ConstantExpression expr = Expression.Constant(num);
/*
    10
*/

double num = 10;
ConstantExpression expr = Expression.Constant(num);
/*
    10D
*/
```

BlockExpression

Si le type d'un objet BlockExpression diffère du type de la dernière expression du bloc, le type est affiché entre crochets pointus (`<` et `>`). Sinon, le type de l'objet BlockExpression n'est pas affiché.

C#

```
BlockExpression block = Expression.Block(Expression.Constant("test"));
/*
    .Block() {
        "test"
    }
*/

BlockExpression block =  Expression.Block(typeof(Object),
Expression.Constant("test"));
/*
    .Block<System.Object>() {
        "test"
    }
*/
```

LambdaExpression

Les objets LambdaExpression sont affichés avec leurs types délégués.

Si une expression lambda n'a pas de nom, un nom généré automatiquement lui est assigné, tel que `#Lambda1` ou `#Lambda2`.

C#

```
LambdaExpression lambda =  Expression.Lambda<Func<int>>
(Expression.Constant(1));
/*
    .Lambda #Lambda1<System.Func'1[System.Int32]>() {
        1
    }
*/

LambdaExpression lambda =  Expression.Lambda<Func<int>>
(Expression.Constant(1), "SampleLambda", null);
/*
    .Lambda #SampleLambda<System.Func'1[System.Int32]>() {
        1
    }
*/
```

LabelExpression

Si vous spécifiez une valeur par défaut pour l'objet LabelExpression, cette valeur est affichée avant l'objet LabelTarget.

Le jeton `.Label` indique le début de l'étiquette. Le jeton `.LabelTarget` indique la destination de la cible à laquelle accéder.

Si une étiquette n'a pas de nom, un nom généré automatiquement lui est assigné, tel que `#Label1` ou `#Label2`.

C#

```
LabelTarget target = Expression.Label(typeof(int), "SampleLabel");
BlockExpression block = Expression.Block(
    Expression.Goto(target, Expression.Constant(0)),
    Expression.Label(target, Expression.Constant(-1))
);
/*
    .Block() {
        .Goto SampleLabel { 0 };
        .Label
            -1
```

```
        .LabelTarget SampleLabel:
    }
*/

LabelTarget target = Expression.Label();
BlockExpression block = Expression.Block(
    Expression.Goto(target),
    Expression.Label(target)
);
/*
    .Block() {
        .Goto #Label1 { };
        .Label
        .LabelTarget #Label1:
    }
*/
```

Opérateurs Checked

Des opérateurs activés sont affichés avec le symbole `#` en regard de l'opérateur. Par exemple, l'opérateur d'addition checked est affiché sous la forme `#+`.

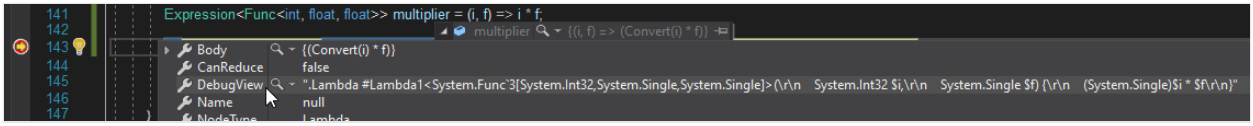
C#

```
Expression expr = Expression.AddChecked( Expression.Constant(1),
Expression.Constant(2));
/*
    1 #+ 2
*/

Expression expr = Expression.ConvertChecked( Expression.Constant(10.0),
typeof(int));
/*
    #(System.Int32)10D
*/
```

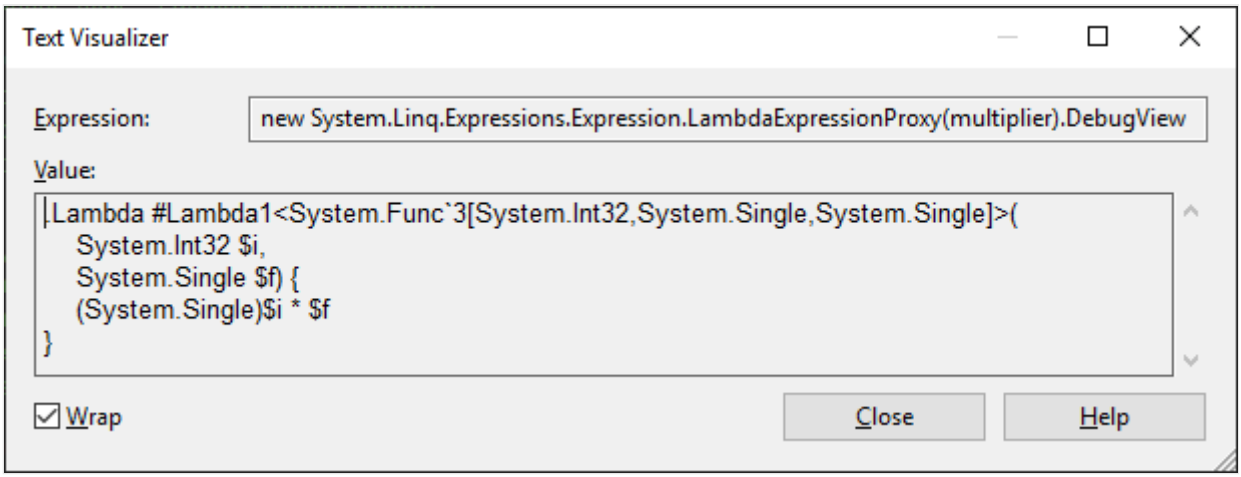
Débogage d'arborescences d'expressions dans Visual Studio

Article • 13/03/2023

Vous pouvez analyser la structure et le contenu d'arborescences d'expression quand vous déboguez vos applications. Pour obtenir un rapide aperçu de l'arborescence d'expression, vous pouvez utiliser la propriété `DebugView`, qui représente des arborescences d'expression en utilisant une syntaxe spéciale. `DebugView` est disponible uniquement en mode débogage.

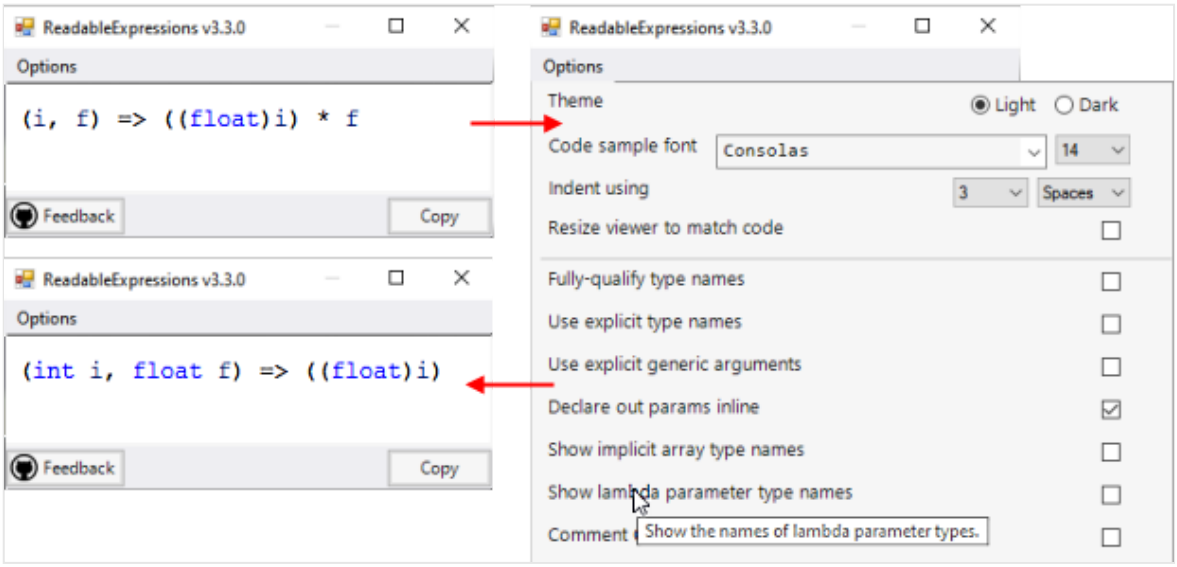


Étant donné que `DebugView` est une chaîne, vous pouvez utiliser le visualiseur de texte intégré pour l'afficher sur plusieurs lignes, en sélectionnant **Visualiseur de texte** à partir de l'icône de loupe située à côté de l'étiquette `DebugView`.

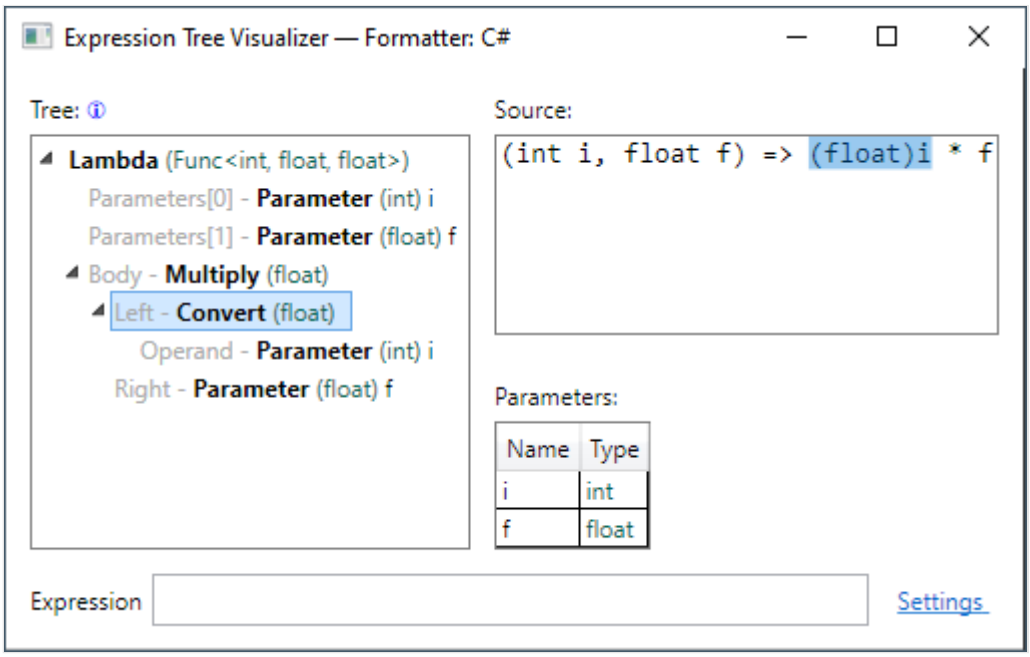


Vous pouvez également installer et utiliser un visualiseur personnalisé pour les arborescences d'expression, notamment :

- Readable Expressions (licence MIT disponible sur Visual Studio Marketplace), restitue l'arborescence d'expression sous forme de code C# pouvant avoir des thèmes, avec diverses options de rendu :

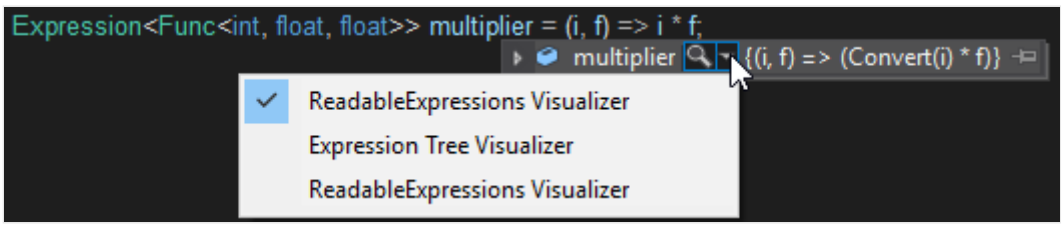


- Le visualiseur Expression Tree (licence MIT) offre une vue de l'arborescence d'expression et de ses nœuds individuels :



Ouvrir un visualiseur pour une arborescence d'expressions

Sélectionnez l'icône de loupe qui s'affiche en regard de l'arborescence d'expression dans **DataTips**, dans une fenêtre **Espion**, dans la fenêtre **Automatique** ou dans la fenêtre **Variables locales**. Une liste de visualiseurs disponibles s'affiche :



Sélectionnez le visualiseur à utiliser.

Voir aussi

- Débogage dans Visual Studio
- Créer des visualiseurs personnalisés
- DebugView syntax

Requêtes en fonction de l'état du runtime (C#)

Article • 13/03/2023

> ⓘ **Notes**
>
> Veillez à ajouter `using System.Linq.Expressions;` et `using static System.Linq.Expressions.Expression;` en haut de votre fichier *.cs*.

Voici le code qui définit une IQueryableIQueryable<T> sur une source de données :

C#

```
var companyNames = new[] {
    "Consolidated Messenger", "Alpine Ski House", "Southridge Video",
    "City Power & Light", "Coho Winery", "Wide World Importers",
    "Graphic Design Institute", "Adventure Works", "Humongous Insurance",
    "Woodgrove Bank", "Margie's Travel", "Northwind Traders",
    "Blue Yonder Airlines", "Trey Research", "The Phone Company",
    "Wingtip Toys", "Lucerne Publishing", "Fourth Coffee"
};

// We're using an in-memory array as the data source, but the IQueryable
could have come
// from anywhere -- an ORM backed by a database, a web request, or any other
LINQ provider.
IQueryable<string> companyNamesSource = companyNames.AsQueryable();
var fixedQry = companyNames.OrderBy(x => x);
```

Chaque fois que vous exécutez ce code, exactement la même requête est exécutée. Ce n'est généralement pas très utile, car vous voudrez peut-être que votre code exécute différentes requêtes en fonction des conditions au moment de l'exécution. Cet article explique comment exécuter une autre requête en fonction de l'état du runtime.

IQueryable / IQueryable<T> et arborescences d'expression

Un IQueryable a deux composants essentiels :

- Expression — représentation, indépendante du langage et des sources de données, des composants de la requête actuelle, sous la forme d'une arborescence d'expression.

- Provider — instance d'un fournisseur LINQ, qui sait comment matérialiser la requête actuelle en une valeur ou un ensemble de valeurs.

Dans le contexte de requêtes dynamiques, le fournisseur reste généralement le même tandis que l'arborescence d'expression de la requête diffère d'une requête à l'autre.

Les arborescences d'expression sont immuables. Si vous souhaitez une autre arborescence d'expression, et donc une autre requête, vous devez traduire l'arborescence d'expression existante en une nouvelle, et donc en une nouvelle IQueryable.

Les sections suivantes décrivent des techniques spécifiques de requête différentes en réponse à l'état du runtime :

- Utiliser l'état du runtime à partir de l'arborescence d'expression
- Appeler des méthodes LINQ supplémentaires
- Changer d'arborescence d'expression passée dans les méthodes LINQ
- Construire une arborescence d'expression Expression<TDelegate> avec les méthodes de fabrique sur Expression
- Ajouter des nœuds d'appel de méthode à l'arborescence d'expression d'une IQueryable
- Construire des chaînes et utiliser la bibliothèque Dynamic LINQ

Utiliser l'état du runtime à partir de l'arborescence d'expression

En supposant que le fournisseur LINQ le prend en charge, le moyen le plus simple d'interroger dynamiquement consiste à référencer l'état du runtime directement dans la requête via une variable « closed-over », comme `length` dans l'exemple de code suivant :

C#

```
var length = 1;
var qry = companyNamesSource
    .Select(x => x.Substring(0, length))
    .Distinct();

Console.WriteLine(string.Join(",", qry));
// prints: C, A, S, W, G, H, M, N, B, T, L, F

length = 2;
Console.WriteLine(string.Join(",", qry));
// prints: Co, Al, So, Ci, Wi, Gr, Ad, Hu, Wo, Ma, No, Bl, Tr, Th, Lu, Fo
```

L'arborescence d'expression interne, et donc la requête, n'ont pas été modifiées. La requête retourne des valeurs différentes uniquement parce que la valeur de `length` a été modifiée.

Appeler des méthodes LINQ supplémentaires

En règle générale, les méthodes LINQ intégrées sur Queryable effectuent deux étapes :

- Wrapper l'arborescence d'expression actuelle dans une MethodCallExpression représentant l'appel de méthode.
- Passer l'arborescence d'expression wrappée au fournisseur, soit pour retourner une valeur via la méthode IQueryProvider.Execute du fournisseur, soit pour retourner un objet de requête traduit via la méthode IQueryProvider.CreateQuery.

Vous pouvez remplacer la requête d'origine par le résultat d'une méthode retournant IQueryable<T> pour obtenir une nouvelle requête. Vous pouvez faire cela sous conditions en fonction de l'état du runtime, comme dans l'exemple suivant :

C#

```
// bool sortByLength = /* ... */;

var qry = companyNamesSource;
if (sortByLength)
{
    qry = qry.OrderBy(x => x.Length);
}
```

Changer d'arborescence d'expression passée dans les méthodes LINQ

Vous pouvez passer différentes expressions aux méthodes LINQ, en fonction de l'état du runtime :

C#

```
// string? startsWith = /* ... */;
// string? endsWith = /* ... */;

Expression<Func<string, bool>> expr = (startsWith, endsWith) switch
{
    ("" or null, "" or null) => x => true,
    (_, "" or null) => x => x.StartsWith(startsWith),
```

```
    ("" or null, _) => x => x.EndsWith(endsWith),
    (_, _) => x => x.StartsWith(startsWith) || x.EndsWith(endsWith)
};

var qry = companyNamesSource.Where(expr);
```

Vous voudrez probablement aussi composer les différentes sous-expressions à l'aide d'une bibliothèque tierce telle que celle de LinqKit, PredicateBuilder :

C#

```
// This is functionally equivalent to the previous example.

// using LinqKit;
// string? startsWith = /* ... */;
// string? endsWith = /* ... */;

Expression<Func<string, bool>>? expr = PredicateBuilder.New<string>(false);
var original = expr;
if (!string.IsNullOrEmpty(startsWith))
{
    expr = expr.Or(x => x.StartsWith(startsWith));
}
if (!string.IsNullOrEmpty(endsWith))
{
    expr = expr.Or(x => x.EndsWith(endsWith));
}
if (expr == original)
{
    expr = x => true;
}

var qry = companyNamesSource.Where(expr);
```

Construire des arborescences d'expression et des requêtes avec des méthodes de fabrique

Dans tous les exemples vus jusqu'ici, nous connaissions le type d'élément au moment de la compilation, `string`, et donc le type de la requête, `IQueryable<string>`. Vous devrez peut-être ajouter des composants à une requête de n'importe quel type d'élément ou ajouter des composants différents, en fonction du type d'élément. Vous pouvez créer des arborescences d'expression de la base jusqu'en haut en utilisant les méthodes de fabrique sur System.Linq.Expressions.Expression, et ainsi adapter l'expression au moment de l'exécution à un type d'élément spécifique.

Construction d'une Expression<TDelegate>

Lorsque vous construisez une expression à passer à l'une des méthodes LINQ, vous construisez en fait une instance d'Expression<TDelegate>, où `TDelegate` est un type délégué tel que `Func<string, bool>`, `Action` ou un type délégué personnalisé.

Expression<TDelegate> hérite de LambdaExpression, qui représente une expression lambda complète comme la suivante :

```
C#
```

```
Expression<Func<string, bool>> expr = x => x.StartsWith("a");
```

Une LambdaExpression a deux composants :

- Une liste de paramètres — `(string x)` — représentée par la propriété Parameters.
- Un corps — `x.StartsWith("a")` — représenté par la propriété Body.

Les étapes de base de la construction d'une Expression<TDelegate> sont les suivantes :

- Définir des objets ParameterExpression pour chacun des paramètres (le cas échéant) dans l'expression lambda, en utilisant la méthode de fabrique Parameter.

  ```
  C#
  ```

  ```
  ParameterExpression x = Expression.Parameter(typeof(string), "x");
  ```

- Construire le corps de votre LambdaExpression en utilisant la ou les ParameterExpressions que vous avez définies et les méthodes de fabrique sur Expression. Par exemple, une expression représentant `x.StartsWith("a")` peut être construite comme ceci :

  ```
  C#
  ```

  ```
  Expression body = Call(
      x,
      typeof(string).GetMethod("StartsWith", new[] { typeof(string) })!,
      Constant("a")
  );
  ```

- Wrapper les paramètres et le corps dans une Expression<TDelegate> typée au moment de la compilation, en utilisant la surcharge de méthode de fabrique Lambda appropriée :

  ```
  C#
  ```

```
Expression<Func<string, bool>> expr = Lambda<Func<string, bool>>(body,
x);
```

Les sections suivantes décrivent un scénario dans lequel vous voudrez probablement construire une Expression<TDelegate> à passer à une méthode LINQ, et fournissent un exemple complet de la façon de procéder à l'aide des méthodes de fabrique.

Scénario

Supposons que vous avez plusieurs types d'entités :

C#

```
record Person(string LastName, string FirstName, DateTime DateOfBirth);
record Car(string Model, int Year);
```

Pour l'un de ces types d'entités, vous souhaitez filtrer et retourner uniquement les entités qui ont un texte donné dans l'un de leurs champs `string`. Pour `Person`, vous souhaitez rechercher dans les propriétés `FirstName` et `LastName` :

C#

```
string term = /* ... */;
var personsQry = new List<Person>()
    .AsQueryable()
    .Where(x => x.FirstName.Contains(term) || x.LastName.Contains(term));
```

Par contre, pour `Car`, vous souhaitez rechercher uniquement dans la propriété `Model` :

C#

```
string term = /* ... */;
var carsQry = new List<Car>()
    .AsQueryable()
    .Where(x => x.Model.Contains(term));
```

Bien que vous puissiez écrire une fonction personnalisée pour `IQueryable<Person>` et une autre pour `IQueryable<Car>`, la fonction suivante ajoute ce filtrage à n'importe quelle requête existante, quel que soit le type d'élément spécifique.

Exemple

C#

```
// using static System.Linq.Expressions.Expression;

IQueryable<T> TextFilter<T>(IQueryable<T> source, string term)
{
    if (string.IsNullOrEmpty(term)) { return source; }

    // T is a compile-time placeholder for the element type of the query.
    Type elementType = typeof(T);

    // Get all the string properties on this specific type.
    PropertyInfo[] stringProperties =
        elementType.GetProperties()
            .Where(x => x.PropertyType == typeof(string))
            .ToArray();
    if (!stringProperties.Any()) { return source; }

    // Get the right overload of String.Contains
    MethodInfo containsMethod = typeof(string).GetMethod("Contains", new[] {
typeof(string) })!;

    // Create a parameter for the expression tree:
    // the 'x' in 'x => x.PropertyName.Contains("term")'
    // The type of this parameter is the query's element type
    ParameterExpression prm = Parameter(elementType);

    // Map each property to an expression tree node
    IEnumerable<Expression> expressions = stringProperties
        .Select(prp =>
            // For each property, we have to construct an expression tree
node like x.PropertyName.Contains("term")
            Call(                   // .Contains(...)
                Property(           // .PropertyName
                    prm,            // x
                    prp
                ),
                containsMethod,
                Constant(term)      // "term"
            )
        );

    // Combine all the resultant expression nodes using ||
    Expression body = expressions
        .Aggregate(
            (prev, current) => Or(prev, current)
        );

    // Wrap the expression body in a compile-time-typed lambda expression
    Expression<Func<T, bool>> lambda = Lambda<Func<T, bool>>(body, prm);

    // Because the lambda is compile-time-typed (albeit with a generic
parameter), we can use it with the Where method
```

```
    return source.Where(lambda);
}
```

Étant donné que la fonction `TextFilter` prend et retourne une IQueryable<T> (et pas juste une IQueryable), vous pouvez ajouter d'autres éléments de requête typés au moment dela compilation après le filtre de texte.

C#

```
var qry = TextFilter(
        new List<Person>().AsQueryable(),
        "abcd"
    )
    .Where(x => x.DateOfBirth < new DateTime(2001, 1, 1));

var qry1 = TextFilter(
        new List<Car>().AsQueryable(),
        "abcd"
    )
    .Where(x => x.Year == 2010);
```

Ajouter des nœuds d'appel de méthode à l'arborescence d'expression d'une IQueryable

Si vous avez une IQueryable au lieu d'une IQueryable<T>, vous ne pouvez pas appeler directement les méthodes LINQ génériques. Une alternative consiste à créer l'arborescence d'expression interne comme ci-dessus et à utiliser la réflexion pour appeler la méthode LINQ appropriée lors du passage de l'arborescence d'expression.

Vous pouvez également dupliquer les fonctionnalités de la méthode LINQ, en wrappant l'arborescence entière dans une MethodCallExpression qui représente un appel à la méthode LINQ :

C#

```
IQueryable TextFilter_Untyped(IQueryable source, string term)
{
    if (string.IsNullOrEmpty(term)) { return source; }
    Type elementType = source.ElementType;

    // The logic for building the ParameterExpression and the
LambdaExpression's body is the same as in the previous example,
    // but has been refactored into the constructBody function.
    (Expression? body, ParameterExpression? prm) =
constructBody(elementType, term);
    if (body is null) {return source;}
```

```
    Expression filteredTree = Call(
        typeof(Queryable),
        "Where",
        new[] { elementType},
        source.Expression,
        Lambda(body, prm!)
    );

    return source.Provider.CreateQuery(filteredTree);
}
```

Dans ce cas, vous n'avez pas d'espace réservé générique `T` au moment de la compilation, donc vous allez utiliser la surcharge de Lambda qui ne nécessite pas d'informations de type au moment de la compilation et qui produit une LambdaExpression au lieu d'une Expression<TDelegate>.

La bibliothèque Dynamic LINQ

La construction d'arborescences d'expression à l'aide de méthodes de fabrique est relativement complexe, il est plus facile de composer des chaînes. La bibliothèque Dynamic LINQ expose un ensemble de méthodes d'extension sur IQueryable correspondant aux méthodes LINQ standard sur Queryable, et qui acceptent les chaînes dans une syntaxe spéciale au lieu d'arborescences d'expression. La bibliothèque génère l'arborescence d'expression appropriée à partir de la chaîne et peut retourner la IQueryable traduite qui en résulte.

Par exemple, l'exemple précédent pourrait être réécrit comme ceci :

C#

```
// using System.Linq.Dynamic.Core

IQueryable TextFilter_Strings(IQueryable source, string term) {
    if (string.IsNullOrEmpty(term)) { return source; }

    var elementType = source.ElementType;

    // Get all the string property names on this specific type.
    var stringProperties =
        elementType.GetProperties()
            .Where(x => x.PropertyType == typeof(string))
            .ToArray();
    if (!stringProperties.Any()) { return source; }

    // Build the string expression
    string filterExpr = string.Join(
        " || ",
```

```
            stringProperties.Select(prp => $"{prp.Name}.Contains(@0)")
        );

        return source.Where(filterExpr, term);
    }
```

Voir aussi

- Exécuter des arborescences d'expressions
- Spécifier dynamiquement des filtres de prédicat au moment de l'exécution

Itérateurs (C#)

Article • 09/05/2023

Un *itérateur* peut être utilisé pour parcourir des collections, comme des listes et des tableaux.

Une méthode d'itérateur ou un accesseur `get` effectue une itération personnalisée sur une collection. Une méthode d'itérateur utilise l'instruction yield return pour retourner chaque élément un par un. Quand une instruction `yield return` est atteinte, l'emplacement actuel dans le code est mémorisé. L'exécution est redémarrée à partir de cet emplacement lors de l'appel suivant de la fonction d'itérateur.

Vous consommez un itérateur à partir du code client en utilisant une instruction foreach ou une requête LINQ.

Dans l'exemple suivant, la première itération de la boucle `foreach` fait que l'exécution continue dans la méthode d'itérateur `SomeNumbers`, jusqu'à ce que la première instruction `yield return` soit atteinte. Cette itération retourne la valeur 3, et l'emplacement actif dans la méthode d'itérateur est mémorisé. Dans l'itération suivante de la boucle, l'exécution de la méthode d'itérateur continue là où elle s'était arrêtée, et s'arrête de nouveau quand elle atteint une instruction `yield return`. Cette itération retourne la valeur 5, et l'emplacement actif dans la méthode d'itérateur est mémorisé. La boucle se termine quand la fin de la méthode d'itérateur est atteinte.

C#

```
static void Main()
{
    foreach (int number in SomeNumbers())
    {
        Console.Write(number.ToString() + " ");
    }
    // Output: 3 5 8
    Console.ReadKey();
}

public static System.Collections.IEnumerable SomeNumbers()
{
    yield return 3;
    yield return 5;
    yield return 8;
}
```

Le type de retour d'une méthode d'itérateur ou de l'accesseur `get` peut être IEnumerable, IEnumerable<T>, IEnumerator ou IEnumerator<T>.

Utilisez une instruction `yield break` pour terminer l'itération.

> ⓘ Notes
>
> Tous les exemples de cette rubrique, à l'exception de l'exemple de l'itérateur simple, incluent des directives **using** pour les espaces de noms `System.Collections` et `System.Collections.Generic`.

Itérateur simple

L'exemple suivant comprend une seule instruction `yield return` qui se trouve dans une boucle for. Dans `Main`, chaque itération du corps d'instruction `foreach` crée un appel à la fonction d'itérateur, qui poursuit avec l'instruction `yield return` suivante.

C#

```
static void Main()
{
    foreach (int number in EvenSequence(5, 18))
    {
        Console.Write(number.ToString() + " ");
    }
    // Output: 6 8 10 12 14 16 18
    Console.ReadKey();
}

public static System.Collections.Generic.IEnumerable<int>
    EvenSequence(int firstNumber, int lastNumber)
{
    // Yield even numbers in the range.
    for (int number = firstNumber; number <= lastNumber; number++)
    {
        if (number % 2 == 0)
        {
            yield return number;
        }
    }
}
```

Création d'une classe de collection

Dans l'exemple suivant, la classe `DaysOfTheWeek` implémente l'interface IEnumerable, ce qui nécessite la méthode GetEnumerator. Le compilateur appelle implicitement la méthode `GetEnumerator`, qui retourne un IEnumerator.

La méthode `GetEnumerator` retourne chaque chaîne une à la fois en utilisant l'instruction `yield return`.

C#

```
static void Main()
{
    DaysOfTheWeek days = new DaysOfTheWeek();

    foreach (string day in days)
    {
        Console.Write(day + " ");
    }
    // Output: Sun Mon Tue Wed Thu Fri Sat
    Console.ReadKey();
}

public class DaysOfTheWeek : IEnumerable
{
    private string[] days = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri",
"Sat" };

    public IEnumerator GetEnumerator()
    {
        for (int index = 0; index < days.Length; index++)
        {
            // Yield each day of the week.
            yield return days[index];
        }
    }
}
```

L'exemple suivant crée une classe `Zoo` qui contient une collection d'animaux.

L'instruction `foreach` qui fait référence à l'instance de classe (`theZoo`) appelle implicitement la méthode `GetEnumerator`. Les instructions `foreach` qui font référence aux propriétés `Birds` et `Mammals` utilisent la méthode d'itérateur nommée `AnimalsForType`.

C#

```
static void Main()
{
    Zoo theZoo = new Zoo();
```

```
    theZoo.AddMammal("Whale");
    theZoo.AddMammal("Rhinoceros");
    theZoo.AddBird("Penguin");
    theZoo.AddBird("Warbler");

    foreach (string name in theZoo)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Whale Rhinoceros Penguin Warbler

    foreach (string name in theZoo.Birds)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Penguin Warbler

    foreach (string name in theZoo.Mammals)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Whale Rhinoceros

    Console.ReadKey();
}

public class Zoo : IEnumerable
{
    // Private members.
    private List<Animal> animals = new List<Animal>();

    // Public methods.
    public void AddMammal(string name)
    {
        animals.Add(new Animal { Name = name, Type = Animal.TypeEnum.Mammal
});
    }

    public void AddBird(string name)
    {
        animals.Add(new Animal { Name = name, Type = Animal.TypeEnum.Bird
});
    }

    public IEnumerator GetEnumerator()
    {
        foreach (Animal theAnimal in animals)
        {
            yield return theAnimal.Name;
        }
    }
```

```
    // Public members.
    public IEnumerable Mammals
    {
        get { return AnimalsForType(Animal.TypeEnum.Mammal); }
    }

    public IEnumerable Birds
    {
        get { return AnimalsForType(Animal.TypeEnum.Bird); }
    }

    // Private methods.
    private IEnumerable AnimalsForType(Animal.TypeEnum type)
    {
        foreach (Animal theAnimal in animals)
        {
            if (theAnimal.Type == type)
            {
                yield return theAnimal.Name;
            }
        }
    }

    // Private class.
    private class Animal
    {
        public enum TypeEnum { Bird, Mammal }

        public string Name { get; set; }
        public TypeEnum Type { get; set; }
    }
}
```

Utilisation d'itérateurs avec une liste générique

Dans l'exemple suivant, la classe générique Stack<T> implémente l'interface générique IEnumerable<T>. La méthode Push affecte des valeurs à un tableau de type `T`. La méthode GetEnumerator retourne les valeurs de tableau à l'aide de l'instruction `yield return`.

Outre la méthode GetEnumerator générique, la méthode GetEnumerator non générique doit également être implémentée. C'est parce que IEnumerable<T> hérite de IEnumerable. L'implémentation non générique s'en remet à l'implémentation générique.

L'exemple utilise des itérateurs nommés pour prendre en charge différentes façons d'itérer au sein de la même collection de données. Ces itérateurs nommés sont les propriétés `TopToBottom` et `BottomToTop`, et la méthode `TopN`.

La propriété `BottomToTop` utilise un itérateur dans un accesseur `get`.

C#

```
static void Main()
{
    Stack<int> theStack = new Stack<int>();

    //  Add items to the stack.
    for (int number = 0; number <= 9; number++)
    {
        theStack.Push(number);
    }

    // Retrieve items from the stack.
    // foreach is allowed because theStack implements IEnumerable<int>.
    foreach (int number in theStack)
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 9 8 7 6 5 4 3 2 1 0

    // foreach is allowed, because theStack.TopToBottom returns
IEnumerable(Of Integer).
    foreach (int number in theStack.TopToBottom)
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 9 8 7 6 5 4 3 2 1 0

    foreach (int number in theStack.BottomToTop)
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 0 1 2 3 4 5 6 7 8 9

    foreach (int number in theStack.TopN(7))
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 9 8 7 6 5 4 3

    Console.ReadKey();
}

public class Stack<T> : IEnumerable<T>
{
    private T[] values = new T[100];
    private int top = 0;
```

```
    public void Push(T t)
    {
        values[top] = t;
        top++;
    }
    public T Pop()
    {
        top--;
        return values[top];
    }

    // This method implements the GetEnumerator method. It allows
    // an instance of the class to be used in a foreach statement.
    public IEnumerator<T> GetEnumerator()
    {
        for (int index = top - 1; index >= 0; index--)
        {
            yield return values[index];
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }

    public IEnumerable<T> TopToBottom
    {
        get { return this; }
    }

    public IEnumerable<T> BottomToTop
    {
        get
        {
            for (int index = 0; index <= top - 1; index++)
            {
                yield return values[index];
            }
        }
    }

    public IEnumerable<T> TopN(int itemsFromTop)
    {
        // Return less than itemsFromTop if necessary.
        int startIndex = itemsFromTop >= top ? 0 : top - itemsFromTop;

        for (int index = top - 1; index >= startIndex; index--)
        {
            yield return values[index];
        }
    }

}
```

Informations sur la syntaxe

Un itérateur peut être une méthode ou un accesseur `get`. Un itérateur ne peut pas être présent dans un événement, un constructeur d'instance, un constructeur statique ou un finaliseur statique.

Il doit y avoir une conversion implicite entre le type d'expression dans l'instruction `yield return` et l'argument de type pour l'`IEnumerable<T>` retourné par l'itérateur.

En C#, une méthode d'itérateur ne peut avoir aucun paramètre `in`, `ref` ou `out`.

En C#, `yield` n'est pas un mot réservé ; il a une signification spéciale seulement quand il est utilisé avant un mot clé `return` ou `break`.

Implémentation technique

Bien que vous écriviez un itérateur comme une méthode, le compilateur le traduit en une classe imbriquée, qui est en réalité une machine à états. Cette classe fait le suivi de la position de l'itérateur tant que la boucle `foreach` du code client se poursuit.

Pour voir ce que fait le compilateur, vous pouvez utiliser l'outil Ildasm.exe et afficher le code du langage intermédiaire Microsoft généré pour une méthode d'itérateur.

Quand vous créez un itérateur pour une classe ou un struct, vous n'avez pas besoin d'implémenter l'ensemble de l'interface IEnumerator. Quand le compilateur détecte l'itérateur, il génère automatiquement les méthodes `Current`, `MoveNext` et `Dispose` de l'interface IEnumerator ou IEnumerator<T>.

À chaque itération successive de la boucle `foreach` (ou de l'appel direct à `IEnumerator.MoveNext`), le corps du code de l'itérateur suivant reprend après l'instruction `yield return` précédente. Il continue ensuite jusqu'à l'instruction `yield return` suivante, jusqu'à atteindre la fin du corps de l'itérateur ou jusqu'à rencontrer une instruction `yield break`.

Les itérateurs ne prennent pas en charge la méthode IEnumerator.Reset. Pour réitérer à partir du début, vous devez obtenir un nouvel itérateur. L'appel de Reset sur l'itérateur retourné par une méthode d'itérateur lève un NotSupportedException.

Pour plus d'informations, consultez la spécification du langage C#.

Utilisation d'itérateurs

Les itérateurs vous permettent de conserver la simplicité d'une boucle `foreach` quand vous avez besoin d'utiliser un code complexe pour remplir la séquence d'une liste. Ceci peut être utile quand vous voulez faire les opérations suivantes :

- Modifier la séquence de la liste après la première itération de la boucle `foreach`.
- Éviter de charger entièrement une grande liste avant la première itération d'une boucle `foreach`. Une extraction paginée pour charger un lot de lignes d'une table est un exemple. Un autre exemple est la méthode EnumerateFiles, qui implémente les itérateurs dans .NET.
- Encapsuler la création de la liste dans l'itérateur. Dans la méthode d'itérateur, vous pouvez créer la liste et générer ensuite chaque résultat dans une boucle.

Voir aussi

- System.Collections.Generic
- IEnumerable<T>
- foreach, in
- Utiliser foreach avec des tableaux
- Génériques

LINQ (Language-Integrated Query)

Article • 24/02/2023

LINQ (Language-Integrated Query) est le nom d'un ensemble de technologies basé sur l'intégration de fonctions de requête directement dans le langage C#. En règle générale, les requêtes de données sont exprimées comme de simples chaînes, sans vérification de type au moment de l'exécution, ni prise en charge IntelliSense. En outre, vous devez apprendre un langage de requête différent pour chaque type de source de données : bases de données SQL, documents XML, services web, etc. Avec LINQ, une requête est une construction de langage de premier ordre, comme les classes, les méthodes et les événements.

Pour un développeur qui écrit des requêtes, la partie « intégrée au langage » la plus visible de LINQ est l'expression de requête. Les expressions de requête sont écrites selon une *syntaxe de requête* déclarative. En utilisant la syntaxe de requête, vous pouvez effectuer des opérations de filtrage, de classement et de regroupement sur des sources de données avec un minimum de code. Vous utilisez les mêmes modèles d'expression de requête de base pour interroger et transformer des données dans les bases de données SQL, les jeux de données ADO .NET, les flux et documents XML et les collections .NET.

L'exemple suivant montre l'opération de requête complète. L'opération complète comprend la création d'une source de données, la définition de l'expression de requête et l'exécution de la requête dans une instruction `foreach`.

C#

```
// Specify the data source.
int[] scores = { 97, 92, 81, 60 };

// Define the query expression.
IEnumerable<int> scoreQuery =
    from score in scores
    where score > 80
    select score;

// Execute the query.
foreach (int i in scoreQuery)
{
    Console.Write(i + " ");
}

// Output: 97 92 81
```

Vue d'ensemble des expressions de requête

- Les expressions de requête peuvent être utilisées pour interroger et transformer des données à partir de n'importe quelle source de données compatible LINQ. Par exemple, une même requête peut récupérer des données d'une base de données SQL et générer un flux XML en sortie.

- Les expressions de requête sont faciles à saisir car elles utilisent de nombreuses constructions familières du langage C#.

- Les variables d'une expression de requête sont toutes fortement typées, même si, dans de nombreux cas, il n'est pas nécessaire de fournir le type explicitement, car le compilateur peut le déduire. Pour plus d'informations, consultez la page Relations entre les types dans les opérations de requête LINQ.

- Une requête ne s'exécute pas tant que vous n'avez pas itéré la variable de requête, par exemple dans une instruction `foreach`. Pour plus d'informations, consultez Introduction aux requêtes LINQ.

- Lors de la compilation, les expressions de requête sont converties en appels de méthode d'opérateur de requête standard en fonction des règles définies dans la spécification du langage C#. Toute requête exprimable avec la syntaxe de requête peut également être exprimée avec la syntaxe de méthode. Toutefois, dans la plupart des cas, la syntaxe de requête est plus lisible et plus concise. Pour plus d'informations, consultez les pages Spécification du langage C# et Vue d'ensemble des opérateurs de requête standard.

- En règle générale, lorsque vous écrivez des requêtes LINQ, nous vous recommandons d'utiliser la syntaxe de requête dans la mesure du possible et la syntaxe de méthode si nécessaire. Il n'y a aucune différence de sémantique ou de performances entre les deux formats. Les expressions de requête sont souvent plus lisibles que les expressions équivalentes écrites avec la syntaxe de méthode.

- Certaines opérations de requête, comme Count ou Max, n'ont pas d'expression de requête équivalente et doivent par conséquent être exprimées sous la forme d'un appel de méthode. La syntaxe de méthode peut être combinée avec la syntaxe de requête de différentes manières. Pour plus d'informations, consultez la page Syntaxe de requête et syntaxe de méthode dans LINQ.

- Les expressions de requête peuvent être compilées dans des arborescences d'expressions ou des délégués, selon le type auquel la requête est appliquée. Les requêtes IEnumerable<T> sont compilées en délégués. Les requêtes IQueryable et

IQueryable<T> sont compilées en arborescences d'expression. Pour plus d'informations, consultez la page Arborescences d'expressions.

Étapes suivantes

Pour approfondir votre connaissance de LINQ, commencez par vous familiariser avec certains concepts de base expliqués sur la page Principes de base des expressions de requête, puis lisez la documentation relative à la technologie LINQ qui vous intéresse :

- Documents XML : LINQ to XML
- ADO.NET Entity Framework : LINQ to Entities
- Collections, fichiers, chaînes, etc. .NET : LINQ to Objects

Pour approfondir votre compréhension de LINQ en général, consultez la page LINQ en C#.

Pour commencer à travailler avec LINQ en C#, consultez le didacticiel Travailler avec LINQ.

Introduction aux requêtes LINQ (C#)

Article • 10/05/2023

Une *requête* est une expression qui récupère des données d'une source de données. Les requêtes sont généralement exprimées dans un langage de requête spécialisé. Les différents langages ont été développés au fil du temps pour les différents types de sources de données, par exemple, SQL pour les bases de données relationnelles et XQuery pour XML. Les développeurs devaient donc apprendre un nouveau langage de requête pour chaque nouveau type de source de données ou format de données qu'ils devaient prendre en charge. LINQ simplifie cette situation en proposant un modèle cohérent pour travailler avec des données dans différents types de sources et de formats de données. Dans une requête LINQ, vous travaillez toujours avec des objets. Vous utilisez les mêmes modèles de codage de base pour interroger et transformer des données dans des documents XML, des bases de données SQL, des jeux de données ADO.NET, des collections .NET et tout autre format pour lequel un fournisseur LINQ est disponible.

Les trois parties d'une opération de requête

Toutes les opérations de requête LINQ se composent de trois actions distinctes :

1. Obtention de la source de données
2. Création de la requête
3. exécutez la requête.

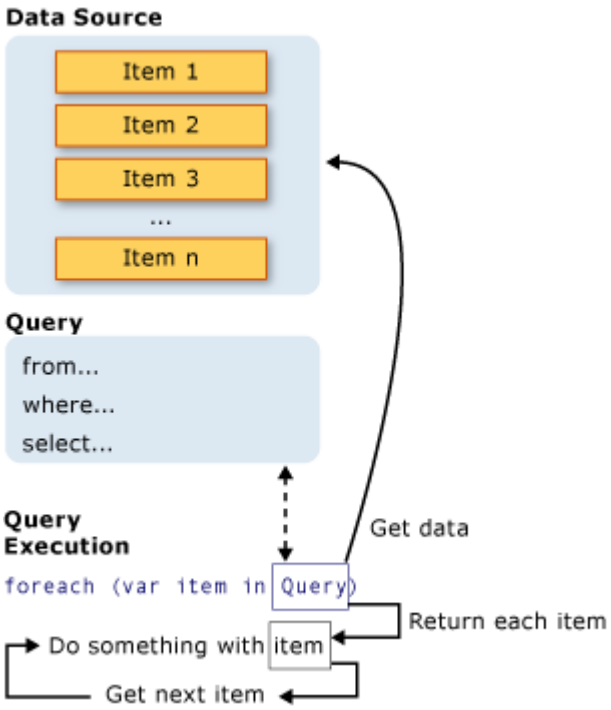
L'exemple suivant montre comment les trois parties d'une opération de requête sont exprimées dans le code source. Cet exemple utilise un tableau d'entiers comme source de données pour des raisons pratiques. Toutefois, ces mêmes concepts s'appliquent également aux autres sources de données. Le reste de la rubrique fait référence à cet exemple.

C#

```
class IntroToLINQ
{
    static void Main()
    {
        // The Three Parts of a LINQ Query:
        // 1. Data source.
        int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };
```

```
        // 2. Query creation.
        // numQuery is an IEnumerable<int>
        var numQuery =
            from num in numbers
            where (num % 2) == 0
            select num;

        // 3. Query execution.
        foreach (int num in numQuery)
        {
            Console.Write("{0,1} ", num);
        }
    }
}
```

L'illustration suivante montre l'intégralité de l'opération de requête. Dans LINQ, l'exécution de la requête est distincte de la requête elle-même. En d'autres termes, vous n'avez récupéré aucune donnée en créant simplement une variable de requête.



Source de données

Dans l'exemple précédent, comme la source de données est un tableau, elle prend en charge implicitement l'interface générique IEnumerable<T>. Cela signifie qu'elle peut être interrogée avec LINQ. Une requête est exécutée dans une instruction `foreach`, et `foreach` nécessite IEnumerable ou IEnumerable<T>. Les types qui prennent en charge IEnumerable<T> ou une interface dérivée, comme l'interface générique IQueryable<T>, sont appelés des *types requêtables*.

Un type requêtable ne nécessite aucune modification ni traitement spécial pour servir de source de données LINQ. Si les données sources ne sont pas déjà en mémoire en tant que type requêtable, le fournisseur LINQ doit le représenter comme tel. Par exemple, LINQ to XML charge un document XML dans un type requêtable XElement :

C#

```
// Create a data source from an XML document.
// using System.Xml.Linq;
XElement contacts = XElement.Load(@"c:\myContactList.xml");
```

Avec LINQ to SQL, vous créez tout d'abord un mappage relationnel d'objet au moment de la conception, soit manuellement, soit à l'aide des outils de LINQ to SQL dans Visual Studio. Vous écrivez vos requêtes sur les objets, et à l'exécution ; LINQ to SQL gère la communication avec la base de données. Dans l'exemple suivant, `Customers` représente une table spécifique de la base de données et le type du résultat de la requête, IQueryable<T>, dérive de IEnumerable<T>.

C#

```
Northwnd db = new Northwnd(@"c:\northwnd.mdf");

// Query for customers in London.
IQueryable<Customer> custQuery =
    from cust in db.Customers
    where cust.City == "London"
    select cust;
```

Pour plus d'informations sur la création de types spécifiques de sources de données, consultez la documentation relative aux différents fournisseurs LINQ. Toutefois, la règle de base est très simple : une source de données LINQ est un objet qui prend en charge l'interface générique IEnumerable<T> ou une interface qui hérite de celui-ci.

> ⓘ **Notes**
>
> Les types tels que **ArrayList** qui prennent en charge l'interface non générique **IEnumerable** peuvent également être utilisés comme source de données Linq. Pour plus d'informations, consultez **Guide pratique pour interroger une liste de tableaux avec LINQ (C#)**.

La requête

La requête spécifie les informations à récupérer à partir de la ou des sources de données. Si vous le souhaitez, une requête peut également spécifier la manière dont ces informations doivent être triées, regroupées et mises en forme avant d'être retournées. Une requête est stockée dans une variable de requête et initialisée avec une expression de requête. Pour faciliter l'écriture de requêtes, le langage C# propose désormais une nouvelle syntaxe de requête.

La requête de l'exemple précédent retourne tous les nombres pairs du tableau d'entiers. L'expression de requête contient trois clauses : `from`, `where` et `select`. (Si vous connaissez SQL, vous aurez remarqué que l'ordre des clauses est inversé par rapport à l'ordre dans SQL.) La clause `from` spécifie la source de données, la clause `where` applique le filtre et la clause `select` spécifie le type des éléments retournés. Celles-ci et les autres clauses de requête sont abordées en détail dans la section Language Integrated Query (LINQ). Pour le moment, le point important est que dans LINQ, la variable de requête elle-même n'effectue aucune action et ne retourne aucune donnée. Elle stocke simplement les informations qui seront nécessaires pour produire des résultats lors de l'exécution ultérieure de la requête. Pour plus d'informations sur la construction des requêtes en arrière-plan, consultez Vue d'ensemble des opérateurs de requête standard (C#).

> ⓘ **Notes**
>
> Les requêtes peuvent également être exprimées à l'aide d'une syntaxe de méthode. Pour plus d'informations, consultez **Syntaxe de requête et syntaxe de méthode dans LINQ**.

Exécution des requêtes

Exécution différée

Comme indiqué précédemment, la variable de requête stocke uniquement les commandes de requête. L'exécution de la requête est différée jusqu'à ce que vous itériez au sein de la variable de requête dans une instruction `foreach`. Ce concept est appelé *exécution différée* et est illustré dans l'exemple suivant :

C#

```
//  Query execution.
foreach (int num in numQuery)
{
```

```
    Console.Write("{0,1} ", num);
}
```

C'est dans l'instruction `foreach` que les résultats de requête sont récupérés. Par exemple, dans la requête précédente, la variable d'itération `num` contient chaque valeur (une à la fois) de la séquence retournée.

Étant donné que la variable de requête ne contient jamais les résultats de requête, vous pouvez l'exécuter aussi souvent que vous le voulez. Par exemple, vous disposez peut-être d'une base de données qui est constamment mise à jour par une application distincte. Vous pouvez créer dans cette dernière une requête dans le but d'extraire les données les plus récentes, puis l'exécuter à intervalle régulier. Vous obtiendrez à chaque fois des résultats différents.

Forcer l'exécution immédiate

Les requêtes qui exécutent des fonctions d'agrégation sur une plage d'éléments sources doivent d'abord itérer au sein de ces éléments. Ces requêtes sont par exemple `Count`, `Max`, `Average` et `First`. Elles s'exécutent sans instruction `foreach` explicite, car les requêtes doivent utiliser `foreach` pour retourner un résultat. Notez également que ces types de requêtes retournent une seule valeur, et non une collection `IEnumerable`. La requête suivante retourne un nombre de chiffres pairs du tableau source :

C#

```
var evenNumQuery =
    from num in numbers
    where (num % 2) == 0
    select num;

int evenNumCount = evenNumQuery.Count();
```

Pour forcer l'exécution immédiate de n'importe quelle requête et mettre en cache ses résultats, vous pouvez appeler les méthodes ToList ou ToArray.

C#

```
List<int> numQuery2 =
    (from num in numbers
     where (num % 2) == 0
     select num).ToList();

// or like this:
// numQuery3 is still an int[]
```

```
var numQuery3 =
    (from num in numbers
     where (num % 2) == 0
     select num).ToArray();
```

Vous pouvez également forcer l'exécution en plaçant la boucle `foreach` immédiatement après l'expression de requête. Toutefois, en appelant `ToList` ou `ToArray`, vous mettez également en cache toutes les données dans un même objet de collection.

Voir aussi

- Mise en route de LINQ en C#
- Procédure pas à pas : écriture de requêtes en C#
- LINQ (Language-Integrated Query)
- foreach, in
- Mots clés de requête (LINQ)

Relations des types dans des opérations de requête LINQ (C#)

Article • 20/07/2023

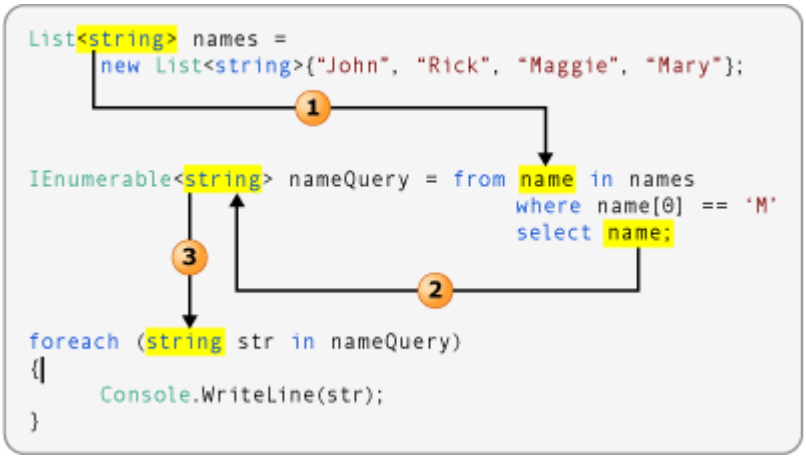
Pour écrire efficacement des requêtes, vous devez comprendre comment les types des variables dans une opération de requête complète sont liés les uns aux autres. Si vous comprenez ces relations, vous comprendrez plus facilement les exemples de code et les échantillons LINQ dans la documentation. En outre, vous comprendrez ce qui se passe en arrière-plan lorsque des variables sont implicitement typées à l'aide de `var`.

Les opérations de requête LINQ sont fortement typées dans la source de données, dans la requête elle-même et dans l'exécution de la requête. Le type des variables dans la requête doit être compatible avec le type des éléments dans la source de données, ainsi qu'avec le type de la variable d'itération dans l'instruction `foreach`. Ce typage fort garantit l'interception des erreurs de type au moment de la compilation lorsqu'elles peuvent être corrigées avant que les utilisateurs ne les rencontrent.

Pour illustrer ces relations de types, la plupart des exemples qui suivent utilisent le typage explicite pour toutes les variables. Le dernier exemple montre comment les mêmes principes s'appliquent même si vous utilisez le typage implicite à l'aide de var.

Requêtes qui ne transforment pas les données sources

L'illustration suivante montre une opération de requête LINQ to Objects qui n'effectue aucune transformation des données. La source contient une séquence de chaînes et le résultat de la requête est également une séquence de chaînes.



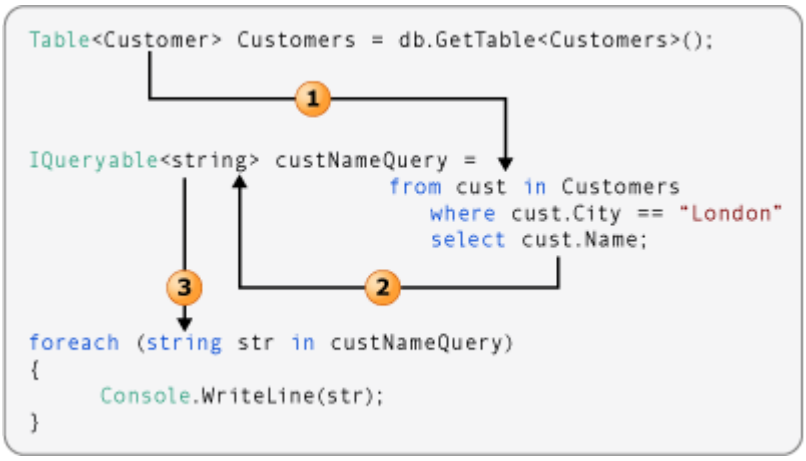
1. L'argument de type de la source de données détermine le type de la variable de portée.

2. Le type de l'objet sélectionné détermine le type de la variable de requête. Ici, `name` est une chaîne. Par conséquent, la variable de requête est un `IEnumerable<string>`.

3. La variable de requête fait l'objet d'une itération dans l'instruction `foreach`. Comme la variable de requête est une séquence de chaînes, la variable d'itération est également une chaîne.

Requêtes qui transforment les données sources

L'illustration suivante montre une opération de requête LINQ to SQL qui effectue une transformation simple des données. La requête accepte une séquence d'objets `Customer` en entrée et sélectionne uniquement la propriété `Name` dans le résultat. Comme `Name` est une chaîne, la requête produit une séquence de chaînes en sortie.

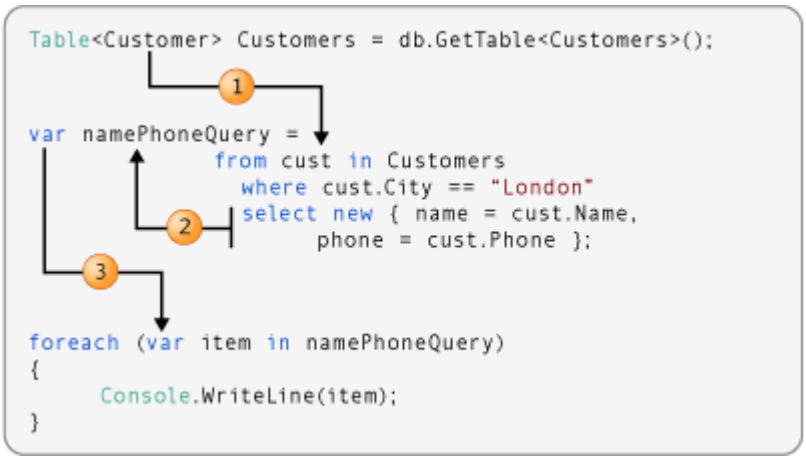


1. L'argument de type de la source de données détermine le type de la variable de portée.

2. L'instruction `select` retourne la propriété `Name` à la place de l'objet `Customer` complet. Comme `Name` est une chaîne, l'argument de type de `custNameQuery` est `string`, et non pas `Customer`.

3. Comme `custNameQuery` est une séquence de chaînes, la variable d'itération de la boucle `foreach` doit également être de type `string`.

L'illustration suivante montre une transformation légèrement plus complexe. L'instruction `select` retourne un type anonyme qui capture seulement deux membres de l'objet `Customer` original.



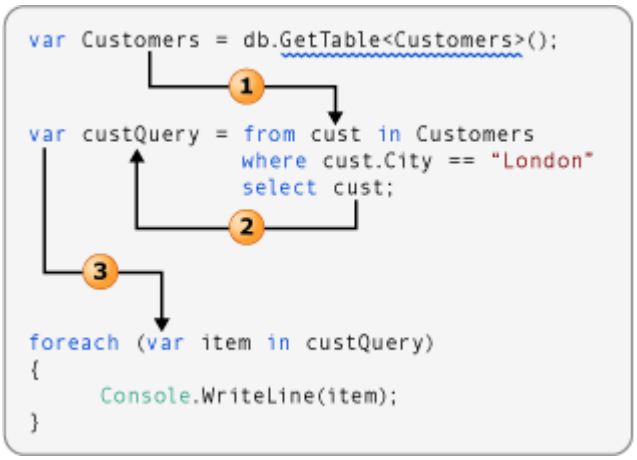
1. L'argument de type de la source de données est toujours le type de la variable de portée dans la requête.

2. Comme l'instruction `select` génère un type anonyme, la variable de requête doit être implicitement typée à l'aide de `var`.

3. Comme le type de la variable de requête est implicite, la variable d'itération dans la boucle `foreach` doit également être implicite.

Laisser le compilateur déduire les informations de type

Vous devez comprendre les relations des types dans une opération de requête. Toutefois, vous avez la possibilité de laisser le compilateur faire tout le travail à votre place. Le mot clé var peut être utilisé pour toute variable locale dans une opération de requête. L'illustration suivante est similaire à l'exemple numéro 2 qui a été abordé précédemment. Toutefois, le compilateur fournit le type fort pour chaque variable dans l'opération de requête.



Pour plus d'informations sur `var`, consultez Variables locales implicitement typées.

LINQ et types génériques (C#)

Les requêtes LINQ sont basées sur des types génériques. Vous ne devez pas avoir une connaissance approfondie des génériques avant de commencer à écrire des requêtes. Il peut cependant être important de comprendre deux concepts de base :

1. Quand vous créez une instance d'une classe de collection générique comme List<T>, vous remplacez le « T » par le type des objets contenus dans la liste. Par exemple, une liste de chaînes est exprimée sous la forme `List<string>`, et une liste d'objets `Customer` est exprimée sous la forme `List<Customer>`. Une liste générique est fortement typée et offre de nombreux avantages par rapport aux collections qui stockent leurs éléments en tant que Object. Si vous essayez d'ajouter un `Customer` à un `List<string>`, vous obtenez une erreur à la compilation. Il est facile d'utiliser des collections génériques, car vous ne devez pas effectuer un cast de type à l'exécution.

2. IEnumerable<T> est l'interface qui permet l'énumération des classes de collection génériques avec l'instruction `foreach`. Les classes de collection génériques prennent en charge IEnumerable<T> de la même façon que les classes de collection non génériques telles que ArrayList prennent en charge IEnumerable.

Pour plus d'informations sur les génériques, consultez Génériques.

Variables IEnumerable<T> dans les requêtes LINQ

Les variables de requête LINQ sont typées en tant que IEnumerable<T> ou un type dérivé tel que IQueryable<T>. Quand vous voyez une variable de requête typée en `IEnumerable<Customer>`, cela signifie simplement que la requête génère à l'exécution une séquence de zéro ou plusieurs objets `Customer`.

C#

```
IEnumerable<Customer> customerQuery =
    from cust in customers
    where cust.City == "London"
    select cust;

foreach (Customer customer in customerQuery)
{
    Console.WriteLine(customer.LastName + ", " + customer.FirstName);
}
```

Pour plus d'informations, consultez Relations des types dans des opérations de requête LINQ.

Laisser le compilateur gérer les déclarations de type générique

Si vous préférez, vous pouvez éviter une syntaxe générique en utilisant le mot clé var. Le mot clé `var` indique au compilateur d'inférer le type d'une variable de requête en examinant la source de données spécifiée dans la clause `from`. L'exemple suivant génère le même code compilé que l'exemple précédent :

C#

```
var customerQuery2 =
    from cust in customers
    where cust.City == "London"
    select cust;

foreach(var customer in customerQuery2)
{
    Console.WriteLine(customer.LastName + ", " + customer.FirstName);
}
```

Le mot clé `var` est utile quand le type de la variable est évident ou quand il est particulièrement important de spécifier explicitement des types génériques imbriqués, comme ceux qui sont produits par des requêtes de groupe. D'une façon générale, si vous utilisez `var`, sachez qu'il peut rendre votre code plus difficile à lire par d'autres développeurs. Pour plus d'informations, consultez Variables locales implicitement typées.

Présentation des opérateurs de requête standard (C#)

Article • 20/07/2023

Les *opérateurs de requête standard* sont les méthodes qui forment le modèle de requête LINQ. La plupart de ces méthodes fonctionnent sur des séquences, où une séquence est un objet dont le type implémente l'interface IEnumerable<T> ou l'interface IQueryable<T>. Les opérateurs de requête standard fournissent des fonctionnalités de requête, dont notamment le filtrage, la projection, l'agrégation, le tri, etc.

Il existe deux ensembles d'opérateurs de requête standard LINQ : l'un opérant sur des objets de type IEnumerable<T> et l'autre sur des objets de type IQueryable<T>. Les méthodes qui composent chaque ensemble sont des membres statiques des classes Enumerable et Queryable, respectivement. Elles sont définies en tant que *méthodes d'extension* du type sur lequel elles opèrent. Les méthodes d'extension peuvent être appelées à l'aide de la syntaxe de méthode statique ou de la syntaxe de méthode d'instance.

De plus, plusieurs méthodes d'opérateur de requête standard fonctionnent sur des types autres que ceux basés sur IEnumerable<T> ou IQueryable<T>. Le type Enumerable définit deux de ces méthodes qui fonctionnent toutes deux sur les objets de type IEnumerable. Ces méthodes, Cast<TResult>(IEnumerable) et OfType<TResult>(IEnumerable), vous permettent d'autoriser l'interrogation d'une collection non paramétrée, ou non générique, dans le modèle LINQ. Pour cela, elles créent une collection fortement typée d'objets. La classe Queryable définit deux méthodes similaires, Cast<TResult>(IQueryable) et OfType<TResult>(IQueryable), qui opèrent sur les objets de type IQueryable.

Les opérateurs de requête standard diffèrent dans le déroulement de leur exécution, selon qu'ils retournent une valeur singleton ou une séquence de valeurs. Les méthodes qui retournent une valeur de singleton (par exemple, Average et Sum) s'exécutent immédiatement. Les méthodes qui retournent une séquence diffèrent l'exécution de la requête et retournent un objet énumérable.

Pour les méthodes qui opèrent sur des collections en mémoire, c'est-à-dire des méthodes qui étendent IEnumerable<T>, l'objet énumérable retourné capture les arguments passés à la méthode. Lorsque cet objet est énuméré, la logique de l'opérateur de requête est employée et les résultats de requête sont retournés.

En revanche, les méthodes qui étendent IQueryable<T> n'implémentent aucun comportement d'interrogation. Elles créent une arborescence d'expressions

représentant la requête à effectuer. Le traitement des requêtes est géré par l'objet IQueryable<T> source.

Les appels aux méthodes de requête peuvent être chaînés dans une requête unique, ce qui permet de rendre les requêtes arbitrairement complexes.

L'exemple de code suivant illustre l'utilisation des opérateurs de requête standard pour obtenir des informations sur une séquence.

C#

```
string sentence = "the quick brown fox jumps over the lazy dog";
// Split the string into individual words to create a collection.
string[] words = sentence.Split(' ');

// Using query expression syntax.
var query = from word in words
            group word.ToUpper() by word.Length into gr
            orderby gr.Key
            select new { Length = gr.Key, Words = gr };

// Using method-based query syntax.
var query2 = words.
    GroupBy(w => w.Length, w => w.ToUpper()).
    Select(g => new { Length = g.Key, Words = g }).
    OrderBy(o => o.Length);

foreach (var obj in query)
{
    Console.WriteLine("Words of length {0}:", obj.Length);
    foreach (string word in obj.Words)
        Console.WriteLine(word);
}

// This code example produces the following output:
//
// Words of length 3:
// THE
// FOX
// THE
// DOG
// Words of length 4:
// OVER
// LAZY
// Words of length 5:
// QUICK
// BROWN
// JUMPS
```

Syntaxe d'expression de requête

Certains des opérateurs de requête standard les plus courants ont une syntaxe de mots clés dédiée des langages C# et Visual Basic, qui permet de les appeler dans le cadre d'une *expression* de *requête*. Pour plus d'informations sur les opérateurs de requête standard qui ont des mots clés dédiés et sur leurs syntaxes correspondantes, consultez Syntaxe des expressions de requête pour les opérateurs de requête standard (C#).

Extension des opérateurs de requête standard

Vous pouvez augmenter l'ensemble d'opérateurs de requête standard en créant des méthodes spécifiques au domaine appropriées pour votre domaine ou technologie cible. Vous pouvez également remplacer les opérateurs de requête standard par vos propres implémentations qui fournissent des services supplémentaires, tels que l'évaluation à distance, la traduction des requêtes et l'optimisation. Pour obtenir un exemple, consultez AsEnumerable.

Obtenir une source de données

Dans une requête LINQ, la première étape consiste à spécifier la source de données. Dans C#, comme dans la plupart des langages de programmation, une variable doit être déclarée avant de pouvoir être utilisée. Dans une requête LINQ, la clause `from` apparaît en premier pour introduire la source de données (`customers`) et la *variable de portée* (`cust`).

C#

```
//queryAllCustomers is an IEnumerable<Customer>
var queryAllCustomers = from cust in customers
                        select cust;
```

La variable de portée est similaire à la variable d'itération dans une boucle `foreach`, à la différence qu'aucune itération réelle ne se produit dans une expression de requête. Quand la requête est exécutée, la variable de portée sert de référence à chaque élément consécutif dans `customers`. Comme le compilateur déduit le type de `cust`, vous n'avez pas à le spécifier explicitement. Des variables de portée supplémentaires peuvent être introduites par une clause `let`. Pour plus d'informations, consultez let, clause.

> ⓘ **Notes**
>
> Pour les sources de données non génériques telles que **ArrayList**, la variable de portée doit être explicitement typée. Pour plus d'informations, consultez **Guide**

pratique pour interroger un ArrayList avec LINQ (C#) et from, clause.

Filtrage

L'opération de requête la plus courante est probablement l'application d'un filtre sous forme d'expression booléenne. Du fait du filtre, la requête retourne uniquement les éléments pour lesquels l'expression a la valeur true. Le résultat est produit à l'aide de la clause `where`. En effet, le filtre spécifie les éléments à exclure de la séquence source. Dans l'exemple suivant, seuls les clients (`customers`) qui ont une adresse à Londres sont retournés.

C#

```
var queryLondonCustomers = from cust in customers
                           where cust.City == "London"
                           select cust;
```

Vous pouvez utiliser les opérateurs logiques C# `AND` et `OR` habituels pour appliquer le nombre d'expressions de filtre nécessaire dans la clause `where`. Par exemple, pour retourner uniquement les clients situés à « Londres » ET (`AND`) dont le nom est « Devon », vous devez écrire le code suivant :

C#

```
where cust.City == "London" && cust.Name == "Devon"
```

Pour retourner les clients situés à Londres ou à Paris, vous devez écrire le code suivant :

C#

```
where cust.City == "London" || cust.City == "Paris"
```

Pour plus d'informations, consultez where, clause.

Classement

Il est souvent utile de trier les données retournées. La clause `orderby` permet de trier les éléments de la séquence retournée en fonction du comparateur par défaut pour le type qui est trié. Par exemple, la requête suivante peut être étendue pour trier les résultats

selon la propriété `Name`. Comme `Name` est une chaîne, le comparateur par défaut effectue un tri alphabétique de A à Z.

C#

```
var queryLondonCustomers3 =
    from cust in customers
    where cust.City == "London"
    orderby cust.Name ascending
    select cust;
```

Pour trier les résultats dans l'ordre inverse, de Z à A, utilisez la clause `orderby… descending`.

Pour plus d'informations, consultez orderby, clause.

Regroupement

La clause `group` vous permet de grouper vos résultats selon une clé que vous spécifiez. Par exemple, vous pouvez spécifier un regroupement des résultats par ville (`City`) pour que tous les clients de Londres ou Paris soient dans des groupes individuels. Dans ce cas, utilisez la clé `cust.City`.

C#

```
// queryCustomersByCity is an IEnumerable<IGrouping<string, Customer>>
  var queryCustomersByCity =
      from cust in customers
      group cust by cust.City;

  // customerGroup is an IGrouping<string, Customer>
  foreach (var customerGroup in queryCustomersByCity)
  {
      Console.WriteLine(customerGroup.Key);
      foreach (Customer customer in customerGroup)
      {
          Console.WriteLine("    {0}", customer.Name);
      }
  }
```

Quand vous terminez une requête avec une clause `group`, les résultats sont présentés sous la forme d'une liste de listes. Chaque élément de la liste est un objet qui a un membre `Key` et une liste d'éléments qui sont regroupés sous cette clé. Quand vous itérez une requête qui produit une séquence de groupes, vous devez utiliser une boucle

`foreach` imbriquée. La boucle externe itère chaque groupe et la boucle interne itère les membres de chaque groupe.

Si vous devez faire référence aux résultats d'une opération de regroupement, utilisez le mot clé `into` pour créer un identificateur susceptible d'être interrogé ultérieurement. La requête suivante retourne uniquement les groupes qui contiennent plus de deux clients :

C#

```
// custQuery is an IEnumerable<IGrouping<string, Customer>>
var custQuery =
    from cust in customers
    group cust by cust.City into custGroup
    where custGroup.Count() > 2
    orderby custGroup.Key
    select custGroup;
```

Pour plus d'informations, consultez group, clause.

Jonction

Les opérations de jointure créent des associations entre les séquences qui ne sont pas explicitement modélisées dans les sources de données. Par exemple, effectuez une jointure pour rechercher tous les clients et distributeurs qui se trouvent au même endroit. Dans LINQ, la clause `join` fonctionne toujours par rapport à des collections d'objets plutôt que directement par rapport à des tables de base de données.

C#

```
var innerJoinQuery =
    from cust in customers
    join dist in distributors on cust.City equals dist.City
    select new { CustomerName = cust.Name, DistributorName = dist.Name };
```

Dans LINQ, vous n'avez pas à utiliser `join` aussi souvent que vous le faites dans SQL, car les clés étrangères dans LINQ sont représentées dans le modèle objet comme des propriétés qui contiennent une collection d'éléments. Par exemple, un objet `Customer` contient une collection d'objets `Order`. Au lieu d'effectuer une jointure, accédez aux commandes en utilisant la notation par points :

C#

```
from order in Customer.Orders...
```

Pour plus d'informations, consultez join, clause.

Sélection (projections)

La clause `select` produit les résultats de la requête et spécifie la « forme » ou le type de chaque élément retourné. Par exemple, spécifiez si les résultats doivent contenir des objets `Customer` complets, un seul membre, un sous-ensemble de membres ou un type de résultat complètement différent basé sur un calcul ou une création d'objet. Quand la clause `select` produit autre chose qu'une copie de l'élément source, l'opération est appelée *projection*. L'utilisation de projections pour transformer des données est une fonctionnalité puissante des expressions de requête LINQ. Pour plus d'informations, consultez Transformations de données avec LINQ (C#) et select, clause.

Tableau de syntaxe des expressions de requête

Le tableau ci-dessous répertorie les opérateurs de requête standard qui comportent des clauses d'expression de requête équivalentes.

Méthode	**Syntaxe d'expression de requête C#**
Cast	Utilisez une variable de portée explicitement typée, par exemple : `from int i in numbers` (Pour plus d'informations, consultez from, clause.)

GroupBy	`group … by`
	
	-ou-
	
	`group … by … into …`
	
	(Pour plus d'informations, consultez group, clause.)

GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>,	`join … in … on … equals … into …`

Méthode	Syntaxe d'expression de requête C#
Func<TOuter,IEnumerable<TInner>, TResult>)	(Pour plus d'informations, consultez join, clause.)
Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>)	`join … in … on … equals …` (Pour plus d'informations, consultez join, clause.)
OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	`orderby` (Pour plus d'informations, consultez orderby, clause.)
OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	`orderby … descending` (Pour plus d'informations, consultez orderby, clause.)
Select	`select` (Pour plus d'informations, consultez select, clause.)
SelectMany	Plusieurs clauses `from`. (Pour plus d'informations, consultez from, clause.)
ThenBy<TSource,TKey>(IOrderedEnumerable<TSource>, Func<TSource,TKey>)	`orderby …, …` (Pour plus d'informations, consultez orderby, clause.)
ThenByDescending<TSource,TKey>(IOrderedEnumerable<TSource>, Func<TSource,TKey>)	`orderby …, … descending` (Pour plus d'informations, consultez orderby, clause.)
Where	`where`

Méthode	Syntaxe d'expression de requête C#
	(Pour plus d'informations, consultez where, clause.)

Voir aussi

- Enumerable
- Queryable
- Méthodes d'extension
- Mots clés de requête (LINQ)
- Types anonymes

Transformations de données avec LINQ (C#)

Article • 09/05/2023

La requête intégrée en langage (LINQ) ne concerne pas seulement la récupération des données. C'est également un outil puissant pour la transformation de données. En utilisant une requête LINQ, vous pouvez utiliser une séquence source en entrée et la modifier de nombreuses façons pour créer une séquence de sortie. Vous pouvez modifier la séquence elle-même sans modifier les éléments eux-mêmes en les triant et en les regroupant. Mais la fonctionnalité la plus puissante des requêtes LINQ est peut-être la possibilité de créer des types. Cette opération est effectuée dans la clause select. Par exemple, il est possible de réaliser les tâches suivantes :

- Fusionner plusieurs séquences d'entrée en une séquence de sortie unique ayant un nouveau type.
- Créer des séquences de sortie dont les éléments se composent d'une seule propriété ou de plusieurs propriétés de chaque élément de la séquence source.
- Créer des séquences de sortie dont les éléments se composent des résultats des opérations effectuées sur les données sources.
- Créer des séquences de sortie dans un format différent. Par exemple, vous pouvez transformer des données de lignes ou de fichiers texte SQL en données XML.

Voici une liste non exhaustive d'exemples. Bien sûr, ces transformations peuvent être combinées de plusieurs façons dans la même requête. Par ailleurs, la séquence de sortie d'une requête peut être utilisée comme séquence d'entrée pour une nouvelle requête.

Combinaison de plusieurs entrées en une séquence de sortie

Vous pouvez utiliser une requête LINQ pour créer une séquence de sortie qui contient des éléments de plusieurs séquences d'entrée. L'exemple suivant montre comment combiner deux structures de données en mémoire, mais les mêmes principes peuvent être appliqués pour combiner des données provenant de sources XML, SQL ou DataSet. Prenons l'exemple des deux types de classe suivants :

```
C#
```

```
class Student
{
    public string First { get; set; }
    public string Last {get; set;}
    public int ID { get; set; }
    public string Street { get; set; }
    public string City { get; set; }
    public List<int> Scores;
}

class Teacher
{
    public string First { get; set; }
    public string Last { get; set; }
    public int ID { get; set; }
    public string City { get; set; }
}
```

L'exemple suivant présente la requête :

C#

```
class DataTransformations
{
    static void Main()
    {
        // Create the first data source.
        List<Student> students = new List<Student>()
        {
            new Student { First="Svetlana",
                Last="Omelchenko",
                ID=111,
                Street="123 Main Street",
                City="Seattle",
                Scores= new List<int> { 97, 92, 81, 60 } },
            new Student { First="Claire",
                Last="O'Donnell",
                ID=112,
                Street="124 Main Street",
                City="Redmond",
                Scores= new List<int> { 75, 84, 91, 39 } },
            new Student { First="Sven",
                Last="Mortensen",
                ID=113,
                Street="125 Main Street",
                City="Lake City",
                Scores= new List<int> { 88, 94, 65, 91 } },
        };

        // Create the second data source.
        List<Teacher> teachers = new List<Teacher>()
        {
```

```
            new Teacher { First="Ann", Last="Beebe", ID=945, City="Seattle"
},
            new Teacher { First="Alex", Last="Robinson", ID=956,
City="Redmond" },
            new Teacher { First="Michiyo", Last="Sato", ID=972,
City="Tacoma" }
        };

        // Create the query.
        var peopleInSeattle = (from student in students
                    where student.City == "Seattle"
                    select student.Last)
                    .Concat(from teacher in teachers
                            where teacher.City == "Seattle"
                            select teacher.Last);

        Console.WriteLine("The following students and teachers live in
Seattle:");
        // Execute the query.
        foreach (var person in peopleInSeattle)
        {
            Console.WriteLine(person);
        }

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    The following students and teachers live in Seattle:
    Omelchenko
    Beebe
 */
```

Pour plus d'informations, consultez join, clause et select, clause.

Sélection d'un sous-ensemble de chaque élément source

Pour sélectionner un sous-ensemble de chaque élément de la séquence source, deux méthodes principales s'offrent à vous :

1. Pour sélectionner un seul membre de l'élément source, utilisez l'opérateur point. Dans l'exemple suivant, supposons qu'un objet `Customer` contienne plusieurs propriétés publiques comprenant une chaîne nommée `City`. Une fois exécutée, cette requête génère une séquence de sortie de chaînes.

   ```
   C#
   ```

```
var query = from cust in Customers
            select cust.City;
```

2. Pour créer des éléments qui contiennent plusieurs propriétés de l'élément source, vous pouvez utiliser un initialiseur d'objet avec un objet nommé ou un type anonyme. L'exemple suivant montre comment utiliser un type anonyme pour encapsuler deux propriétés de chaque élément `Customer` :

```
C#

var query = from cust in Customer
            select new {Name = cust.Name, City = cust.City};
```

Pour plus d'informations, consultez Initialiseurs d'objets et de collections et Types anonymes.

Transformation d'objets en mémoire en XML

Les requêtes LINQ permettent de transformer facilement des données entre des structures de données en mémoire, des bases de données SQL, des jeux de données ADO.NET et des flux de données ou des documents XML. L'exemple suivant transforme des objets d'une structure de données en mémoire en éléments XML.

```
C#

class XMLTransform
{
    static void Main()
    {
        // Create the data source by using a collection initializer.
        // The Student class was defined previously in this topic.
        List<Student> students = new List<Student>()
        {
            new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores
= new List<int>{97, 92, 81, 60}},
            new Student {First="Claire", Last="O'Donnell", ID=112, Scores =
new List<int>{75, 84, 91, 39}},
            new Student {First="Sven", Last="Mortensen", ID=113, Scores =
new List<int>{88, 94, 65, 91}},
        };

        // Create the query.
        var studentsToXML = new XElement("Root",
            from student in students
            let scores = string.Join(",", student.Scores)
            select new XElement("student",
```

```
                        new XElement("First", student.First),
                        new XElement("Last", student.Last),
                        new XElement("Scores", scores)
                    ) // end "student"
                ); // end "Root"

        // Execute the query.
        Console.WriteLine(studentsToXML);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

Le code génère la sortie XML suivante :

XML

```
<Root>
  <student>
    <First>Svetlana</First>
    <Last>Omelchenko</Last>
    <Scores>97,92,81,60</Scores>
  </student>
  <student>
    <First>Claire</First>
    <Last>O'Donnell</Last>
    <Scores>75,84,91,39</Scores>
  </student>
  <student>
    <First>Sven</First>
    <Last>Mortensen</Last>
    <Scores>88,94,65,91</Scores>
  </student>
</Root>
```

Pour plus d'informations, consultez Création d'arborescences XML en C# (LINQ to XML).

Exécution d'opérations sur les éléments sources

Une séquence de sortie ne contient pas toujours des éléments ou des propriétés d'élément de la séquence source. La sortie peut être plutôt une séquence de valeurs calculée en utilisant les éléments sources comme arguments d'entrée.

La requête suivante prend une séquence de nombres qui représentent des rayons de cercles, calcule la zone pour chaque rayon et retourne une séquence de sortie contenant des chaînes mises en forme avec la zone calculée.

Chaque chaîne de la séquence de sortie sera mise en forme à l'aide de l'interpolation de chaîne. Une chaîne interpolée aura un `$` ouvrant de la chaîne et les opérations peuvent être effectuées dans des accolades placées à l'intérieur de la chaîne interpolée. Une fois ces opérations effectuées, les résultats sont concaténés.

> **Notes**
>
> Les méthodes d'appel dans les expressions de requête ne sont pas prises en charge si la requête est traduite dans un autre domaine. Par exemple, vous ne pouvez pas appeler une méthode C# ordinaire dans LINQ vers SQL, car SQL Server n'a pas de contexte pour cette méthode. Toutefois, vous pouvez mapper des procédures stockées à des méthodes et appeler celles-ci. Pour plus d'informations, consultez **Procédures stockées**.

C#

```
class FormatQuery
{
    static void Main()
    {
        // Data source.
        double[] radii = { 1, 2, 3 };

        // LINQ query using method syntax.
        IEnumerable<string> output =
            radii.Select(r => $"Area for a circle with a radius of '{r}' =
{r * r * Math.PI:F2}");

        /*
        // LINQ query using query syntax.
        IEnumerable<string> output =
            from rad in radii
            select $"Area for a circle with a radius of '{rad}' = {rad * rad
* Math.PI:F2}";
        */

        foreach (string s in output)
        {
            Console.WriteLine(s);
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Area for a circle with a radius of '1' = 3.14
    Area for a circle with a radius of '2' = 12.57
```

```
    Area for a circle with a radius of '3' = 28.27
*/
```

Voir aussi

- LINQ (Language-Integrated Query) (C#)
- LINQ to SQL
- LINQ to DataSet
- LINQ to XML (C#)
- Expressions de requête LINQ
- select, clause

Relations des types dans des opérations de requête LINQ (C#)

Article • 20/07/2023

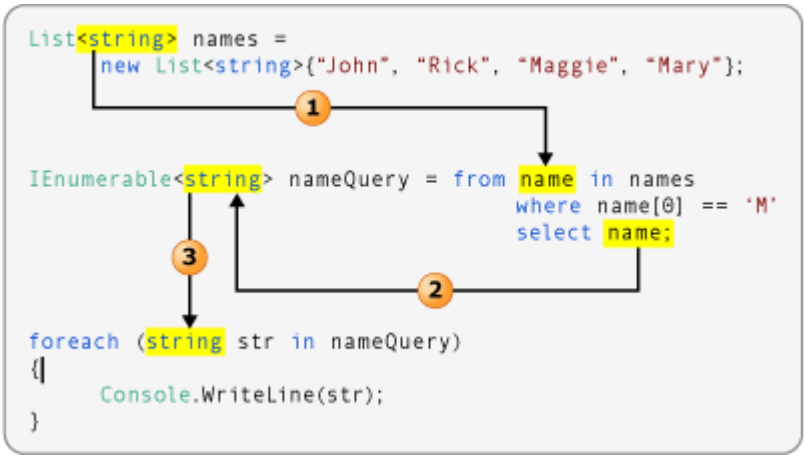
Pour écrire efficacement des requêtes, vous devez comprendre comment les types des variables dans une opération de requête complète sont liés les uns aux autres. Si vous comprenez ces relations, vous comprendrez plus facilement les exemples de code et les échantillons LINQ dans la documentation. En outre, vous comprendrez ce qui se passe en arrière-plan lorsque des variables sont implicitement typées à l'aide de `var`.

Les opérations de requête LINQ sont fortement typées dans la source de données, dans la requête elle-même et dans l'exécution de la requête. Le type des variables dans la requête doit être compatible avec le type des éléments dans la source de données, ainsi qu'avec le type de la variable d'itération dans l'instruction `foreach`. Ce typage fort garantit l'interception des erreurs de type au moment de la compilation lorsqu'elles peuvent être corrigées avant que les utilisateurs ne les rencontrent.

Pour illustrer ces relations de types, la plupart des exemples qui suivent utilisent le typage explicite pour toutes les variables. Le dernier exemple montre comment les mêmes principes s'appliquent même si vous utilisez le typage implicite à l'aide de var.

Requêtes qui ne transforment pas les données sources

L'illustration suivante montre une opération de requête LINQ to Objects qui n'effectue aucune transformation des données. La source contient une séquence de chaînes et le résultat de la requête est également une séquence de chaînes.



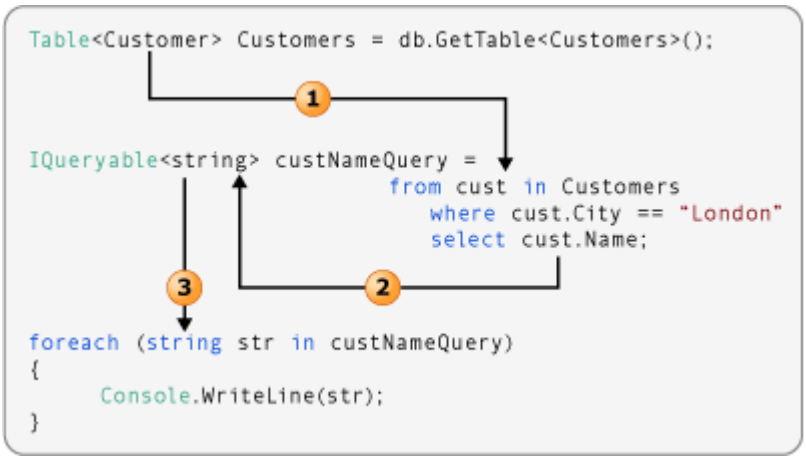
1. L'argument de type de la source de données détermine le type de la variable de portée.

2. Le type de l'objet sélectionné détermine le type de la variable de requête. Ici, `name` est une chaîne. Par conséquent, la variable de requête est un `IEnumerable<string>`.

3. La variable de requête fait l'objet d'une itération dans l'instruction `foreach`. Comme la variable de requête est une séquence de chaînes, la variable d'itération est également une chaîne.

Requêtes qui transforment les données sources

L'illustration suivante montre une opération de requête LINQ to SQL qui effectue une transformation simple des données. La requête accepte une séquence d'objets `Customer` en entrée et sélectionne uniquement la propriété `Name` dans le résultat. Comme `Name` est une chaîne, la requête produit une séquence de chaînes en sortie.

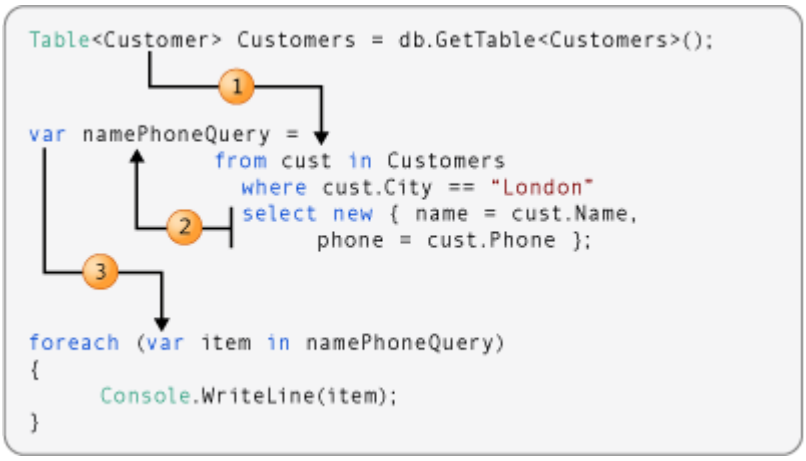


1. L'argument de type de la source de données détermine le type de la variable de portée.

2. L'instruction `select` retourne la propriété `Name` à la place de l'objet `Customer` complet. Comme `Name` est une chaîne, l'argument de type de `custNameQuery` est `string`, et non pas `Customer`.

3. Comme `custNameQuery` est une séquence de chaînes, la variable d'itération de la boucle `foreach` doit également être de type `string`.

L'illustration suivante montre une transformation légèrement plus complexe. L'instruction `select` retourne un type anonyme qui capture seulement deux membres de l'objet `Customer` original.



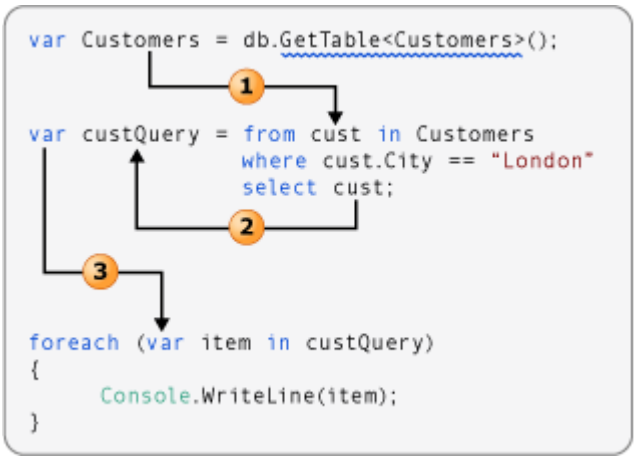
1. L'argument de type de la source de données est toujours le type de la variable de portée dans la requête.

2. Comme l'instruction `select` génère un type anonyme, la variable de requête doit être implicitement typée à l'aide de `var`.

3. Comme le type de la variable de requête est implicite, la variable d'itération dans la boucle `foreach` doit également être implicite.

Laisser le compilateur déduire les informations de type

Vous devez comprendre les relations des types dans une opération de requête. Toutefois, vous avez la possibilité de laisser le compilateur faire tout le travail à votre place. Le mot clé var peut être utilisé pour toute variable locale dans une opération de requête. L'illustration suivante est similaire à l'exemple numéro 2 qui a été abordé précédemment. Toutefois, le compilateur fournit le type fort pour chaque variable dans l'opération de requête.



Pour plus d'informations sur `var`, consultez Variables locales implicitement typées.

LINQ et types génériques (C#)

Les requêtes LINQ sont basées sur des types génériques. Vous ne devez pas avoir une connaissance approfondie des génériques avant de commencer à écrire des requêtes. Il peut cependant être important de comprendre deux concepts de base :

1. Quand vous créez une instance d'une classe de collection générique comme List<T>, vous remplacez le « T » par le type des objets contenus dans la liste. Par exemple, une liste de chaînes est exprimée sous la forme `List<string>`, et une liste d'objets `Customer` est exprimée sous la forme `List<Customer>`. Une liste générique est fortement typée et offre de nombreux avantages par rapport aux collections qui stockent leurs éléments en tant que Object. Si vous essayez d'ajouter un `Customer` à un `List<string>`, vous obtenez une erreur à la compilation. Il est facile d'utiliser des collections génériques, car vous ne devez pas effectuer un cast de type à l'exécution.

2. IEnumerable<T> est l'interface qui permet l'énumération des classes de collection génériques avec l'instruction `foreach`. Les classes de collection génériques prennent en charge IEnumerable<T> de la même façon que les classes de collection non génériques telles que ArrayList prennent en charge IEnumerable.

Pour plus d'informations sur les génériques, consultez Génériques.

Variables IEnumerable<T> dans les requêtes LINQ

Les variables de requête LINQ sont typées en tant que IEnumerable<T> ou un type dérivé tel que IQueryable<T>. Quand vous voyez une variable de requête typée en `IEnumerable<Customer>`, cela signifie simplement que la requête génère à l'exécution une séquence de zéro ou plusieurs objets `Customer`.

C#

```
IEnumerable<Customer> customerQuery =
    from cust in customers
    where cust.City == "London"
    select cust;

foreach (Customer customer in customerQuery)
{
    Console.WriteLine(customer.LastName + ", " + customer.FirstName);
}
```

Pour plus d'informations, consultez Relations des types dans des opérations de requête LINQ.

Laisser le compilateur gérer les déclarations de type générique

Si vous préférez, vous pouvez éviter une syntaxe générique en utilisant le mot clé var. Le mot clé `var` indique au compilateur d'inférer le type d'une variable de requête en examinant la source de données spécifiée dans la clause `from`. L'exemple suivant génère le même code compilé que l'exemple précédent :

C#

```
var customerQuery2 =
    from cust in customers
    where cust.City == "London"
    select cust;

foreach(var customer in customerQuery2)
{
    Console.WriteLine(customer.LastName + ", " + customer.FirstName);
}
```

Le mot clé `var` est utile quand le type de la variable est évident ou quand il est particulièrement important de spécifier explicitement des types génériques imbriqués, comme ceux qui sont produits par des requêtes de groupe. D'une façon générale, si vous utilisez `var`, sachez qu'il peut rendre votre code plus difficile à lire par d'autres développeurs. Pour plus d'informations, consultez Variables locales implicitement typées.

Écrire des requêtes LINQ en C#

Article • 20/07/2023

La plupart des requêtes de la documentation de présentation de Langage Integrated Query (LINQ) sont écrites à l'aide de la syntaxe de requête déclarative de LINQ. Toutefois, la syntaxe de requête doit être traduite en appels de méthode pour le Common Langage Runtime (CLR) .NET lorsque le code est compilé. Ces appels de méthode appellent les opérateurs de requête standard, qui ont des noms tels que `Where`, `Select`, `GroupBy`, `Join`, `Max` et `Average`. Vous pouvez les appeler directement en utilisant la syntaxe de méthode à la place de la syntaxe de requête.

La syntaxe de requête et la syntaxe de méthode sont identiques sémantiquement, mais de nombreuses personnes trouvent la syntaxe de requête plus simple et plus facile à lire. Certaines requêtes doivent être exprimées en tant qu'appels de méthode. Par exemple, vous devez utiliser un appel de méthode pour exprimer une requête qui récupère le nombre d'éléments qui correspondent à une condition spécifiée. Vous devez également utiliser un appel de méthode pour une requête qui récupère dans une séquence source l'élément qui a la valeur maximale. En général, la documentation de référence des opérateurs de requête standard dans l'espace de noms System.Linq utilise la syntaxe de méthode. Par conséquent, même lorsque vous commencez à écrire des requêtes LINQ, il est utile de se familiariser avec l'utilisation de la syntaxe de méthode dans les requêtes et dans les expressions de requête elles-mêmes.

Méthodes d'extension d'opérateur de requête standard

L'exemple suivant présente une *expression de requête* simple et la requête sémantiquement équivalente écrite en tant que *requête fondée sur une méthode*.

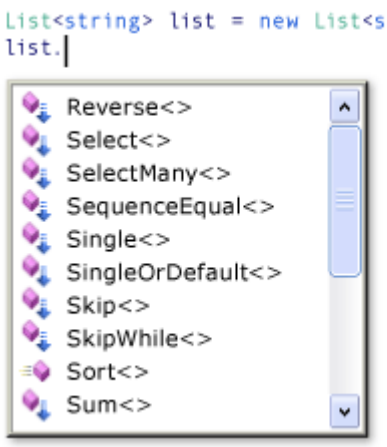
C#

```
class QueryVMethodSyntax
{
    static void Main()
    {
        int[] numbers = { 5, 10, 8, 3, 6, 12};

        //Query syntax:
        IEnumerable<int> numQuery1 =
            from num in numbers
            where num % 2 == 0
            orderby num
            select num;
```

```
        //Method syntax:
        IEnumerable<int> numQuery2 = numbers.Where(num => num % 2 ==
0).OrderBy(n => n);

        foreach (int i in numQuery1)
        {
            Console.Write(i + " ");
        }
        Console.WriteLine(System.Environment.NewLine);
        foreach (int i in numQuery2)
        {
            Console.Write(i + " ");
        }

        // Keep the console open in debug mode.
        Console.WriteLine(System.Environment.NewLine);
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/*
    Output:
    6 8 10 12
    6 8 10 12
 */
```

La sortie des deux exemples est identique. Vous pouvez voir que le type de la variable de requête est le même dans les deux formes : IEnumerable<T>.

Pour comprendre la requête basée sur une méthode, examinons-la de plus près. Du côté droit de l'expression, remarquez que la clause `where` est maintenant exprimée comme une méthode d'instance sur l'objet `numbers`, qui, comme vous vous en rappelez, a un type `IEnumerable<int>`. Si vous êtes connaissez bien l'interface IEnumerable<T> générique, vous savez qu'elle n'a pas de méthode `Where`. Toutefois, si vous appelez la liste de saisie semi-automatique IntelliSense dans l'IDE de Visual Studio, vous ne verrez pas seulement une méthode `Where`, mais de nombreuses autres méthodes telles que `Select`, `SelectMany`, `Join` et `Orderby`. Il s'agit de tous les opérateurs de requête standard.



Même s'il semble que IEnumerable<T> a été redéfini pour inclure ces méthodes supplémentaires, ce n'est pas le cas, en réalité. Les opérateurs de requête standard sont implémentés comme un nouveau type de méthode, appelé *méthodes d'extension*. Les méthodes d'extension « étendent » un type existant. Elles peuvent être appelées comme s'il s'agissait de méthodes d'instance sur le type. Les opérateurs de requête standard étendent IEnumerable<T>, si bien que vous pouvez écrire `numbers.Where(...)`.

Pour commencer à utiliser LINQ, tout ce que vous devez savoir sur les méthodes d'extension est la façon de les mettre à portée de votre application à l'aide des directives `using` appropriées. Du point de vue de votre application, une méthode d'extension et une méthode d'instance normale sont identiques.

Pour plus d'informations sur les méthodes d'extension, consultez Méthodes d'extension. Pour plus d'informations sur les opérateurs de requête standard, consultez Présentation des opérateurs de requête standard (C#). Certains fournisseurs LINQ, tels que LINQ to SQL ET LINQ to XML, implémentent leurs propres opérateurs de requête standard et des méthodes d'extension supplémentaires pour d'autres types que IEnumerable<T>.

Expressions lambda

Dans l'exemple précédent, notez que l'expression conditionnelle (`num % 2 == 0`) est passée comme argument de ligne à la méthode `Where` : `Where(num => num % 2 == 0).` Cette expression inline est appelée expression lambda. C'est un moyen pratique d'écrire du code qui devrait autrement être écrit sous une forme plus fastidieuse, comme une méthode anonyme, un délégué générique ou une arborescence d'expressions. En C#, `=>` est l'opérateur lambda, qui se lit « conduit à ». Le `num` situé à gauche de l'opérateur est la variable d'entrée qui correspond à `num` dans l'expression de requête. Le compilateur peut déduire le type de `num`, car il sait que `numbers` est un type IEnumerable<T> générique. Le corps de l'expression lambda est identique à l'expression dans la syntaxe de requête ou dans toute autre expression ou instruction C#. Il peut

inclure des appels de méthode et une autre logique complexe. La « valeur de retour » est simplement le résultat de l'expression.

Vous n'avez pas besoin d'utiliser fréquemment les expressions lambda pour commencer à utiliser LINQ. Toutefois, certaines requêtes peuvent être exprimées uniquement dans la syntaxe de méthode. Parmi elles, certaines nécessitent des expressions lambda. Une fois que vous serez plus familiarisé avec les expressions lambda, vous constaterez qu'elles sont un outil puissant et flexible dans votre boîte à outils LINQ. Pour plus d'informations, consultez Expressions lambda.

Composabilité des requêtes

Dans l'exemple de code précédent, notez que la méthode `OrderBy` est appelée à l'aide de l'opérateur point sur l'appel à `Where`. `Where` produit une séquence filtrée, puis `Orderby` agit sur cette séquence en la triant. Étant donné que les requêtes retournent un `IEnumerable`, vous les composez dans la syntaxe de méthode en chaînant les appels de méthode ensemble. Le compilateur effectue cette opération en arrière-plan lorsque vous écrivez des requêtes à l'aide de la syntaxe de requête. De plus, étant donné qu'une variable de requête ne stocke pas les résultats de la requête, vous pouvez la modifier ou l'utiliser à tout moment comme base d'une nouvelle requête, même après son exécution.

Les exemples suivants montrent des requêtes LINQ simples en utilisant chaque approche répertoriée précédemment. En général, la règle consiste à utiliser (1) dès que possible et à utiliser (2) et (3) dès que cela est nécessaire.

> ⓘ **Notes**
>
> Ces requêtes fonctionnent sur les collections simples en mémoire ; toutefois, la syntaxe de base est identique à celle utilisée dans LINQ to Entities et LINQ to XML.

Exemple – Syntaxe de requête

La méthode recommandée pour écrire la plupart des requêtes consiste à utiliser la *syntaxe de requête* pour créer des *expressions de requête*. L'exemple suivant présente trois expressions de requête. La première expression de requête montre comment filtrer ou restreindre des résultats en appliquant des conditions avec une clause `where`. Tous les éléments de la séquence source dont la valeur est supérieure à 7 ou inférieure à 3 sont retournés. La deuxième expression montre comment classer les résultats retournés.

La troisième expression montre comment regrouper des résultats en fonction d'une clé. Cette requête retourne deux groupes en fonction de la première lettre du mot.

C#

```
List<int> numbers = new() { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

// The query variables can also be implicitly typed by using var

// Query #1.
IEnumerable<int> filteringQuery =
    from num in numbers
    where num < 3 || num > 7
    select num;

// Query #2.
IEnumerable<int> orderingQuery =
    from num in numbers
    where num < 3 || num > 7
    orderby num ascending
    select num;

// Query #3.
string[] groupingQuery = { "carrots", "cabbage", "broccoli", "beans",
"barley" };
IEnumerable<IGrouping<char, string>> queryFoodGroups =
    from item in groupingQuery
    group item by item[0];
```

Notez que le type des requêtes est IEnumerable<T>. Toutes ces requêtes pourraient être écrites à l'aide de `var`, comme illustré dans l'exemple suivant :

```
var query = from num in numbers...
```

Dans chacun des exemples précédents, les requêtes ne s'exécutent pas réellement tant vous n'avez pas itéré la variable de requête dans une instruction `foreach` ou une autre instruction. Pour plus d'informations, consultez Introduction aux requêtes LINQ.

Exemple – Syntaxe de méthode

Certaines opérations de requête doivent être exprimées comme un appel de méthode. Les plus répandues de ces méthodes retournent des valeurs numériques uniques, telles que Sum, Max, Min, Average, et ainsi de suite. Ces méthodes doivent toujours être appelées en dernier dans toutes les requêtes, car elles ne représentent qu'une valeur unique et ne peuvent pas servir de source pour une opération de requête supplémentaire. L'exemple suivant présente un appel de méthode dans une expression de requête :

C#

```
List<int> numbers1 = new() { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
List<int> numbers2 = new() { 15, 14, 11, 13, 19, 18, 16, 17, 12, 10 };

// Query #4.
double average = numbers1.Average();

// Query #5.
IEnumerable<int> concatenationQuery = numbers1.Concat(numbers2);
```

Si la méthode a des paramètres Action ou Func, ceux-ci sont fournis sous la forme d'une expression lambda, comme dans l'exemple suivant :

C#

```
// Query #6.
IEnumerable<int> largeNumbersQuery = numbers2.Where(c => c > 15);
```

Dans les requêtes précédentes, seule la requête 4 s'exécute immédiatement. Cela s'explique par le fait qu'elle retourne une valeur unique et non une collection IEnumerable<T> générique. La méthode elle-même doit utiliser `foreach` pour calculer sa valeur.

Chacune des requêtes précédentes peut être écrite en utilisant des types implicites avec var, comme dans l'exemple suivant :

C#

```
// var is used for convenience in these queries
var average = numbers1.Average();
var concatenationQuery = numbers1.Concat(numbers2);
var largeNumbersQuery = numbers2.Where(c => c > 15);
```

Exemple – Syntaxe de méthode et de requête mixte

Cet exemple montre comment utiliser la syntaxe de méthode sur les résultats d'une clause de requête. Encadrez simplement l'expression de requête entre parenthèses, puis appliquez l'opérateur point et appelez la méthode. Dans l'exemple suivant, la requête 7 retourne les nombres dont la valeur est comprise entre 3 et 7. Toutefois, il est généralement préférable d'utiliser une deuxième variable pour stocker le résultat de

l'appel de méthode. De cette manière, il est moins probable que la requête ne soit confondue avec les résultats de la requête.

C#

```
// Query #7.

// Using a query expression with method syntax
int numCount1 = (
    from num in numbers1
    where num < 3 || num > 7
    select num
).Count();

// Better: Create a new variable to store
// the method call result
IEnumerable<int> numbersQuery =
    from num in numbers1
    where num < 3 || num > 7
    select num;

int numCount2 = numbersQuery.Count();
```

Comme la requête 7 retourne une valeur unique et non une collection, la requête s'exécute immédiatement.

La requête précédente peut être écrite en utilisant des types implicites avec `var`, comme suit :

C#

```
var numCount = (from num in numbers...
```

Elle peut être écrite dans la syntaxe de méthode comme suit :

C#

```
var numCount = numbers.Where(n => n < 3 || n > 7).Count();
```

Elle peut être écrite en utilisant des types explicites, comme suit :

C#

```
int numCount = numbers.Where(n => n < 3 || n > 7).Count();
```

Voir aussi

- Procédure pas à pas : écriture de requêtes en C#
- LINQ (Language-Integrated Query)
- where, clause

Fonctionnalités C# qui prennent en charge LINQ

Article • 09/03/2023

Ces nouvelles fonctionnalités sont toutes plus ou moins utilisées avec les requêtes LINQ, elles ne sont pas limitées à LINQ et peuvent être utilisées dans n'importe quel contexte.

Expressions de requête

Les expressions de requête utilisent une syntaxe déclarative similaire à SQL ou XQuery pour interroger les collections IEnumerable. Au moment de la compilation, la syntaxe de requête est convertie en appels de méthode à l'implémentation de méthodes d'extension d'opérateur de requête standard d'un fournisseur LINQ. Les applications contrôlent les opérateurs de requête standard qui se trouvent dans la portée en spécifiant l'espace de noms approprié avec une directive `using`. L'expression de requête suivante accepte un tableau de chaînes, regroupe les chaînes qui commencent par le même caractère, puis trie ces groupes de chaînes.

C#

```
var query = from str in stringArray
            group str by str[0] into stringGroup
            orderby stringGroup.Key
            select stringGroup;
```

Pour plus d'informations, consultez Expressions de requête LINQ.

Variables implicitement typées (var)

Au lieu de spécifier explicitement un type lorsque vous déclarez et initialisez une variable, vous pouvez utiliser le modificateur var pour indiquer au compilateur qu'il doit déduire et assigner le type, comme illustré ici :

C#

```
var number = 5;
var name = "Virginia";
var query = from str in stringArray
            where str[0] == 'm'
            select str;
```

Les variables déclarées comme `var` sont aussi fortement typées que les variables dont vous spécifiez explicitement le type. `var` permet de créer des types anonymes, mais peut être utilisé uniquement pour les variables locales. Les tableaux peuvent également être déclarés avec un typage implicite.

Pour plus d'informations, consultez Variables locales implicitement typées.

Initialiseurs d'objets et de collections

Les initialiseurs d'objets et de collections permettent d'initialiser un objet sans appeler explicitement un constructeur pour cet objet. Les initialiseurs sont généralement utilisés dans les expressions de requête pour projeter la source de données en un nouveau type de données. En supposant une classe nommée `Customer` avec les propriétés publiques `Name` et `Phone`, l'initialiseur d'objet peut être utilisé, comme dans le code suivant :

```
C#

var cust = new Customer { Name = "Mike", Phone = "555-1212" };
```

Si l'on reprend notre classe `Customer`, supposons qu'il existe une source de données appelée `IncomingOrders` et que, pour chaque commande caractérisée par un grand `OrderSize`, nous souhaitons créer un nouveau `Customer` basé sur cette commande. Une requête LINQ peut être exécutée sur cette source de données et utiliser l'initialisation d'objets pour remplir une collection :

```
C#

var newLargeOrderCustomers = from o in IncomingOrders
                             where o.OrderSize > 5
                             select new Customer { Name = o.Name, Phone =
o.Phone };
```

La source de données peut avoir plus de propriétés sous-jacentes que la classe `Customer`, par exemple, `OrderSize` ; cependant, avec l'initialisation d'objets, les données retournées par la requête sont moulées dans le type de données souhaité ; nous choisissons les données correspondant à notre classe. Par conséquent, nous disposons désormais d'un `IEnumerable` contenant les nouveaux `Customer` que nous voulions. La requête ci-dessus peut également s'écrire avec la syntaxe de méthode LINQ :

```
C#
```

```
var newLargeOrderCustomers = IncomingOrders.Where(x => x.OrderSize >
5).Select(y => new Customer { Name = y.Name, Phone = y.Phone });
```

Pour plus d'informations, consultez les pages suivantes :

- Initialiseurs d'objets et de collections
- Syntaxe des expressions de requête pour les opérateurs de requête standard

Types anonymes

Un type anonyme est construit par le compilateur et le nom du type est uniquement disponible pour le compilateur. Les types anonymes permettent de regrouper facilement des propriétés de manière temporaire dans un résultat de requête, sans avoir à définir un type nommé distinct. Les types anonymes sont initialisés avec une nouvelle expression et un initialiseur d'objet, comme indiqué ici :

C#

```
select new {name = cust.Name, phone = cust.Phone};
```

Pour plus d'informations, consultez Types anonymes.

Méthodes d'extension

Une méthode d'extension est une méthode statique qui peut être associée à un type, de manière à être appelée comme une méthode d'instance de ce type. Cette fonctionnalité permet d'ajouter de nouvelles méthodes aux types existants sans les modifier. Les opérateurs de requête standard sont un ensemble de méthodes d'extension qui fournissent des fonctionnalités de requête LINQ pour tout type qui implémente IEnumerable<T>.

Pour plus d'informations, consultez Méthodes d'extension.

Expressions lambda

Une expression lambda est une fonction inline qui utilise l'opérateur `=>` pour séparer les paramètres d'entrée du corps de la fonction, et qui peut être convertie au moment de la compilation en un délégué ou une arborescence d'expressions. En programmation

LINQ, vous rencontrerez des expressions lambda quand vous effectuerez des appels de méthode directs aux opérateurs de requête standard.

Pour plus d'informations, consultez les pages suivantes :

- Expressions lambda
- Arborescences d’expressions (C#)

Voir aussi

- LINQ (Language-Integrated Query) (C#)

Procédure pas à pas : Écrire des requêtes en C# (LINQ)

Article • 07/04/2023

Cette procédure pas à pas présente les fonctionnalités du langage C# utilisées pour écrire des expressions de requête LINQ.

Créer un projet C#

> **Notes**
>
> Les instructions suivantes s'appliquent à Visual Studio. Si vous utilisez un environnement de développement différent, créez un projet console avec une référence à System.Core.dll et une directive `using` pour l'espace de noms **System.Linq**.

Pour créer un projet dans Visual Studio

1. Démarrez Visual Studio.
2. Dans le menu principal, sélectionnez **Fichier**, **Nouveau**, **Projet**.

   La boîte de dialogue **Nouveau projet** s'affiche.
3. Développez **Installé**, **Modèles**, **Visual C#**, puis choisissez **Application console**.
4. Dans la zone de texte **Nom**, entrez un nom différent ou acceptez le nom par défaut, puis choisissez le bouton **OK**.

   Le nouveau projet s'affiche dans l'**Explorateur de solutions**.
5. Notez que votre projet a une référence à System.Core.dll et une directive `using` pour l'espace de noms System.Linq.

Créer une source de données en mémoire

La source de données pour les requêtes est une simple liste d'objets `Student`. Chaque enregistrement `Student` comporte un prénom, un nom et un tableau d'entiers

représentant les notes d'examens en classe. Copiez ce code dans votre projet. Notez les caractéristiques suivantes :

- La classe `Student` se compose de propriétés implémentées automatiquement.
- Chaque étudiant de la liste est initialisé avec un initialiseur d'objet.
- La liste elle-même est initialisée avec un initialiseur de collection.

La totalité de cette structure de données est initialisée et instanciée sans appels explicites à un constructeur quelconque ni accès au membre explicite. Pour plus d'informations sur ces nouvelles fonctionnalités, consultez Propriétés implémentées automatiquement et Initialiseurs d'objets et de collection.

Pour ajouter la source de données

- Ajoutez la classe `Student` et la liste initialisée d'étudiants à la classe `Program` dans votre projet.

```
C#

public class Student
{
    public string First { get; set; }
    public string Last { get; set; }
    public int ID { get; set; }
    public List<int> Scores;
}

// Create a data source by using a collection initializer.
static List<Student> students = new List<Student>
{
    new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores=
new List<int> {97, 92, 81, 60}},
    new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new
List<int> {75, 84, 91, 39}},
    new Student {First="Sven", Last="Mortensen", ID=113, Scores= new
List<int> {88, 94, 65, 91}},
    new Student {First="Cesar", Last="Garcia", ID=114, Scores= new
List<int> {97, 89, 85, 82}},
    new Student {First="Debra", Last="Garcia", ID=115, Scores= new
List<int> {35, 72, 91, 70}},
    new Student {First="Fadi", Last="Fakhouri", ID=116, Scores= new
List<int> {99, 86, 90, 94}},
    new Student {First="Hanying", Last="Feng", ID=117, Scores= new
List<int> {93, 92, 80, 87}},
    new Student {First="Hugo", Last="Garcia", ID=118, Scores= new
List<int> {92, 90, 83, 78}},
    new Student {First="Lance", Last="Tucker", ID=119, Scores= new
List<int> {68, 79, 88, 92}},
```

```
    new Student {First="Terry", Last="Adams", ID=120, Scores= new
List<int> {99, 82, 81, 79}},
    new Student {First="Eugene", Last="Zabokritski", ID=121, Scores=
new List<int> {96, 85, 91, 60}},
    new Student {First="Michael", Last="Tucker", ID=122, Scores= new
List<int> {94, 92, 91, 91}}
};
```

Pour ajouter un nouvel étudiant à la liste des étudiants

1. Ajoutez un nouvel élément `Student` à la liste `Students` et utilisez le nom et les notes d'examens de votre choix. Essayez d'entrer toutes les informations relatives au nouvel étudiant pour mieux apprendre la syntaxe de l'initialiseur d'objet.

Créer la requête

Pour créer une requête simple

- Dans la méthode `Main` de l'application, créez une requête simple qui, quand elle est exécutée, produit une liste de tous les étudiants dont la note pour le premier test est supérieure à 90. Notez que, puisque l'objet `Student` entier est sélectionné, le type de la requête est `IEnumerable<Student>`. Bien que le code puisse également utiliser un type implicite à l'aide du mot clé var, les types explicites sont utilisés pour illustrer clairement les résultats. (Pour plus d'informations sur `var`, consultez Variables locales implicitement typées.)

  Notez également que la variable de portée de la requête, `student`, sert de référence à chaque `Student` dans la source, en fournissant l'accès au membre pour chaque objet.

C#

```
// Create the query.
// The first line could also be written as "var studentQuery ="
IEnumerable<Student> studentQuery =
    from student in students
    where student.Scores[0] > 90
    select student;
```

Exécuter la requête

Pour exécuter la requête

1. Écrivez maintenant la boucle `foreach` qui entraînera l'exécution de la requête. Notez les points suivants concernant le code :

   - Chaque élément de la séquence retournée est accessible via la variable d'itération dans la boucle `foreach`.

   - Le type de cette variable est `Student` et le type de la variable de requête est compatible, `IEnumerable<Student>`.

2. Après avoir ajouté ce code, générez et exécutez l'application pour visualiser les résultats dans la fenêtre de **console**.

C#

```
// Execute the query.
// var could be used here also.
foreach (Student student in studentQuery)
{
    Console.WriteLine("{0}, {1}", student.Last, student.First);
}

// Output:
// Omelchenko, Svetlana
// Garcia, Cesar
// Fakhouri, Fadi
// Feng, Hanying
// Garcia, Hugo
// Adams, Terry
// Zabokritski, Eugene
// Tucker, Michael
```

Pour ajouter une autre condition de filtre

1. Vous pouvez associer plusieurs conditions booléennes dans la clause `where` pour affiner davantage une requête. Le code suivant ajoute une condition afin que la requête retourne les étudiants dont la première note est supérieure à 90 et dont la dernière est inférieure à 80. La clause `where` doit être semblable au code suivant.

   C#

   ```
   where student.Scores[0] > 90 && student.Scores[3] < 80
   ```

   Pour plus d'informations, consultez where, clause.

Modifier la requête

Pour classer les résultats

1. Il est plus facile d'analyser les résultats s'ils sont classés. Vous pouvez classer la séquence retournée selon tous les champs accessibles dans les éléments sources. Par exemple, la clause `orderby` suivante classe les résultats dans l'ordre alphabétique de A à Z d'après le nom de chaque étudiant. Ajoutez la clause `orderby` suivante à votre requête, juste après l'instruction `where` et avant l'instruction `select` :

   C#

   ```
   orderby student.Last ascending
   ```

2. Modifiez à présent la clause `orderby` pour qu'elle classe les résultats dans l'ordre inverse d'après la note au premier examen, de la plus élevée à la plus faible.

   C#

   ```
   orderby student.Scores[0] descending
   ```

3. Modifiez la chaîne de format `WriteLine` pour pouvoir consulter les notes :

   C#

   ```
   Console.WriteLine("{0}, {1} {2}", student.Last, student.First,
   student.Scores[0]);
   ```

   Pour plus d'informations, consultez orderby, clause.

Pour regrouper les résultats

1. Le regroupement est une fonction puissante des expressions de requête. Une requête avec une clause group produit une séquence de groupes dont chaque groupe contient lui-même un élément `Key` et une séquence composée de tous les membres de ce groupe. La nouvelle requête suivante regroupe les étudiants en utilisant la première lettre de leur nom comme clé.

   C#

```
IEnumerable<IGrouping<char, Student>> studentQuery2 =
    from student in students
    group student by student.Last[0];
```

2. Notez que le type de la requête a maintenant changé. Elle produit désormais une séquence de groupes qui ont un type `char` comme clé et une séquence d'objets `Student`. Étant donné que le type de la requête a changé, le code suivant modifie également la boucle d'exécution `foreach` :

```
C#

foreach (IGrouping<char, Student> studentGroup in studentQuery2)
{
    Console.WriteLine(studentGroup.Key);
    foreach (Student student in studentGroup)
    {
        Console.WriteLine("   {0}, {1}",
                  student.Last, student.First);
    }
}

// Output:
// O
//   Omelchenko, Svetlana
//   O'Donnell, Claire
// M
//   Mortensen, Sven
// G
//   Garcia, Cesar
//   Garcia, Debra
//   Garcia, Hugo
// F
//   Fakhouri, Fadi
//   Feng, Hanying
// T
//   Tucker, Lance
//   Tucker, Michael
// A
//   Adams, Terry
// Z
//   Zabokritski, Eugene
```

3. Exécutez l'application et affichez les résultats dans la fenêtre de **console**.

   Pour plus d'informations, consultez group, clause.

Pour rendre les variables implicitement typées

1. Le codage explicite de `IEnumerables` de `IGroupings` peut rapidement s'avérer laborieux. Vous pouvez écrire la même requête et la même boucle `foreach` beaucoup plus facilement à l'aide de `var`. Le mot clé `var` ne modifie pas les types de vos objets ; il indique simplement au compilateur de déduire les types. Remplacez le type de `studentQuery` et la variable d'itération `group` par `var` et réexécutez la requête. Notez que, dans la boucle `foreach` interne, la variable d'itération est encore typée comme `Student` et que la requête fonctionne exactement comme précédemment. Remplacez la variable d'itération `student` par `var` et exécutez à nouveau la requête. Vous pouvez constater que les résultats obtenus sont exactement les mêmes.

```
C#
```

```
var studentQuery3 =
    from student in students
    group student by student.Last[0];

foreach (var groupOfStudents in studentQuery3)
{
    Console.WriteLine(groupOfStudents.Key);
    foreach (var student in groupOfStudents)
    {
        Console.WriteLine("   {0}, {1}",
            student.Last, student.First);
    }
}

// Output:
// O
//   Omelchenko, Svetlana
//   O'Donnell, Claire
// M
//   Mortensen, Sven
// G
//   Garcia, Cesar
//   Garcia, Debra
//   Garcia, Hugo
// F
//   Fakhouri, Fadi
//   Feng, Hanying
// T
//   Tucker, Lance
//   Tucker, Michael
// A
//   Adams, Terry
// Z
//   Zabokritski, Eugene
```

Pour plus d'informations sur var, consultez Variables locales implicitement typées.

Pour classer les groupes selon leur valeur de clé

1. Quand vous exécutez la requête précédente, vous remarquez que les groupes ne sont pas dans l'ordre alphabétique. Pour modifier cet ordre, vous devez fournir une clause `orderby` après la clause `group`. Toutefois, pour utiliser une clause `orderby`, vous avez d'abord besoin d'un identificateur servant de référence aux groupes créés par la clause `group`. Fournissez l'identificateur à l'aide du mot clé `into`, comme suit :

```
C#
```

```
var studentQuery4 =
    from student in students
    group student by student.Last[0] into studentGroup
    orderby studentGroup.Key
    select studentGroup;

foreach (var groupOfStudents in studentQuery4)
{
    Console.WriteLine(groupOfStudents.Key);
    foreach (var student in groupOfStudents)
    {
        Console.WriteLine("   {0}, {1}",
            student.Last, student.First);
    }
}

// Output:
//A
//   Adams, Terry
//F
//   Fakhouri, Fadi
//   Feng, Hanying
//G
//   Garcia, Cesar
//   Garcia, Debra
//   Garcia, Hugo
//M
//   Mortensen, Sven
//O
//   Omelchenko, Svetlana
//   O'Donnell, Claire
//T
//   Tucker, Lance
//   Tucker, Michael
//Z
//   Zabokritski, Eugene
```

Quand vous exécutez cette requête, vous voyez que les groupes sont désormais classés par ordre alphabétique.

Pour introduire un identificateur à l'aide de let

1. Vous pouvez utiliser le mot clé `let` pour introduire un identificateur pour tous les résultats d'expression dans l'expression de requête. Cet identificateur peut être pratique, comme dans l'exemple suivant, ou améliorer les performances en stockant les résultats d'une expression afin d'éviter qu'ils ne soient calculés plusieurs fois.

```
C#
```

```
// studentQuery5 is an IEnumerable<string>
// This query returns those students whose
// first test score was higher than their
// average score.
var studentQuery5 =
    from student in students
    let totalScore = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    where totalScore / 4 < student.Scores[0]
    select student.Last + " " + student.First;

foreach (string s in studentQuery5)
{
    Console.WriteLine(s);
}

// Output:
// Omelchenko Svetlana
// O'Donnell Claire
// Mortensen Sven
// Garcia Cesar
// Fakhouri Fadi
// Feng Hanying
// Garcia Hugo
// Adams Terry
// Zabokritski Eugene
// Tucker Michael
```

Pour plus d'informations, consultez let, clause.

Pour utiliser la syntaxe de méthode dans une expression de requête

1. Comme décrit dans Syntaxe de requête et syntaxe de méthode dans LINQ, certaines opérations de requête peuvent être exprimées uniquement à l'aide de la syntaxe de méthode. Le code suivant calcule la note totale de chaque `Student` dans la séquence source, puis appelle la méthode `Average()` sur les résultats de cette requête pour calculer la note moyenne de la classe.

C#

```
var studentQuery6 =
    from student in students
    let totalScore = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    select totalScore;

double averageScore = studentQuery6.Average();
Console.WriteLine("Class average score = {0}", averageScore);

// Output:
// Class average score = 334.166666666667
```

Pour transformer ou projeter la clause select

1. Il n'est pas rare qu'une requête produise une séquence dont les éléments diffèrent des éléments des séquences sources. Supprimez ou commentez votre requête et votre boucle d'exécution précédentes, et remplacez-les par le code suivant. Notez que la requête retourne une séquence de chaînes (pas `Students`) et que ce fait est répercuté dans la boucle `foreach`.

   C#

   ```
   IEnumerable<string> studentQuery7 =
       from student in students
       where student.Last == "Garcia"
       select student.First;

   Console.WriteLine("The Garcias in the class are:");
   foreach (string s in studentQuery7)
   {
       Console.WriteLine(s);
   }

   // Output:
   // The Garcias in the class are:
   // Cesar
   // Debra
   // Hugo
   ```

2. Le code utilisé précédemment dans cette procédure pas à pas indique que la note moyenne de la classe est d'environ 334. Pour produire une séquence de `Students` dont la note totale est supérieure à la moyenne de classe, avec les `Student ID` correspondants, vous pouvez utiliser un type anonyme dans l'instruction `select` :

   C#

```
var studentQuery8 =
    from student in students
    let x = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    where x > averageScore
    select new { id = student.ID, score = x };

foreach (var item in studentQuery8)
{
    Console.WriteLine("Student ID: {0}, Score: {1}", item.id,
item.score);
}

// Output:
// Student ID: 113, Score: 338
// Student ID: 114, Score: 353
// Student ID: 116, Score: 369
// Student ID: 117, Score: 352
// Student ID: 118, Score: 343
// Student ID: 120, Score: 341
// Student ID: 122, Score: 368
```

Étapes suivantes

Une fois que vous êtes familiarisé avec les aspects de base de l'utilisation de requêtes en C#, vous êtes prêt à lire la documentation et les exemples pour le type spécifique de fournisseur LINQ qui vous intéresse :

LINQ to SQL

LINQ to DataSet

LINQ to XML (C#)

LINQ to Objects (C#)

Voir aussi

- LINQ (Language-Integrated Query) (C#)
- Expressions de requête LINQ

Présentation des opérateurs de requête standard (C#)

Article • 20/07/2023

Les *opérateurs de requête standard* sont les méthodes qui forment le modèle de requête LINQ. La plupart de ces méthodes fonctionnent sur des séquences, où une séquence est un objet dont le type implémente l'interface IEnumerable<T> ou l'interface IQueryable<T>. Les opérateurs de requête standard fournissent des fonctionnalités de requête, dont notamment le filtrage, la projection, l'agrégation, le tri, etc.

Il existe deux ensembles d'opérateurs de requête standard LINQ : l'un opérant sur des objets de type IEnumerable<T> et l'autre sur des objets de type IQueryable<T>. Les méthodes qui composent chaque ensemble sont des membres statiques des classes Enumerable et Queryable, respectivement. Elles sont définies en tant que *méthodes d'extension* du type sur lequel elles opèrent. Les méthodes d'extension peuvent être appelées à l'aide de la syntaxe de méthode statique ou de la syntaxe de méthode d'instance.

De plus, plusieurs méthodes d'opérateur de requête standard fonctionnent sur des types autres que ceux basés sur IEnumerable<T> ou IQueryable<T>. Le type Enumerable définit deux de ces méthodes qui fonctionnent toutes deux sur les objets de type IEnumerable. Ces méthodes, Cast<TResult>(IEnumerable) et OfType<TResult>(IEnumerable), vous permettent d'autoriser l'interrogation d'une collection non paramétrée, ou non générique, dans le modèle LINQ. Pour cela, elles créent une collection fortement typée d'objets. La classe Queryable définit deux méthodes similaires, Cast<TResult>(IQueryable) et OfType<TResult>(IQueryable), qui opèrent sur les objets de type IQueryable.

Les opérateurs de requête standard diffèrent dans le déroulement de leur exécution, selon qu'ils retournent une valeur singleton ou une séquence de valeurs. Les méthodes qui retournent une valeur de singleton (par exemple, Average et Sum) s'exécutent immédiatement. Les méthodes qui retournent une séquence diffèrent l'exécution de la requête et retournent un objet énumérable.

Pour les méthodes qui opèrent sur des collections en mémoire, c'est-à-dire des méthodes qui étendent IEnumerable<T>, l'objet énumérable retourné capture les arguments passés à la méthode. Lorsque cet objet est énuméré, la logique de l'opérateur de requête est employée et les résultats de requête sont retournés.

En revanche, les méthodes qui étendent IQueryable<T> n'implémentent aucun comportement d'interrogation. Elles créent une arborescence d'expressions

représentant la requête à effectuer. Le traitement des requêtes est géré par l'objet IQueryable<T> source.

Les appels aux méthodes de requête peuvent être chaînés dans une requête unique, ce qui permet de rendre les requêtes arbitrairement complexes.

L'exemple de code suivant illustre l'utilisation des opérateurs de requête standard pour obtenir des informations sur une séquence.

C#

```
string sentence = "the quick brown fox jumps over the lazy dog";
// Split the string into individual words to create a collection.
string[] words = sentence.Split(' ');

// Using query expression syntax.
var query = from word in words
            group word.ToUpper() by word.Length into gr
            orderby gr.Key
            select new { Length = gr.Key, Words = gr };

// Using method-based query syntax.
var query2 = words.
    GroupBy(w => w.Length, w => w.ToUpper()).
    Select(g => new { Length = g.Key, Words = g }).
    OrderBy(o => o.Length);

foreach (var obj in query)
{
    Console.WriteLine("Words of length {0}:", obj.Length);
    foreach (string word in obj.Words)
        Console.WriteLine(word);
}

// This code example produces the following output:
//
// Words of length 3:
// THE
// FOX
// THE
// DOG
// Words of length 4:
// OVER
// LAZY
// Words of length 5:
// QUICK
// BROWN
// JUMPS
```

Syntaxe d'expression de requête

Certains des opérateurs de requête standard les plus courants ont une syntaxe de mots clés dédiée des langages C# et Visual Basic, qui permet de les appeler dans le cadre d'une *expression* de *requête*. Pour plus d'informations sur les opérateurs de requête standard qui ont des mots clés dédiés et sur leurs syntaxes correspondantes, consultez Syntaxe des expressions de requête pour les opérateurs de requête standard (C#).

Extension des opérateurs de requête standard

Vous pouvez augmenter l'ensemble d'opérateurs de requête standard en créant des méthodes spécifiques au domaine appropriées pour votre domaine ou technologie cible. Vous pouvez également remplacer les opérateurs de requête standard par vos propres implémentations qui fournissent des services supplémentaires, tels que l'évaluation à distance, la traduction des requêtes et l'optimisation. Pour obtenir un exemple, consultez AsEnumerable.

Obtenir une source de données

Dans une requête LINQ, la première étape consiste à spécifier la source de données. Dans C#, comme dans la plupart des langages de programmation, une variable doit être déclarée avant de pouvoir être utilisée. Dans une requête LINQ, la clause `from` apparaît en premier pour introduire la source de données (`customers`) et la *variable de portée* (`cust`).

C#

```
//queryAllCustomers is an IEnumerable<Customer>
var queryAllCustomers = from cust in customers
                        select cust;
```

La variable de portée est similaire à la variable d'itération dans une boucle `foreach`, à la différence qu'aucune itération réelle ne se produit dans une expression de requête. Quand la requête est exécutée, la variable de portée sert de référence à chaque élément consécutif dans `customers`. Comme le compilateur déduit le type de `cust`, vous n'avez pas à le spécifier explicitement. Des variables de portée supplémentaires peuvent être introduites par une clause `let`. Pour plus d'informations, consultez let, clause.

> ⓘ **Notes**
>
> Pour les sources de données non génériques telles que **ArrayList**, la variable de portée doit être explicitement typée. Pour plus d'informations, consultez **Guide**

pratique pour interroger un ArrayList avec LINQ (C#) et from, clause.

Filtrage

L'opération de requête la plus courante est probablement l'application d'un filtre sous forme d'expression booléenne. Du fait du filtre, la requête retourne uniquement les éléments pour lesquels l'expression a la valeur true. Le résultat est produit à l'aide de la clause `where`. En effet, le filtre spécifie les éléments à exclure de la séquence source. Dans l'exemple suivant, seuls les clients (`customers`) qui ont une adresse à Londres sont retournés.

C#

```
var queryLondonCustomers = from cust in customers
                           where cust.City == "London"
                           select cust;
```

Vous pouvez utiliser les opérateurs logiques C# `AND` et `OR` habituels pour appliquer le nombre d'expressions de filtre nécessaire dans la clause `where`. Par exemple, pour retourner uniquement les clients situés à « Londres » ET (`AND`) dont le nom est « Devon », vous devez écrire le code suivant :

C#

```
where cust.City == "London" && cust.Name == "Devon"
```

Pour retourner les clients situés à Londres ou à Paris, vous devez écrire le code suivant :

C#

```
where cust.City == "London" || cust.City == "Paris"
```

Pour plus d'informations, consultez where, clause.

Classement

Il est souvent utile de trier les données retournées. La clause `orderby` permet de trier les éléments de la séquence retournée en fonction du comparateur par défaut pour le type qui est trié. Par exemple, la requête suivante peut être étendue pour trier les résultats

selon la propriété `Name`. Comme `Name` est une chaîne, le comparateur par défaut effectue un tri alphabétique de A à Z.

C#

```
var queryLondonCustomers3 =
    from cust in customers
    where cust.City == "London"
    orderby cust.Name ascending
    select cust;
```

Pour trier les résultats dans l'ordre inverse, de Z à A, utilisez la clause `orderby… descending`.

Pour plus d'informations, consultez orderby, clause.

Regroupement

La clause `group` vous permet de grouper vos résultats selon une clé que vous spécifiez. Par exemple, vous pouvez spécifier un regroupement des résultats par ville (`City`) pour que tous les clients de Londres ou Paris soient dans des groupes individuels. Dans ce cas, utilisez la clé `cust.City`.

C#

```
// queryCustomersByCity is an IEnumerable<IGrouping<string, Customer>>
  var queryCustomersByCity =
      from cust in customers
      group cust by cust.City;

  // customerGroup is an IGrouping<string, Customer>
  foreach (var customerGroup in queryCustomersByCity)
  {
      Console.WriteLine(customerGroup.Key);
      foreach (Customer customer in customerGroup)
      {
          Console.WriteLine("    {0}", customer.Name);
      }
  }
```

Quand vous terminez une requête avec une clause `group`, les résultats sont présentés sous la forme d'une liste de listes. Chaque élément de la liste est un objet qui a un membre `Key` et une liste d'éléments qui sont regroupés sous cette clé. Quand vous itérez une requête qui produit une séquence de groupes, vous devez utiliser une boucle

`foreach` imbriquée. La boucle externe itère chaque groupe et la boucle interne itère les membres de chaque groupe.

Si vous devez faire référence aux résultats d'une opération de regroupement, utilisez le mot clé `into` pour créer un identificateur susceptible d'être interrogé ultérieurement. La requête suivante retourne uniquement les groupes qui contiennent plus de deux clients :

C#

```
// custQuery is an IEnumerable<IGrouping<string, Customer>>
var custQuery =
    from cust in customers
    group cust by cust.City into custGroup
    where custGroup.Count() > 2
    orderby custGroup.Key
    select custGroup;
```

Pour plus d'informations, consultez group, clause.

Jonction

Les opérations de jointure créent des associations entre les séquences qui ne sont pas explicitement modélisées dans les sources de données. Par exemple, effectuez une jointure pour rechercher tous les clients et distributeurs qui se trouvent au même endroit. Dans LINQ, la clause `join` fonctionne toujours par rapport à des collections d'objets plutôt que directement par rapport à des tables de base de données.

C#

```
var innerJoinQuery =
    from cust in customers
    join dist in distributors on cust.City equals dist.City
    select new { CustomerName = cust.Name, DistributorName = dist.Name };
```

Dans LINQ, vous n'avez pas à utiliser `join` aussi souvent que vous le faites dans SQL, car les clés étrangères dans LINQ sont représentées dans le modèle objet comme des propriétés qui contiennent une collection d'éléments. Par exemple, un objet `Customer` contient une collection d'objets `Order`. Au lieu d'effectuer une jointure, accédez aux commandes en utilisant la notation par points :

C#

```
from order in Customer.Orders...
```

Pour plus d'informations, consultez join, clause.

Sélection (projections)

La clause `select` produit les résultats de la requête et spécifie la « forme » ou le type de chaque élément retourné. Par exemple, spécifiez si les résultats doivent contenir des objets `Customer` complets, un seul membre, un sous-ensemble de membres ou un type de résultat complètement différent basé sur un calcul ou une création d'objet. Quand la clause `select` produit autre chose qu'une copie de l'élément source, l'opération est appelée *projection*. L'utilisation de projections pour transformer des données est une fonctionnalité puissante des expressions de requête LINQ. Pour plus d'informations, consultez Transformations de données avec LINQ (C#) et select, clause.

Tableau de syntaxe des expressions de requête

Le tableau ci-dessous répertorie les opérateurs de requête standard qui comportent des clauses d'expression de requête équivalentes.

Méthode	**Syntaxe d'expression de requête C#**
Cast	Utilisez une variable de portée explicitement typée, par exemple : `from int i in numbers` (Pour plus d'informations, consultez from, clause.)

GroupBy	`group … by`
	
	-ou-
	
	`group … by … into …`
	
	(Pour plus d'informations, consultez group, clause.)

GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>,	`join … in … on … equals … into …`

Méthode	Syntaxe d'expression de requête C#
Func<TOuter,IEnumerable<TInner>, TResult>)	(Pour plus d'informations, consultez join, clause.)
Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>)	`join … in … on … equals …` (Pour plus d'informations, consultez join, clause.)
OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	`orderby` (Pour plus d'informations, consultez orderby, clause.)
OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	`orderby … descending` (Pour plus d'informations, consultez orderby, clause.)
Select	`select` (Pour plus d'informations, consultez select, clause.)
SelectMany	Plusieurs clauses `from`. (Pour plus d'informations, consultez from, clause.)
ThenBy<TSource,TKey>(IOrderedEnumerable<TSource>, Func<TSource,TKey>)	`orderby …, …` (Pour plus d'informations, consultez orderby, clause.)
ThenByDescending<TSource,TKey>(IOrderedEnumerable<TSource>, Func<TSource,TKey>)	`orderby …, … descending` (Pour plus d'informations, consultez orderby, clause.)
Where	`where`

Méthode	Syntaxe d'expression de requête C#
	(Pour plus d'informations, consultez where, clause.)

Voir aussi

- Enumerable
- Queryable
- Méthodes d'extension
- Mots clés de requête (LINQ)
- Types anonymes

Présentation des opérateurs de requête standard (C#)

Article • 20/07/2023

Les *opérateurs de requête standard* sont les méthodes qui forment le modèle de requête LINQ. La plupart de ces méthodes fonctionnent sur des séquences, où une séquence est un objet dont le type implémente l'interface IEnumerable<T> ou l'interface IQueryable<T>. Les opérateurs de requête standard fournissent des fonctionnalités de requête, dont notamment le filtrage, la projection, l'agrégation, le tri, etc.

Il existe deux ensembles d'opérateurs de requête standard LINQ : l'un opérant sur des objets de type IEnumerable<T> et l'autre sur des objets de type IQueryable<T>. Les méthodes qui composent chaque ensemble sont des membres statiques des classes Enumerable et Queryable, respectivement. Elles sont définies en tant que *méthodes d'extension* du type sur lequel elles opèrent. Les méthodes d'extension peuvent être appelées à l'aide de la syntaxe de méthode statique ou de la syntaxe de méthode d'instance.

De plus, plusieurs méthodes d'opérateur de requête standard fonctionnent sur des types autres que ceux basés sur IEnumerable<T> ou IQueryable<T>. Le type Enumerable définit deux de ces méthodes qui fonctionnent toutes deux sur les objets de type IEnumerable. Ces méthodes, Cast<TResult>(IEnumerable) et OfType<TResult>(IEnumerable), vous permettent d'autoriser l'interrogation d'une collection non paramétrée, ou non générique, dans le modèle LINQ. Pour cela, elles créent une collection fortement typée d'objets. La classe Queryable définit deux méthodes similaires, Cast<TResult>(IQueryable) et OfType<TResult>(IQueryable), qui opèrent sur les objets de type IQueryable.

Les opérateurs de requête standard diffèrent dans le déroulement de leur exécution, selon qu'ils retournent une valeur singleton ou une séquence de valeurs. Les méthodes qui retournent une valeur de singleton (par exemple, Average et Sum) s'exécutent immédiatement. Les méthodes qui retournent une séquence diffèrent l'exécution de la requête et retournent un objet énumérable.

Pour les méthodes qui opèrent sur des collections en mémoire, c'est-à-dire des méthodes qui étendent IEnumerable<T>, l'objet énumérable retourné capture les arguments passés à la méthode. Lorsque cet objet est énuméré, la logique de l'opérateur de requête est employée et les résultats de requête sont retournés.

En revanche, les méthodes qui étendent IQueryable<T> n'implémentent aucun comportement d'interrogation. Elles créent une arborescence d'expressions

représentant la requête à effectuer. Le traitement des requêtes est géré par l'objet IQueryable<T> source.

Les appels aux méthodes de requête peuvent être chaînés dans une requête unique, ce qui permet de rendre les requêtes arbitrairement complexes.

L'exemple de code suivant illustre l'utilisation des opérateurs de requête standard pour obtenir des informations sur une séquence.

C#

```
string sentence = "the quick brown fox jumps over the lazy dog";
// Split the string into individual words to create a collection.
string[] words = sentence.Split(' ');

// Using query expression syntax.
var query = from word in words
            group word.ToUpper() by word.Length into gr
            orderby gr.Key
            select new { Length = gr.Key, Words = gr };

// Using method-based query syntax.
var query2 = words.
    GroupBy(w => w.Length, w => w.ToUpper()).
    Select(g => new { Length = g.Key, Words = g }).
    OrderBy(o => o.Length);

foreach (var obj in query)
{
    Console.WriteLine("Words of length {0}:", obj.Length);
    foreach (string word in obj.Words)
        Console.WriteLine(word);
}

// This code example produces the following output:
//
// Words of length 3:
// THE
// FOX
// THE
// DOG
// Words of length 4:
// OVER
// LAZY
// Words of length 5:
// QUICK
// BROWN
// JUMPS
```

Syntaxe d'expression de requête

Certains des opérateurs de requête standard les plus courants ont une syntaxe de mots clés dédiée des langages C# et Visual Basic, qui permet de les appeler dans le cadre d'une *expression* de *requête*. Pour plus d'informations sur les opérateurs de requête standard qui ont des mots clés dédiés et sur leurs syntaxes correspondantes, consultez Syntaxe des expressions de requête pour les opérateurs de requête standard (C#).

Extension des opérateurs de requête standard

Vous pouvez augmenter l'ensemble d'opérateurs de requête standard en créant des méthodes spécifiques au domaine appropriées pour votre domaine ou technologie cible. Vous pouvez également remplacer les opérateurs de requête standard par vos propres implémentations qui fournissent des services supplémentaires, tels que l'évaluation à distance, la traduction des requêtes et l'optimisation. Pour obtenir un exemple, consultez AsEnumerable.

Obtenir une source de données

Dans une requête LINQ, la première étape consiste à spécifier la source de données. Dans C#, comme dans la plupart des langages de programmation, une variable doit être déclarée avant de pouvoir être utilisée. Dans une requête LINQ, la clause `from` apparaît en premier pour introduire la source de données (`customers`) et la *variable de portée* (`cust`).

C#

```
//queryAllCustomers is an IEnumerable<Customer>
var queryAllCustomers = from cust in customers
                        select cust;
```

La variable de portée est similaire à la variable d'itération dans une boucle `foreach`, à la différence qu'aucune itération réelle ne se produit dans une expression de requête. Quand la requête est exécutée, la variable de portée sert de référence à chaque élément consécutif dans `customers`. Comme le compilateur déduit le type de `cust`, vous n'avez pas à le spécifier explicitement. Des variables de portée supplémentaires peuvent être introduites par une clause `let`. Pour plus d'informations, consultez let, clause.

> ⓘ **Notes**
>
> Pour les sources de données non génériques telles que **ArrayList**, la variable de portée doit être explicitement typée. Pour plus d'informations, consultez **Guide**

pratique pour interroger un ArrayList avec LINQ (C#) et from, clause.

Filtrage

L'opération de requête la plus courante est probablement l'application d'un filtre sous forme d'expression booléenne. Du fait du filtre, la requête retourne uniquement les éléments pour lesquels l'expression a la valeur true. Le résultat est produit à l'aide de la clause `where`. En effet, le filtre spécifie les éléments à exclure de la séquence source. Dans l'exemple suivant, seuls les clients (`customers`) qui ont une adresse à Londres sont retournés.

C#

```
var queryLondonCustomers = from cust in customers
                           where cust.City == "London"
                           select cust;
```

Vous pouvez utiliser les opérateurs logiques C# `AND` et `OR` habituels pour appliquer le nombre d'expressions de filtre nécessaire dans la clause `where`. Par exemple, pour retourner uniquement les clients situés à « Londres » ET (`AND`) dont le nom est « Devon », vous devez écrire le code suivant :

C#

```
where cust.City == "London" && cust.Name == "Devon"
```

Pour retourner les clients situés à Londres ou à Paris, vous devez écrire le code suivant :

C#

```
where cust.City == "London" || cust.City == "Paris"
```

Pour plus d'informations, consultez where, clause.

Classement

Il est souvent utile de trier les données retournées. La clause `orderby` permet de trier les éléments de la séquence retournée en fonction du comparateur par défaut pour le type qui est trié. Par exemple, la requête suivante peut être étendue pour trier les résultats

selon la propriété `Name`. Comme `Name` est une chaîne, le comparateur par défaut effectue un tri alphabétique de A à Z.

C#

```
var queryLondonCustomers3 =
    from cust in customers
    where cust.City == "London"
    orderby cust.Name ascending
    select cust;
```

Pour trier les résultats dans l'ordre inverse, de Z à A, utilisez la clause `orderby… descending`.

Pour plus d'informations, consultez orderby, clause.

Regroupement

La clause `group` vous permet de grouper vos résultats selon une clé que vous spécifiez. Par exemple, vous pouvez spécifier un regroupement des résultats par ville (`City`) pour que tous les clients de Londres ou Paris soient dans des groupes individuels. Dans ce cas, utilisez la clé `cust.City`.

C#

```
// queryCustomersByCity is an IEnumerable<IGrouping<string, Customer>>
  var queryCustomersByCity =
      from cust in customers
      group cust by cust.City;

  // customerGroup is an IGrouping<string, Customer>
  foreach (var customerGroup in queryCustomersByCity)
  {
      Console.WriteLine(customerGroup.Key);
      foreach (Customer customer in customerGroup)
      {
          Console.WriteLine("    {0}", customer.Name);
      }
  }
```

Quand vous terminez une requête avec une clause `group`, les résultats sont présentés sous la forme d'une liste de listes. Chaque élément de la liste est un objet qui a un membre `Key` et une liste d'éléments qui sont regroupés sous cette clé. Quand vous itérez une requête qui produit une séquence de groupes, vous devez utiliser une boucle

`foreach` imbriquée. La boucle externe itère chaque groupe et la boucle interne itère les membres de chaque groupe.

Si vous devez faire référence aux résultats d'une opération de regroupement, utilisez le mot clé `into` pour créer un identificateur susceptible d'être interrogé ultérieurement. La requête suivante retourne uniquement les groupes qui contiennent plus de deux clients :

C#

```
// custQuery is an IEnumerable<IGrouping<string, Customer>>
var custQuery =
    from cust in customers
    group cust by cust.City into custGroup
    where custGroup.Count() > 2
    orderby custGroup.Key
    select custGroup;
```

Pour plus d'informations, consultez group, clause.

Jonction

Les opérations de jointure créent des associations entre les séquences qui ne sont pas explicitement modélisées dans les sources de données. Par exemple, effectuez une jointure pour rechercher tous les clients et distributeurs qui se trouvent au même endroit. Dans LINQ, la clause `join` fonctionne toujours par rapport à des collections d'objets plutôt que directement par rapport à des tables de base de données.

C#

```
var innerJoinQuery =
    from cust in customers
    join dist in distributors on cust.City equals dist.City
    select new { CustomerName = cust.Name, DistributorName = dist.Name };
```

Dans LINQ, vous n'avez pas à utiliser `join` aussi souvent que vous le faites dans SQL, car les clés étrangères dans LINQ sont représentées dans le modèle objet comme des propriétés qui contiennent une collection d'éléments. Par exemple, un objet `Customer` contient une collection d'objets `Order`. Au lieu d'effectuer une jointure, accédez aux commandes en utilisant la notation par points :

C#

```
from order in Customer.Orders...
```

Pour plus d'informations, consultez join, clause.

Sélection (projections)

La clause `select` produit les résultats de la requête et spécifie la « forme » ou le type de chaque élément retourné. Par exemple, spécifiez si les résultats doivent contenir des objets `Customer` complets, un seul membre, un sous-ensemble de membres ou un type de résultat complètement différent basé sur un calcul ou une création d'objet. Quand la clause `select` produit autre chose qu'une copie de l'élément source, l'opération est appelée *projection*. L'utilisation de projections pour transformer des données est une fonctionnalité puissante des expressions de requête LINQ. Pour plus d'informations, consultez Transformations de données avec LINQ (C#) et select, clause.

Tableau de syntaxe des expressions de requête

Le tableau ci-dessous répertorie les opérateurs de requête standard qui comportent des clauses d'expression de requête équivalentes.

Méthode	**Syntaxe d'expression de requête C#**
Cast	Utilisez une variable de portée explicitement typée, par exemple : `from int i in numbers` (Pour plus d'informations, consultez from, clause.)

GroupBy	`group … by`
	
	-ou-
	
	`group … by … into …`
	
	(Pour plus d'informations, consultez group, clause.)

GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>,	`join … in … on … equals … into …`

Méthode	Syntaxe d'expression de requête C#
Func<TOuter,IEnumerable<TInner>, TResult>)	(Pour plus d'informations, consultez join, clause.)
Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>)	`join … in … on … equals …` (Pour plus d'informations, consultez join, clause.)
OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	`orderby` (Pour plus d'informations, consultez orderby, clause.)
OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	`orderby … descending` (Pour plus d'informations, consultez orderby, clause.)
Select	`select` (Pour plus d'informations, consultez select, clause.)
SelectMany	Plusieurs clauses `from`. (Pour plus d'informations, consultez from, clause.)
ThenBy<TSource,TKey>(IOrderedEnumerable<TSource>, Func<TSource,TKey>)	`orderby …, …` (Pour plus d'informations, consultez orderby, clause.)
ThenByDescending<TSource,TKey>(IOrderedEnumerable<TSource>, Func<TSource,TKey>)	`orderby …, … descending` (Pour plus d'informations, consultez orderby, clause.)
Where	`where`

Méthode	Syntaxe d'expression de requête C#
	(Pour plus d'informations, consultez where, clause.)

Voir aussi

- Enumerable
- Queryable
- Méthodes d'extension
- Mots clés de requête (LINQ)
- Types anonymes

Interroger une collection d'objets

Article • 20/07/2023

« LINQ to Objects » fait référence à l'utilisation directe de requêtes LINQ avec n'importe quelle collection IEnumerable ou IEnumerable<T>, sans utiliser de fournisseur LINQ ou d'API intermédiaire comme LINQ to SQL ou LINQ to XML. Vous pouvez utiliser LINQ pour interroger des collections énumérables telles que List<T>, Array ou Dictionary<TKey,TValue>. La collection peut être définie par l'utilisateur ou retournée par une API .NET. Avec l'approche LINQ, vous écrivez du code déclaratif qui décrit ce que vous voulez récupérer.

De plus, les requêtes LINQ offrent trois avantages principaux par rapport aux boucles `foreach` classiques :

- Elles sont plus concises et lisibles, en particulier durant le filtrage de plusieurs conditions.
- Elles fournissent de puissantes fonctionnalités de filtrage, de classement et de regroupement avec un minimum de code d'application.
- Elles peuvent être portées vers d'autres sources de données avec peu ou pas de changements.

En général, plus l'opération que vous voulez effectuer sur les données est complexe, plus vous avez intérêt à utiliser LINQ à la place des techniques d'itération classiques.

Cet exemple montre comment effectuer une requête simple sur une liste d'objets `Student`. Chaque objet `Student` contient des informations de base sur l'étudiant et une liste qui représente les notes de l'étudiant lors de quatre examens.

> **Notes**
>
> De nombreux autres exemples de cette section utilisent la même `Student` classe et `students` la même collection.

C#

```
class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int ID { get; set; }
    public GradeLevel? Year { get; set; }
```

```
    public List<int> ExamScores { get; set; }

    public Student(string FirstName, string LastName, int ID, GradeLevel
Year, List<int> ExamScores)
    {
        this.FirstName = FirstName;
        this.LastName = LastName;
        this.ID = ID;
        this.Year = Year;
        this.ExamScores = ExamScores;
    }

    public Student(string FirstName, string LastName, int StudentID,
List<int>? ExamScores = null)
    {
        this.FirstName = FirstName;
        this.LastName = LastName;
        ID = StudentID;
        this.ExamScores = ExamScores ?? Enumerable.Empty<int>().ToList();
    }

    public static List<Student> students = new()
    {
        new(
            FirstName: "Terry", LastName: "Adams", ID: 120,
            Year: GradeLevel.SecondYear,
            ExamScores: new() { 99, 82, 81, 79 }
        ),
        new(
            "Fadi", "Fakhouri", 116,
            GradeLevel.ThirdYear,
            new() { 99, 86, 90, 94 }
        ),
        new(
            "Hanying", "Feng", 117,
            GradeLevel.FirstYear,
            new() { 93, 92, 80, 87 }
        ),
        new(
            "Cesar", "Garcia", 114,
            GradeLevel.FourthYear,
            new() { 97, 89, 85, 82 }
        ),
        new(
            "Debra", "Garcia", 115,
            GradeLevel.ThirdYear,
            new() { 35, 72, 91, 70 }
        ),
        new(
            "Hugo", "Garcia", 118,
            GradeLevel.SecondYear,
            new() { 92, 90, 83, 78 }
        ),
        new(
            "Sven", "Mortensen", 113,
```

```
            GradeLevel.FirstYear,
            new() { 88, 94, 65, 91 }
        ),
        new(
            "Claire", "O'Donnell", 112,
            GradeLevel.FourthYear,
            new() { 75, 84, 91, 39 }
        ),
        new(
            "Svetlana", "Omelchenko", 111,
            GradeLevel.SecondYear,
            new() { 97, 92, 81, 60 }
        ),
        new(
            "Lance", "Tucker", 119,
            GradeLevel.ThirdYear,
            new() { 68, 79, 88, 92 }
        ),
        new(
            "Michael", "Tucker", 122,
            GradeLevel.FirstYear,
            new() { 94, 92, 91, 91 }
        ),
        new(
            "Eugene", "Zabokritski", 121,
            GradeLevel.FourthYear,
            new() { 96, 85, 91, 60 }
        )
    };
}

enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};
```

Exemple

La requête suivante retourne les étudiants qui ont reçu une note supérieure ou égale à 90 lors de leur premier examen.

C#

```
void QueryHighScores(int exam, int score)
{
    var highScores =
        from student in students
        where student.ExamScores[exam] > score
```

```
        select new
        {
            Name = student.FirstName,
            Score = student.ExamScores[exam]
        };

    foreach (var item in highScores)
    {
        Console.WriteLine($"{item.Name,-15}{item.Score}");
    }
}

QueryHighScores(1, 90);
```

Cette requête est volontairement simple pour vous permettre de faire des essais. Par exemple, vous pouvez essayer plus de conditions dans la clause `where` ou utiliser une clause `orderby` pour trier les résultats.

Classification des opérateurs de requête standard en fonction de leur mode d'exécution

Les implémentations LINQ to Objects des méthodes d'opérateur de requête standard s'exécutent de deux manières principales : immédiate ou différée. En outre, les opérateurs de requête qui utilisent l'exécution différée peuvent être divisés en deux catégories : ceux prenant en charge la diffusion en continu et ceux ne la prenant pas en charge. Le fait de connaître le mode d'exécution des différents opérateurs de requête peut vous aider à comprendre les résultats obtenus à partir d'une requête donnée. Ceci est particulièrement vrai si la source de données change ou si vous générez une requête sur une autre requête. Cette rubrique classe les opérateurs de requête standard selon leur mode d'exécution.

Immédiat

L'exécution immédiate signifie que la source de données est lue et que l'opération est effectuée une seule fois. Tous les opérateurs de requête standard qui retournent un résultat scalaire s'exécutent immédiatement. Vous pouvez forcer l'exécution d'une requête immédiatement à l'aide des méthodes Enumerable.ToList ou Enumerable.ToArray. L'exécution immédiate permet la réutilisation des résultats de la requête, et non la déclaration de requête. Les résultats sont récupérés une fois, puis stockés pour une utilisation ultérieure.

Différé

L'exécution différée signifie que l'opération n'est pas effectuée au point où la requête est déclarée dans le code. L'opération est effectuée uniquement quand la variable de requête est énumérée, par exemple à l'aide d'une instruction `foreach`. Cela signifie que les résultats de l'exécution de la requête dépendent du contenu de la source de données au moment de l'exécution de la requête, et non au moment de sa définition. Si la variable de requête est énumérée plusieurs fois, les résultats peuvent différer chaque fois. Presque tous les opérateurs de requête standard dont le type de retour est IEnumerable<T> ou IOrderedEnumerable<TElement> s'exécutent de manière différée. L'exécution différée permet la réutilisation des requêtes, car la requête extrait les données mises à jour de la source de données chaque fois que les résultats de la requête sont itérés.

Les opérateurs de requête qui utilisent l'exécution différée peuvent également être classés comme prenant en charge la diffusion en continu ou non.

Diffusion en continu

Les opérateurs prenant en charge la diffusion en continu n'ont pas à lire toutes les données sources avant de générer des éléments. Au moment de l'exécution, un opérateur prenant en charge la diffusion en continu effectue son opération sur chaque élément source à mesure qu'il est lu et génère l'élément si nécessaire. Ce type d'opérateur continue à lire des éléments source jusqu'à ce qu'un élément de résultat puisse être produit. Cela signifie que plusieurs éléments source peuvent être lus pour produire un élément de résultat.

Requêtes qui ne sont pas des requêtes de streaming

Les opérateurs ne prenant pas en charge la diffusion en continu doivent lire toutes les données source avant de générer un élément de résultat. Les opérations telles que le tri ou le regroupement font partie de cette catégorie. Au moment de l'exécution, les opérateurs de requête ne prenant pas en charge la diffusion en continu lisent toutes les données source, les placent dans une structure de données, puis effectuent l'opération et génèrent les éléments de résultat.

Tableau de classification

Le tableau suivant classe chaque méthode d'opérateur de requête standard en fonction de son mode d'exécution.

> ⓘ **Notes**

Si un opérateur est présent dans deux colonnes, deux séquences d'entrée sont impliquées dans l'opération et chacune d'elles est évaluée différemment. Dans ces cas, la première séquence dans la liste de paramètres est toujours évaluée de façon différée, avec diffusion en continu.

Opérateur de requête standard	Type de retour	Exécution immédiate	Exécution différée avec diffusion en continu	Exécution différée sans diffusion en continu
Aggregate	`TSource`	X		
All	Boolean	X		
Any	Boolean	X		
AsEnumerable	IEnumerable<T>		X	
Average	Valeur numérique unique	X		
Cast	IEnumerable<T>		X	
Concat	IEnumerable<T>		X	
Contains	Boolean	X		
Count	Int32	X		
DefaultIfEmpty	IEnumerable<T>		X	
Distinct	IEnumerable<T>		X	
ElementAt	`TSource`	X		
ElementAtOrDefault	`TSource?`	X		
Empty	IEnumerable<T>	X		
Except	IEnumerable<T>		X	X
First	`TSource`	X		
FirstOrDefault	`TSource?`	X		
GroupBy	IEnumerable<T>			X
GroupJoin	IEnumerable<T>		X	X
Intersect	IEnumerable<T>		X	X

Opérateur de requête standard	Type de retour	Exécution immédiate	Exécution différée avec diffusion en continu	Exécution différée sans diffusion en continu
Join	IEnumerable<T>		X	X
Last	`TSource`	X		
LastOrDefault	`TSource?`	X		
LongCount	Int64	X		
Max	Valeur numérique unique, `TSource`, ou `TResult?`	X		
Min	Valeur numérique unique, `TSource`, ou `TResult?`	X		
OfType	IEnumerable<T>		X	
OrderBy	IOrderedEnumerable<TElement>			X
OrderByDescending	IOrderedEnumerable<TElement>			X
Range	IEnumerable<T>		X	
Repeat	IEnumerable<T>		X	
Reverse	IEnumerable<T>			X
Select	IEnumerable<T>		X	
SelectMany	IEnumerable<T>		X	
SequenceEqual	Boolean	X		
Single	`TSource`	X		
SingleOrDefault	`TSource?`	X		
Skip	IEnumerable<T>		X	
SkipWhile	IEnumerable<T>		X	
Sum	Valeur numérique unique	X		
Take	IEnumerable<T>		X	
TakeWhile	IEnumerable<T>		X	

Opérateur de requête standard	Type de retour	Exécution immédiate	Exécution différée avec diffusion en continu	Exécution différée sans diffusion en continu
ThenBy	IOrderedEnumerable<TElement>			X
ThenByDescending	IOrderedEnumerable<TElement>			X
ToArray	Tableau `TSource[]`	X		
ToDictionary	Dictionary<TKey,TValue>	X		
ToList	IList<T>	X		
ToLookup	ILookup<TKey,TElement>	X		
Union	IEnumerable<T>		X	
Where	IEnumerable<T>		X	

Voir aussi

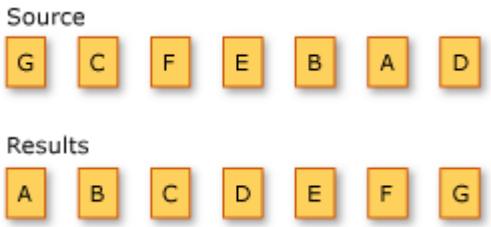
- LINQ (Language-Integrated Query)
- Interpolation de chaîne

Tri des données (C#)

Article • 07/04/2023

Une opération de tri ordonne les éléments d'une séquence en fonction d'un ou de plusieurs attributs. Le premier critère de tri effectue un tri principal sur les éléments. En spécifiant un deuxième critère de tri, vous pouvez trier les éléments dans chaque groupe de tri principal.

L'illustration suivante montre les résultats d'une opération de tri par ordre alphabétique sur une séquence de caractères :



Les méthodes d'opérateurs de requête standard qui trient les données sont répertoriées dans la section suivante.

Méthodes

Nom de la méthode	Description	Syntaxe d'expression de requête C#	Informations complémentaires
OrderBy	Trie les valeurs dans l'ordre croissant.	`orderby`	Enumerable.OrderBy Queryable.OrderBy
OrderByDescending	Trie les valeurs dans l'ordre décroissant.	`orderby … descending`	Enumerable.OrderByDescending Queryable.OrderByDescending
ThenBy	Effectue un tri secondaire dans l'ordre croissant.	`orderby …, …`	Enumerable.ThenBy Queryable.ThenBy
ThenByDescending	Effectue un tri secondaire dans l'ordre décroissant.	`orderby …, … descending`	Enumerable.ThenByDescending Queryable.ThenByDescending
Inverser	Inverse l'ordre des éléments dans une collection.	Non applicable.	Enumerable.Reverse Queryable.Reverse

Exemples de syntaxe d'expression de requête

Exemples de tri principal

Tri principal croissant

L'exemple suivant montre comment utiliser la clause `orderby` dans une requête LINQ pour trier les chaînes dans un tableau par longueur de chaîne, dans l'ordre croissant.

C#

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                            orderby word.Length
                            select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    the
    fox
    quick
    brown
    jumps
*/
```

Tri principal décroissant

L'exemple suivant montre comment utiliser la clause `orderby descending` dans une requête LINQ pour trier les chaînes selon leur première lettre, dans l'ordre décroissant.

C#

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                            orderby word.Substring(0, 1) descending
                            select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:
```

```
    the
    quick
    jumps
    fox
    brown
*/
```

Exemples de tri secondaire

Tri secondaire croissant

L'exemple suivant montre comment utiliser la clause `orderby` dans une requête LINQ pour effectuer un tri principal et un tri secondaire des chaînes dans un tableau. Les chaînes sont triées en premier lieu selon leur longueur, puis secondairement en fonction de la première lettre de la chaîne, chaque fois dans l'ordre croissant.

C#

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                            orderby word.Length, word.Substring(0, 1)
                            select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    fox
    the
    brown
    jumps
    quick
*/
```

Tri secondaire décroissant

L'exemple suivant montre comment utiliser la clause `orderby descending` dans une requête LINQ pour effectuer un tri principal dans l'ordre croissant et un tri secondaire dans l'ordre décroissant. Les chaînes sont triées en premier lieu selon leur longueur, puis secondairement en fonction de la première lettre de la chaîne.

C#

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                            orderby word.Length, word.Substring(0, 1)
descending
                            select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    the
    fox
    quick
    jumps
    brown
*/
```

Voir aussi

- System.Linq
- Présentation des opérateurs de requête standard (C#)
- orderby, clause
- Classer les résultats d'une clause join
- Guide pratique pour trier ou filtrer des données texte par mot ou par champ (LINQ) (C#)

Opérations ensemblistes (C#)

Article • 07/04/2023

Les opérations ensemblistes dans LINQ font référence à des opérations de requête qui génèrent un jeu de résultats basé sur la présence ou l'absence d'éléments équivalents dans la même collection (ou le même ensemble) ou dans une collection distincte (ou un ensemble distinct).

Les méthodes d'opérateur de requête standard qui effectuent des opérations ensemblistes sont répertoriées dans la section suivante.

Méthodes

Noms des méthodes	Description	Syntaxe d'expression de requête C#	Informations complémentaires
Distinct ou DistinctBy	Supprime les valeurs en double d'une collection.	Non applicable.	Enumerable.Distinct Enumerable.DistinctBy Queryable.Distinct Queryable.DistinctBy
Except ou ExceptBy	Retourne la différence ensembliste, à savoir les éléments d'une collection qui n'apparaissent pas dans une seconde collection.	Non applicable.	Enumerable.Except Enumerable.ExceptBy Queryable.Except Queryable.ExceptBy
Intersect ou IntersectBy	Retourne l'intersection ensembliste, à savoir les éléments qui apparaissent dans chacune des deux collections.	Non applicable.	Enumerable.Intersect Enumerable.IntersectBy Queryable.Intersect Queryable.IntersectBy
Union ou UnionBy	Retourne l'union ensembliste, à savoir les éléments uniques qui apparaissent dans l'une ou l'autre des deux collections.	Non applicable.	Enumerable.Union Enumerable.UnionBy Queryable.Union Queryable.UnionBy

Exemples

Certains des exemples suivants s'appuient sur un type `record` qui représente les planètes de notre système solaire.

```
C#
```

```
namespace SolarSystem;

record Planet(
    string Name,
    PlanetType Type,
    int OrderFromSun)
{
    public static readonly Planet Mercury =
        new(nameof(Mercury), PlanetType.Rock, 1);

    public static readonly Planet Venus =
        new(nameof(Venus), PlanetType.Rock, 2);

    public static readonly Planet Earth =
        new(nameof(Earth), PlanetType.Rock, 3);

    public static readonly Planet Mars =
        new(nameof(Mars), PlanetType.Rock, 4);

    public static readonly Planet Jupiter =
        new(nameof(Jupiter), PlanetType.Gas, 5);

    public static readonly Planet Saturn =
        new(nameof(Saturn), PlanetType.Gas, 6);

    public static readonly Planet Uranus =
        new(nameof(Uranus), PlanetType.Liquid, 7);

    public static readonly Planet Neptune =
        new(nameof(Neptune), PlanetType.Liquid, 8);

    // Yes, I know... not technically a planet anymore
    public static readonly Planet Pluto =
        new(nameof(Pluto), PlanetType.Ice, 9);
}
```

`record Planet` est un enregistrement positionnel qui a besoin des arguments `Name`, `Type` et `OrderFromSun` pour être instancié. Il existe plusieurs instances de planète `static readonly` sur le type `Planet`. Il s'agit de définitions pratiques pour les planètes connues. Le membre `Type` identifie le type de planète.

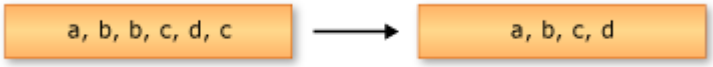
C#

```
namespace SolarSystem;

enum PlanetType
{
    Rock,
    Ice,
    Gas,
```

```
    Liquid
};
```

`Distinct` et `DistinctBy`

L'exemple suivant illustre le comportement de la méthode Enumerable.Distinct sur une séquence de chaînes. La séquence retournée contient les éléments uniques de la séquence d'entrée.



C#

```
string[] planets = { "Mercury", "Venus", "Venus", "Earth", "Mars", "Earth"
};

IEnumerable<string> query = from planet in planets.Distinct()
                            select planet;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Mercury
 * Venus
 * Earth
 * Mars
 */
```

DistinctBy est une approche alternative à `Distinct` qui prend un `keySelector`. `keySelector` est utilisé comme discriminateur comparatif du type source. Regardez le tableau de planètes suivant :

C#

```
Planet[] planets =
{
    Planet.Mercury,
    Planet.Venus,
    Planet.Earth,
    Planet.Mars,
    Planet.Jupiter,
    Planet.Saturn,
    Planet.Uranus,
```

```
    Planet.Neptune,
    Planet.Pluto
};
```

Dans le code suivant, les planètes sont discriminées en fonction de leur `PlanetType`, et la première planète de chaque type s'affiche :

C#

```
foreach (Planet planet in planets.DistinctBy(p => p.Type))
{
    Console.WriteLine(planet);
}

// This code produces the following output:
//     Planet { Name = Mercury, Type = Rock, OrderFromSun = 1 }
//     Planet { Name = Jupiter, Type = Gas, OrderFromSun = 5 }
//     Planet { Name = Uranus, Type = Liquid, OrderFromSun = 7 }
//     Planet { Name = Pluto, Type = Ice, OrderFromSun = 9 }
```

Dans le code C# précédent :

- Le tableau `Planet` est filtré distinctement sur la première occurrence de chaque type de planète unique.
- Les instances `planet` résultantes sont écrites dans la console.

`Except` et `ExceptBy`

L'exemple suivant illustre le comportement de Enumerable.Except. La séquence retournée contient uniquement les éléments de la première séquence d'entrée qui ne figurent pas dans la seconde séquence d'entrée.

C#

```
string[] planets1 = { "Mercury", "Venus", "Earth", "Jupiter" };
string[] planets2 = { "Mercury", "Earth", "Mars", "Jupiter" };

IEnumerable<string> query = from planet in planets1.Except(planets2)
                            select planet;

foreach (var str in query)
{
```

```
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Venus
 */
```

La méthode ExceptBy est une approche alternative à `Except` qui prend deux séquences de types potentiellement hétérogènes et un `keySelector`. `keySelector` est le même type que le type de la deuxième collection et il est utilisé comme discriminateur comparatif du type source. Regardez les tableaux de planètes suivants :

C#

```
Planet[] planets =
{
    Planet.Mercury,
    Planet.Venus,
    Planet.Earth,
    Planet.Jupiter
};

Planet[] morePlanets =
{
    Planet.Mercury,
    Planet.Earth,
    Planet.Mars,
    Planet.Jupiter
};
```

Pour rechercher les planètes de la première collection qui ne sont pas dans la deuxième collection, vous pouvez projeter les noms des planètes en tant que collection `second` et fournir le même `keySelector` :

C#

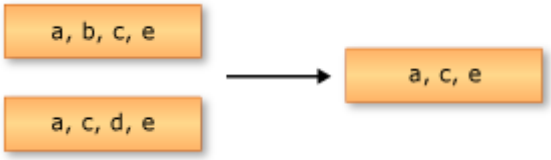
```
// A shared "keySelector"
static string PlanetNameSelector(Planet planet) => planet.Name;

foreach (Planet planet in
    planets.ExceptBy(
        morePlanets.Select(PlanetNameSelector), PlanetNameSelector))
{
    Console.WriteLine(planet);
}

// This code produces the following output:
//     Planet { Name = Venus, Type = Rock, OrderFromSun = 2 }
```

Dans le code C# précédent :

- `keySelector` est défini comme une fonction locale `static` qui discrimine sur un nom de planète.
- Le premier tableau de planètes est filtré sur les planètes qui ne se trouvent pas dans le deuxième tableau de planètes, en fonction de leur nom.
- L'instance `planet` résultante est écrite dans la console.

`Intersect` et `IntersectBy`

L'exemple suivant illustre le comportement de Enumerable.Intersect. La séquence retournée contient les éléments qui sont communs aux deux séquences d'entrée.



C#

```
string[] planets1 = { "Mercury", "Venus", "Earth", "Jupiter" };
string[] planets2 = { "Mercury", "Earth", "Mars", "Jupiter" };

IEnumerable<string> query = from planet in planets1.Intersect(planets2)
                            select planet;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Mercury
 * Earth
 * Jupiter
 */
```

La méthode IntersectBy est une approche alternative à `Intersect` qui prend deux séquences de types potentiellement hétérogènes et un `keySelector`. `keySelector` est utilisé comme discriminateur comparatif du type de la deuxième collection. Regardez les tableaux de planètes suivants :

C#

```
Planet[] firstFivePlanetsFromTheSun =
{
    Planet.Mercury,
    Planet.Venus,
    Planet.Earth,
    Planet.Mars,
    Planet.Jupiter
};

Planet[] lastFivePlanetsFromTheSun =
{
    Planet.Mars,
    Planet.Jupiter,
    Planet.Saturn,
    Planet.Uranus,
    Planet.Neptune
};
```

Il existe deux tableaux de planètes : le premier représente les cinq premières planètes les plus proches du soleil et le second représente les cinq dernières. Étant donné que le type `Planet` est un type `record` positionnel, vous pouvez utiliser sa sémantique de comparaison de valeurs sous la forme du `keySelector` :

C#

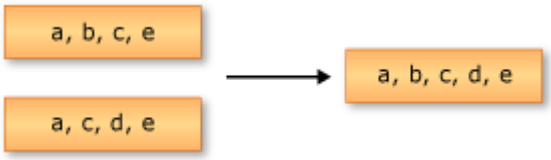
```
foreach (Planet planet in
    firstFivePlanetsFromTheSun.IntersectBy(
        lastFivePlanetsFromTheSun, planet => planet))
{
    Console.WriteLine(planet);
}

// This code produces the following output:
//     Planet { Name = Mars, Type = Rock, OrderFromSun = 4 }
//     Planet { Name = Jupiter, Type = Gas, OrderFromSun = 5 }
```

Dans le code C# précédent :

- Les deux tableaux `Planet` s'entrecroisent avec leur sémantique de comparaison de valeurs.
- Seules les planètes qui se trouvent dans les deux tableaux sont présentes dans la séquence résultante.
- Les instances `planet` résultantes sont écrites dans la console.

`Union` et `UnionBy`

L'exemple suivant illustre une opération d'union sur deux séquences de chaînes. La séquence retournée contient les éléments uniques des deux séquences d'entrée.



C#

```
string[] planets1 = { "Mercury", "Venus", "Earth", "Jupiter" };
string[] planets2 = { "Mercury", "Earth", "Mars", "Jupiter" };

IEnumerable<string> query = from planet in planets1.Union(planets2)
                            select planet;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Mercury
 * Venus
 * Earth
 * Jupiter
 * Mars
 */
```

La méthode UnionBy est une approche alternative à `Union` qui prend deux séquences du même type et un `keySelector`. `keySelector` est utilisé comme discriminateur comparatif du type source. Regardez les tableaux de planètes suivants :

C#

```
Planet[] firstFivePlanetsFromTheSun =
{
    Planet.Mercury,
    Planet.Venus,
    Planet.Earth,
    Planet.Mars,
    Planet.Jupiter
};

Planet[] lastFivePlanetsFromTheSun =
{
    Planet.Mars,
    Planet.Jupiter,
    Planet.Saturn,
```

```
    Planet.Uranus,
    Planet.Neptune
};
```

Pour unir ces deux collections dans une seule séquence, vous fournissez le `keySelector` :

C#

```
foreach (Planet planet in
    firstFivePlanetsFromTheSun.UnionBy(
        lastFivePlanetsFromTheSun, planet => planet))
{
    Console.WriteLine(planet);
}

// This code produces the following output:
//     Planet { Name = Mercury, Type = Rock, OrderFromSun = 1 }
//     Planet { Name = Venus, Type = Rock, OrderFromSun = 2 }
//     Planet { Name = Earth, Type = Rock, OrderFromSun = 3 }
//     Planet { Name = Mars, Type = Rock, OrderFromSun = 4 }
//     Planet { Name = Jupiter, Type = Gas, OrderFromSun = 5 }
//     Planet { Name = Saturn, Type = Gas, OrderFromSun = 6 }
//     Planet { Name = Uranus, Type = Liquid, OrderFromSun = 7 }
//     Planet { Name = Neptune, Type = Liquid, OrderFromSun = 8 }
```

Dans le code C# précédent :

- Les deux tableaux `Planet` sont assemblés en utilisant leur sémantique de comparaison de valeurs `record`.
- Les instances `planet` résultantes sont écrites dans la console.

Voir aussi

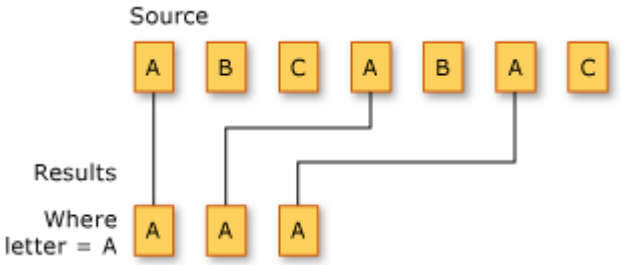
- System.Linq
- Présentation des opérateurs de requête standard (C#)
- Guide pratique pour combiner et comparer des collections de chaînes (LINQ) (C#)
- Guide pratique pour rechercher la différence ensembliste entre deux listes (LINQ) (C#)

Filtrage des données (C#)

Article • 16/03/2023

Le filtrage fait référence à l'opération de restriction du jeu de résultats pour que celui-ci contienne uniquement les éléments qui répondent à une condition spécifiée. Cette opération est également appelée « sélection ».

L'illustration suivante montre les résultats du filtrage d'une séquence de caractères. Le prédicat de l'opération de filtrage spécifie que le caractère doit être « A ».



Les méthodes d'opérateurs de requête standard qui effectuent des opérations de sélection sont répertoriées dans la section suivante.

Méthodes

Nom de la méthode	Description	Syntaxe d'expression de requête C#	Informations complémentaires
OfType	Sélectionne des valeurs, en fonction de leur capacité à être castées en un type spécifié.	Non applicable.	Enumerable.OfType Queryable.OfType
Where	Sélectionne les valeurs qui sont basées sur une fonction de prédicat.	`where`	Enumerable.Where Queryable.Where

Exemple de syntaxe d'expression de requête

L'exemple suivant utilise la clause `where` pour filtrer les chaînes d'un tableau qui ont une longueur spécifique.

C#

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };
```

```
IEnumerable<string> query = from word in words
                            where word.Length == 3
                            select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    the
    fox
*/
```

Voir aussi

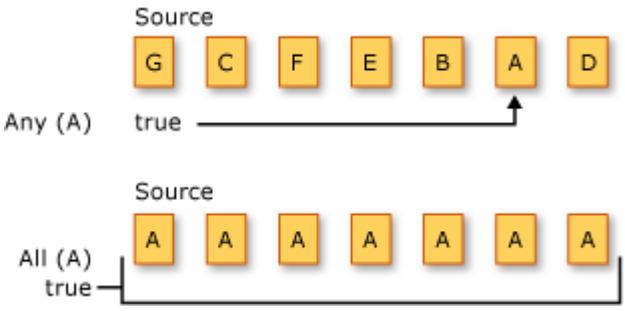
- System.Linq
- Présentation des opérateurs de requête standard (C#)
- where, clause
- Spécifier dynamiquement des filtres de prédicat au moment de l'exécution
- Guide pratique pour interroger les métadonnées d'un assembly avec la réflexion (LINQ) (C#)
- Guide pratique pour interroger des fichiers ayant un attribut ou un nom donné (C#)
- Guide pratique pour trier ou filtrer des données texte par mot ou par champ (LINQ) (C#)

Opérations de quantificateur dans LINQ (C#)

Article • 07/04/2023

Les opérations de quantificateur retournent une valeur Boolean qui indique si certains ou tous les éléments d'une séquence remplissent une condition.

L'illustration suivante représente deux opérations de quantificateur différentes sur deux séquences sources différentes. La première opération demande s'il y a des éléments qui sont le caractère « A ». La seconde opération demande si tous les éléments sont le caractère « A ». Les deux méthodes retournent `true` dans cet exemple.



Les méthodes d'opérateur de requête standard qui effectuent des opérations de quantificateur sont répertoriées dans la section suivante.

Méthodes

Nom de la méthode	Description	Syntaxe d'expression de requête C#	Informations complémentaires
Tous	Détermine si tous les éléments d'une séquence remplissent une condition.	Non applicable.	Enumerable.All Queryable.All
Quelconque	Détermine si certains éléments d'une séquence remplissent une condition.	Non applicable.	Enumerable.Any Queryable.Any
Contient	Détermine si une séquence contient un élément spécifié.	Non applicable.	Enumerable.Contains Queryable.Contains

Exemples de syntaxe d'expression de requête

Tous

L'exemple suivant utilise `All` pour vérifier que toutes les chaînes sont d'une longueur spécifique.

C#

```
class Market
{
    public string Name { get; set; }
    public string[] Items { get; set; }
}

public static void Example()
{
    List<Market> markets = new List<Market>
    {
        new Market { Name = "Emily's", Items = new string[] { "kiwi",
"cheery", "banana" } },
        new Market { Name = "Kim's", Items = new string[] { "melon",
"mango", "olive" } },
        new Market { Name = "Adam's", Items = new string[] { "kiwi",
"apple", "orange" } },
    };

    // Determine which market have all fruit names length equal to 5
    IEnumerable<string> names = from market in markets
                                where market.Items.All(item => item.Length
== 5)
                                select market.Name;

    foreach (string name in names)
    {
        Console.WriteLine($"{name} market");
    }

    // This code produces the following output:
    //
    // Kim's market
}
```

Quelconque

L'exemple suivant utilise `Any` pour vérifier si des chaînes commencent par « o ».

C#

```
class Market
{
    public string Name { get; set; }
    public string[] Items { get; set; }
}
```

```
public static void Example()
{
    List<Market> markets = new List<Market>
    {
        new Market { Name = "Emily's", Items = new string[] { "kiwi",
"cheery", "banana" } },
        new Market { Name = "Kim's", Items = new string[] { "melon",
"mango", "olive" } },
        new Market { Name = "Adam's", Items = new string[] { "kiwi",
"apple", "orange" } },
    };

    // Determine which market have any fruit names start with 'o'
    IEnumerable<string> names = from market in markets
                                where market.Items.Any(item =>
item.StartsWith("o"))
                                select market.Name;

    foreach (string name in names)
    {
        Console.WriteLine($"{name} market");
    }

    // This code produces the following output:
    //
    // Kim's market
    // Adam's market
}
```

Contient

L'exemple suivant utilise `Contains` pour vérifier qu'un tableau a un élément spécifique.

C#

```
class Market
{
    public string Name { get; set; }
    public string[] Items { get; set; }
}

public static void Example()
{
    List<Market> markets = new List<Market>
    {
        new Market { Name = "Emily's", Items = new string[] { "kiwi",
"cheery", "banana" } },
        new Market { Name = "Kim's", Items = new string[] { "melon",
"mango", "olive" } },
        new Market { Name = "Adam's", Items = new string[] { "kiwi",
"apple", "orange" } },
    };
```

```
    // Determine which market contains fruit names equal 'kiwi'
    IEnumerable<string> names = from market in markets
                                where market.Items.Contains("kiwi")
                                select market.Name;

    foreach (string name in names)
    {
        Console.WriteLine($"{name} market");
    }

    // This code produces the following output:
    //
    // Emily's market
    // Adam's market
}
```

Voir aussi

- System.Linq
- Présentation des opérateurs de requête standard (C#)
- Spécifier dynamiquement des filtres de prédicat au moment de l'exécution
- Guide pratique pour rechercher des phrases qui contiennent un groupe de mots spécifié (LINQ) (C#)

Opérations de projection (C#)

Article • 07/04/2023

La projection désigne l'opération de transformation d'un objet en une nouvelle forme qui se compose souvent uniquement des propriétés à utiliser ensuite. À l'aide de la projection, vous pouvez créer un nouveau type qui est généré à partir de chaque objet. Vous pouvez projeter une propriété et effectuer une fonction mathématique sur celle-ci. Vous pouvez également projeter l'objet d'origine sans le modifier.

Les méthodes d'opérateurs de requête standard qui effectuent des opérations de projection sont répertoriées dans la section suivante.

Méthodes

Noms des méthodes	Description	Syntaxe d'expression de requête C#	Informations complémentaires
Sélectionnez	Projette les valeurs qui sont basées sur une fonction de transformation.	`select`	Enumerable.Select Queryable.Select
SelectMany	Projette les séquences de valeurs qui sont basées sur une fonction de transformation, puis les aplatit en une seule séquence.	Utilisation de plusieurs clauses `from`	Enumerable.SelectMany Queryable.SelectMany
Zip	Produit une séquence de tuples avec des éléments de 2 à 3 séquences spécifiées.	Non applicable.	Enumerable.Zip Queryable.Zip

`Select`

L'exemple suivant utilise la clause `select` pour projeter la première lettre de chaque chaîne dans une liste de chaînes.

C#

```
List<string> words = new() { "an", "apple", "a", "day" };

var query = from word in words
            select word.Substring(0, 1);

foreach (string s in query)
```

```
    Console.WriteLine(s);

/* This code produces the following output:

    a
    a
    a
    d
*/
```

SelectMany

L'exemple suivant utilise plusieurs clauses `from` pour projeter tous les mots de chaque chaîne dans une liste de chaînes.

C#

```
List<string> phrases = new() { "an apple a day", "the quick brown fox" };

var query = from phrase in phrases
            from word in phrase.Split(' ')
            select word;

foreach (string s in query)
    Console.WriteLine(s);

/* This code produces the following output:

    an
    apple
    a
    day
    the
    quick
    brown
    fox
*/
```

Zip

Il existe plusieurs surcharges pour l'opérateur de projection `Zip`. Toutes les méthodes `Zip` fonctionnent sur des séquences de deux types ou plus potentiellement hétérogènes. Les deux premières surcharges retournent des tuples, avec le type positionnel correspondant à partir des séquences données.

Tenez compte des collections suivantes :

C#

```
// An int array with 7 elements.
IEnumerable<int> numbers = new[]
{
    1, 2, 3, 4, 5, 6, 7
};
// A char array with 6 elements.
IEnumerable<char> letters = new[]
{
    'A', 'B', 'C', 'D', 'E', 'F'
};
```

Pour projeter ces séquences ensemble, utilisez l'opérateur Enumerable.Zip<TFirst,TSecond>(IEnumerable<TFirst>, IEnumerable<TSecond>) :

C#

```
foreach ((int number, char letter) in numbers.Zip(letters))
{
    Console.WriteLine($"Number: {number} zipped with letter: '{letter}'");
}
// This code produces the following output:
//     Number: 1 zipped with letter: 'A'
//     Number: 2 zipped with letter: 'B'
//     Number: 3 zipped with letter: 'C'
//     Number: 4 zipped with letter: 'D'
//     Number: 5 zipped with letter: 'E'
//     Number: 6 zipped with letter: 'F'
```

> ⓘ **Important**
>
> La séquence résultante d'une opération zip n'est jamais plus longue que la séquence la plus courte. Les collections `numbers` et `letters` diffèrent par leur longueur, et la séquence qui en résulte omet le dernier élément de la collection `numbers`, car il n'a rien à compresser.

La deuxième surcharge accepte une séquence `third`. Créons une autre collection, à savoir `emoji` :

C#

```
// A string array with 8 elements.
IEnumerable<string> emoji = new[]
{
    "😳", "🔥", "🎉", "👀", "⭐", "💜", "✓", "💯"
};
```

Pour projeter ces séquences ensemble, utilisez l'opérateur Enumerable.Zip<TFirst,TSecond,TThird>(IEnumerable<TFirst>, IEnumerable<TSecond>, IEnumerable<TThird>) :

C#

```
foreach ((int number, char letter, string em) in numbers.Zip(letters,
emoji))
{
    Console.WriteLine(
        $"Number: {number} is zipped with letter: '{letter}' and emoji:
{em}");
}
// This code produces the following output:
//     Number: 1 is zipped with letter: 'A' and emoji: 😳
//     Number: 2 is zipped with letter: 'B' and emoji: 🔥
//     Number: 3 is zipped with letter: 'C' and emoji: 🥕
//     Number: 4 is zipped with letter: 'D' and emoji: 👀
//     Number: 5 is zipped with letter: 'E' and emoji: ⭐
//     Number: 6 is zipped with letter: 'F' and emoji: 💜
```

Tout comme la surcharge précédente, la méthode `Zip` projette un tuple, mais cette fois avec trois éléments.

La troisième surcharge accepte un argument `Func<TFirst, TSecond, TResult>` qui agit comme sélecteur de résultats. Étant donné les deux types des séquences en cours de compression, vous pouvez projeter une nouvelle séquence résultante.

C#

```
foreach (string result in
    numbers.Zip(letters, (number, letter) => $"{number} = {letter}
({(int)letter})"))
{
    Console.WriteLine(result);
}
// This code produces the following output:
//     1 = A (65)
//     2 = B (66)
//     3 = C (67)
//     4 = D (68)
//     5 = E (69)
//     6 = F (70)
```

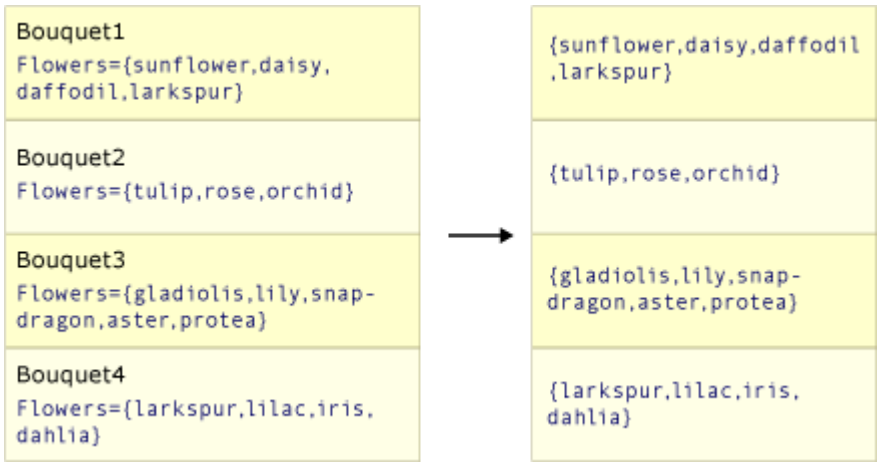
Avec la surcharge précédente `Zip`, la fonction spécifiée est appliquée aux éléments `numbers` et `letter` correspondants, ce qui produit une séquence des résultats `string`.

`Select` contre `SelectMany`

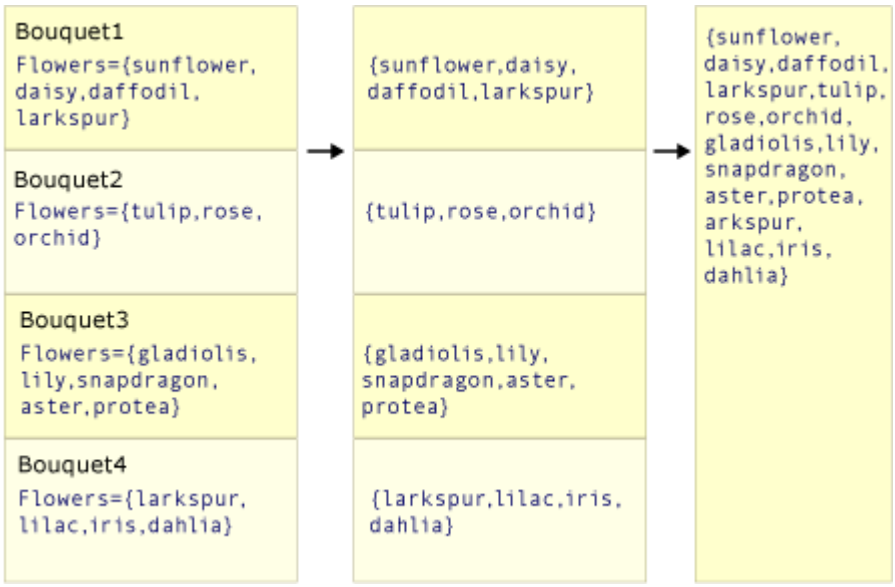
Les deux clauses `Select` et `SelectMany` retournent une valeur de résultat (ou des valeurs) à partir des valeurs sources. `Select` retourne une seule valeur de résultat pour chaque valeur source. Le résultat global est donc une collection qui a le même nombre d'éléments que la collection source. En revanche, `SelectMany` retourne un résultat global unique qui contient des sous-collections concaténées à partir de chaque valeur source. La fonction de transformation qui est passée comme argument à `SelectMany` doit retourner une séquence énumérable de valeurs pour chaque valeur source. Ces séquences énumérables sont ensuite concaténées par `SelectMany` pour créer une seule grande séquence.

Les deux illustrations suivantes montrent en quoi les actions de ces deux méthodes sont différentes d'un point de vue conceptuel. Dans chaque cas, supposons que la fonction (de transformation) du sélecteur sélectionne le tableau de fleurs (Flowers) à partir de chaque valeur source.

Cette illustration montre de quelle manière `Select` retourne une collection qui a le même nombre d'éléments que la collection source.



Cette illustration montre de quelle façon `SelectMany` concatène la séquence intermédiaire de tableaux en une seule valeur de résultat final qui contient chaque valeur de chaque tableau intermédiaire.



Exemple de code

L'exemple suivant compare le comportement de `Select` et de `SelectMany`. Le code crée un « bouquet » de fleurs en prenant les éléments de chaque liste de noms de fleurs de la collection source. Dans cet exemple, la « valeur unique » que la fonction de transformation Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>) utilise est elle-même une collection de valeurs. Cela nécessite d'utiliser la boucle `foreach` supplémentaire pour énumérer chaque chaîne dans chaque sous-séquence.

C#

```
class Bouquet
{
    public List<string> Flowers { get; set; }
}

static void SelectVsSelectMany()
{
    List<Bouquet> bouquets = new()
    {
        new Bouquet { Flowers = new List<string> { "sunflower", "daisy",
"daffodil", "larkspur" }},
        new Bouquet { Flowers = new List<string> { "tulip", "rose", "orchid"
}},
        new Bouquet { Flowers = new List<string> { "gladiolis", "lily",
"snapdragon", "aster", "protea" }},
        new Bouquet { Flowers = new List<string> { "larkspur", "lilac",
"iris", "dahlia" }}
    };

    IEnumerable<List<string>> query1 = bouquets.Select(bq => bq.Flowers);

    IEnumerable<string> query2 = bouquets.SelectMany(bq => bq.Flowers);
```

```
    Console.WriteLine("Results by using Select():");
    // Note the extra foreach loop here.
    foreach (IEnumerable<String> collection in query1)
        foreach (string item in collection)
            Console.WriteLine(item);

    Console.WriteLine("\nResults by using SelectMany():");
    foreach (string item in query2)
        Console.WriteLine(item);

    /* This code produces the following output:

       Results by using Select():
        sunflower
        daisy
        daffodil
        larkspur
        tulip
        rose
        orchid
        gladiolis
        lily
        snapdragon
        aster
        protea
        larkspur
        lilac
        iris
        dahlia

       Results by using SelectMany():
        sunflower
        daisy
        daffodil
        larkspur
        tulip
        rose
        orchid
        gladiolis
        lily
        snapdragon
        aster
        protea
        larkspur
        lilac
        iris
        dahlia
    */
}
```

Voir aussi

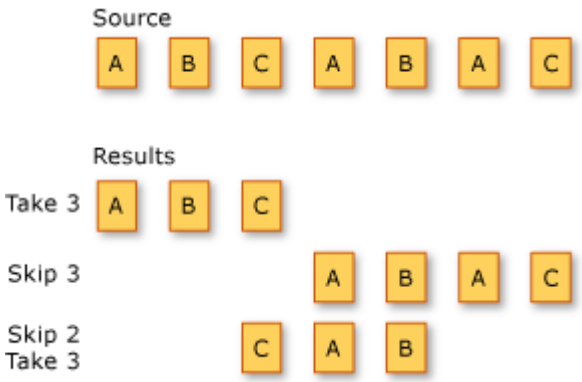
- System.Linq
- Présentation des opérateurs de requête standard (C#)
- select, clause
- Comment remplir des collections d’objets issues de plusieurs sources (LINQ) (C#)
- Comment fractionner un fichier en plusieurs fichiers à l’aide de groupes (LINQ) (C#)

Partitionnement des données (C#)

Article • 07/04/2023

Le partitionnement dans LINQ désigne l'opération consistant à diviser une séquence d'entrée en deux sections, sans réorganiser les éléments, puis à retourner l'une des sections.

L'illustration suivante montre les résultats de trois opérations de partitionnement différentes sur une séquence de caractères. La première opération retourne les trois premiers éléments de la séquence. La deuxième opération ignore les trois premiers éléments et retourne les éléments restants. La troisième opération ignore les deux premiers éléments de la séquence et retourne les trois éléments suivants.



Les méthodes d'opérateurs de requête standard qui partitionnent des séquences sont répertoriées dans la section suivante.

Opérateurs

Noms des méthodes	Description	Syntaxe d'expression de requête C#	Informations complémentaires
Ignorer	Ignore les éléments jusqu'à une position spécifiée dans une séquence.	Non applicable.	Enumerable.Skip Queryable.Skip
SkipWhile	Ignore les éléments basés sur une fonction de prédicat jusqu'à un élément qui ne remplit pas la condition.	Non applicable.	Enumerable.SkipWhile Queryable.SkipWhile
Take	Prend les éléments jusqu'à une position spécifiée dans une séquence.	Non applicable.	Enumerable.Take Queryable.Take
TakeWhile	Prend les éléments basés sur une fonction de prédicat jusqu'à un élément qui ne remplit pas la condition.	Non applicable.	Enumerable.TakeWhile Queryable.TakeWhile

Noms des méthodes	Description	Syntaxe d'expression de requête C#	Informations complémentaires
Bloc	Fractionne les éléments d'une séquence en blocs d'une taille maximale spécifiée.	Non applicable.	Enumerable.Chunk Queryable.Chunk

Exemple

L'opérateur `Chunk` est utilisé pour fractionner des éléments d'une séquence en fonction d'un donné `size`.

C#

```
int chunkNumber = 1;
foreach (int[] chunk in Enumerable.Range(0, 8).Chunk(3))
{
    Console.WriteLine($"Chunk {chunkNumber++}:");
    foreach (int item in chunk)
    {
        Console.WriteLine($"    {item}");
    }

    Console.WriteLine();
}
// This code produces the following output:
// Chunk 1:
//    0
//    1
//    2
//
//Chunk 2:
//    3
//    4
//    5
//
//Chunk 3:
//    6
//    7
```

Le code C# précédent :

- S'appuie sur Enumerable.Range(Int32, Int32) pour générer une séquence de nombres.
- Applique l'opérateur `Chunk`, fractionnant la séquence en blocs d'une taille maximale de trois.

Voir aussi

- System.Linq
- Présentation des opérateurs de requête standard (C#)

Join Opérations (C#)

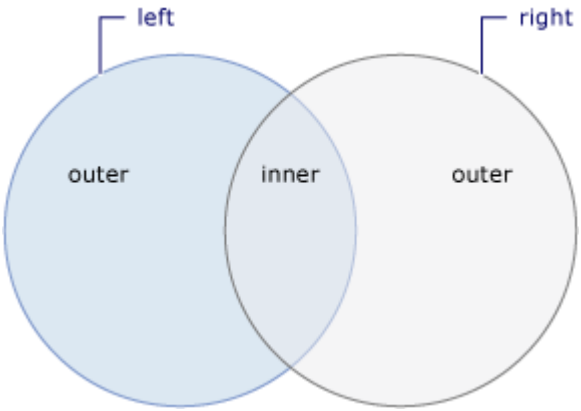
Article • 28/04/2023

Une *jointure* de deux sources de données est l'association des objets d'une source de données aux objets qui partagent un attribut commun dans une autre source de données.

Joining est une opération importante dans les requêtes qui ciblent des sources de données dont les relations entre elles ne peuvent pas être suivies directement. En programmation orientée objet, cela peut correspondre à une corrélation entre objets qui n'est pas modélisée, par exemple la direction vers l'arrière dans une relation à sens unique. Voici un exemple de relation à sens unique : une classe Customer a une propriété de type City, alors que la classe City n'a aucune propriété correspondant à une collection d'objets Customer. Si vous avez une liste d'objets City et si vous souhaitez rechercher tous les clients de chaque ville, vous pouvez recourir à une opération de jointure.

Les méthodes de jointure fournies dans le framework LINQ sont Join et GroupJoin. Ces méthodes effectuent des équijointures, qui sont des jointures associant deux sources de données en fonction de l'égalité de leurs clés. (À titre de comparaison, Transact-SQL prend en charge les opérateurs de jointure autres que « égaux », par exemple l'opérateur « inférieur à »). En termes de base de données relationnelle, Join implémente une jointure interne, un type de jointure dans lequel seuls les objets ayant une correspondance dans l'autre jeu de données sont retournés. La méthode GroupJoin n'a aucun équivalent direct dans le contexte des bases de données relationnelles, mais elle implémente un sur-ensemble de jointures internes et de jointures externes gauches. Une jointure externe gauche est une jointure qui retourne chaque élément de la source de données (gauche), même si elle n'a pas d'éléments corrélés dans l'autre source de données.

L'illustration suivante présente une vue conceptuelle de deux ensembles, ainsi que leurs éléments inclus dans une jointure interne ou une jointure externe gauche.



Méthodes

Nom de la méthode	Description	Syntaxe d'expression de requête C#	Informations complémentaires
Join	Joins deux séquences selon les fonctions de sélecteur de clés et extrait des paires de valeurs.	`join … in … on … equals …`	Enumerable.Join Queryable.Join
GroupJoin	Joins deux séquences selon les fonctions de sélecteur de clés et regroupe les correspondances obtenues pour chaque élément.	`join … in … on … equals … into …`	Enumerable.GroupJoin Queryable.GroupJoin

Exemples de syntaxe d'expression de requête

Join

L'exemple suivant utilise la clause `join … in … on … equals …` pour joindre deux séquences basées sur une valeur spécifique :

C#

```
class Product
{
    public string? Name { get; set; }
    public int CategoryId { get; set; }
}

class Category
{
    public int Id { get; set; }
    public string? CategoryName { get; set; }
}

public static void Example()
{
    List<Product> products = new List<Product>
    {
        new Product { Name = "Cola", CategoryId = 0 },
        new Product { Name = "Tea", CategoryId = 0 },
        new Product { Name = "Apple", CategoryId = 1 },
        new Product { Name = "Kiwi", CategoryId = 1 },
        new Product { Name = "Carrot", CategoryId = 2 },
    };
```

```
    List<Category> categories = new List<Category>
    {
        new Category { Id = 0, CategoryName = "Beverage" },
        new Category { Id = 1, CategoryName = "Fruit" },
        new Category { Id = 2, CategoryName = "Vegetable" }
    };

    // Join products and categories based on CategoryId
    var query = from product in products
                join category in categories on product.CategoryId equals
category.Id
                select new { product.Name, category.CategoryName };

    foreach (var item in query)
    {
        Console.WriteLine($"{item.Name} - {item.CategoryName}");
    }

    // This code produces the following output:
    //
    // Cola - Beverage
    // Tea - Beverage
    // Apple - Fruit
    // Kiwi - Fruit
    // Carrot - Vegetable
}
```

GroupJoin

L'exemple suivant utilise la clause `join … in … on … equals … into …` pour joindre deux séquences basées sur une valeur spécifique et regroupe les correspondances obtenues pour chaque élément :

C#

```
class Product
{
    public string? Name { get; set; }
    public int CategoryId { get; set; }
}

class Category
{
    public int Id { get; set; }
    public string? CategoryName { get; set; }
}

public static void Example()
{
    List<Product> products = new List<Product>
    {
```

```
        new Product { Name = "Cola", CategoryId = 0 },
        new Product { Name = "Tea", CategoryId = 0 },
        new Product { Name = "Apple", CategoryId = 1 },
        new Product { Name = "Kiwi", CategoryId = 1 },
        new Product { Name = "Carrot", CategoryId = 2 },
    };

    List<Category> categories = new List<Category>
    {
        new Category { Id = 0, CategoryName = "Beverage" },
        new Category { Id = 1, CategoryName = "Fruit" },
        new Category { Id = 2, CategoryName = "Vegetable" }
    };

    // Join categories and product based on CategoryId and grouping result
    var productGroups = from category in categories
                        join product in products on category.Id equals
product.CategoryId into productGroup
                        select productGroup;

    foreach (IEnumerable<Product> productGroup in productGroups)
    {
        Console.WriteLine("Group");
        foreach (Product product in productGroup)
        {
            Console.WriteLine($"{product.Name,8}");
        }
    }

    // This code produces the following output:
    //
    // Group
    //     Cola
    //      Tea
    // Group
    //    Apple
    //     Kiwi
    // Group
    //   Carrot
}
```

Voir aussi

- System.Linq
- Présentation des opérateurs de requête standard (C#)
- Types anonymes
- Formuler des Joins et des requêtes de produit croisé
- join, clause
- Join à l'aide de clés composites
- Guide pratique pour joindre du contenu issu de fichiers non similaires (LINQ) (C#)

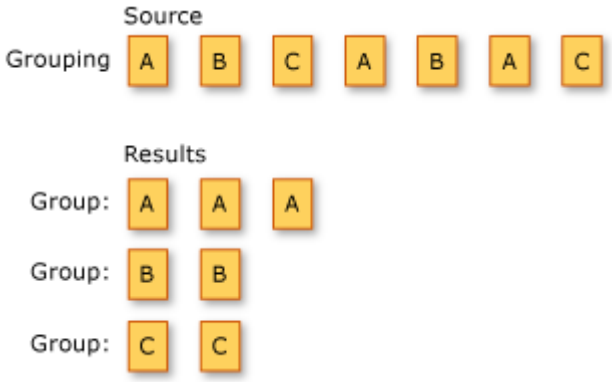
- Classer les résultats d’une clause join
- Effectuer des opérations de jointure personnalisées
- Effectuer des jointures groupées
- Effectuer des jointures internes
- Effectuer des jointures externes gauches
- Guide pratique pour remplir des collections d’objets issues de plusieurs sources (LINQ) (C#)

Regroupement des données (C#)

Article • 08/06/2023

Le regroupement consiste à placer des données dans des groupes afin que les éléments de chaque groupe partagent un attribut commun.

L'illustration suivante montre les résultats du regroupement d'une séquence de caractères. La clé de chaque groupe est le caractère.



Les méthodes d'opérateurs de requête standard qui regroupent les éléments de données sont répertoriées dans la section suivante.

Méthodes

Nom de la méthode	Description	Syntaxe d'expression de requête C#	Informations complémentaires

GroupBy	Regroupe les éléments qui partagent un attribut commun. Chaque groupe est représenté par un objet IGrouping<TKey,TElement>.	`group … by`
		
		-ou-
		
		`group … by … into …`
			Enumerable.GroupBy
			
			Queryable.GroupBy

ToLookup	Insère des éléments dans un Lookup<TKey,TElement> (un dictionnaire de type un-à-plusieurs) basé sur une fonction de sélecteur de clés.	Non applicable.	Enumerable.ToLookup

Exemple de syntaxe d'expression de requête

L'exemple de code suivant utilise la clause `group by` pour regrouper des entiers dans une liste selon qu'ils sont pairs ou impairs.

C#

```
List<int> numbers = new List<int>() { 35, 44, 200, 84, 3987, 4, 199, 329,
446, 208 };

IEnumerable<IGrouping<int, int>> query = from number in numbers
                                         group number by number % 2;

foreach (var group in query)
{
    Console.WriteLine(group.Key == 0 ? "\nEven numbers:" : "\nOdd
numbers:");
    foreach (int i in group)
        Console.WriteLine(i);
}

/* This code produces the following output:

    Odd numbers:
    35
    3987
    199
    329

    Even numbers:
    44
    200
    84
    4
    446
    208
*/
```

Voir aussi

- System.Linq
- Présentation des opérateurs de requête standard (C#)
- group, clause
- Créer un groupe imbriqué
- Guide pratique pour regrouper des fichiers par extension (LINQ) (C#)
- Regrouper les résultats d'une requête
- Effectuer une sous-requête sur une opération de regroupement
- Guide pratique pour diviser un fichier en plusieurs fichiers à l'aide de groupes (LINQ) (C#)

Opérations de génération (C#)

Article • 07/04/2023

La génération fait référence à la création d'une nouvelle séquence de valeurs.

Les méthodes d'opérateurs de requête standard qui effectuent des opérations de génération sont répertoriées dans la section suivante.

Méthodes

Nom de la méthode	Description	Syntaxe d'expression de requête C#	Informations complémentaires
DefaultIfEmpty	Remplace une collection vide par une collection de singletons par défaut.	Non applicable.	Enumerable.DefaultIfEmpty Queryable.DefaultIfEmpty
Vide	Retourne une collection vide.	Non applicable.	Enumerable.Empty
Plage	Génère une collection qui contient une séquence de nombres.	Non applicable.	Enumerable.Range
Répéter	Génère une collection qui contient une valeur répétée.	Non applicable.	Enumerable.Repeat

Voir aussi

- System.Linq
- Présentation des opérateurs de requête standard (C#)

Opérations d’égalité (C#)

Article • 07/04/2023

Deux séquences dont les éléments correspondants sont égaux et qui ont le même nombre d’éléments sont considérées comme égales.

Méthodes

Nom de la méthode	Description	Syntaxe d'expression de requête C#	Informations complémentaires
SequenceEqual	Détermine si deux séquences sont égales en comparant les éléments par paires.	Non applicable.	Enumerable.SequenceEqual Queryable.SequenceEqual

Voir aussi

- System.Linq
- Présentation des opérateurs de requête standard (C#)
- Guide pratique pour comparer le contenu de deux dossiers (LINQ) (C#)

Opérations d'éléments (C#)

Article • 07/04/2023

Les opérations d'éléments retournent un élément unique et spécifique à partir d'une séquence.

Les méthodes d'opérateurs de requête standard qui effectuent des opérations d'éléments sont répertoriées dans la section suivante.

Méthodes

Nom de la méthode	Description	Syntaxe d'expression de requête C#	Informations complémentaires
ElementAt	Retourne l'élément situé à un index spécifié dans une collection.	Non applicable.	Enumerable.ElementAt Queryable.ElementAt
ElementAtOrDefault	Retourne l'élément situé à un index spécifié dans une collection ou une valeur par défaut si l'index est hors limites.	Non applicable.	Enumerable.ElementAtOrDefault Queryable.ElementAtOrDefault
Premier	Retourne le premier élément d'une collection ou le premier élément qui remplit une condition.	Non applicable.	Enumerable.First Queryable.First
FirstOrDefault	Retourne le premier élément d'une collection ou le premier élément qui remplit une condition. Retourne une valeur par défaut s'il n'existe aucun élément répondant aux critères.	Non applicable.	Enumerable.FirstOrDefault Queryable.FirstOrDefault Queryable.FirstOrDefault<TSource> (IQueryable<TSource>)
Dernier	Retourne le dernier élément d'une collection ou le dernier élément qui remplit une condition.	Non applicable.	Enumerable.Last Queryable.Last

Nom de la méthode	Description	Syntaxe d'expression de requête C#	Informations complémentaires
LastOrDefault	Retourne le dernier élément d'une collection ou le dernier élément qui remplit une condition. Retourne une valeur par défaut s'il n'existe aucun élément répondant aux critères.	Non applicable.	Enumerable.LastOrDefault Queryable.LastOrDefault
Unique	Retourne le seul élément d'une collection ou le seul élément qui remplit une condition. Lève une InvalidOperationException s'il n'y a aucun élément ou s'il y a plusieurs éléments à retourner.	Non applicable.	Enumerable.Single Queryable.Single
SingleOrDefault	Retourne le seul élément d'une collection ou le seul élément qui remplit une condition. Retourne une valeur par défaut s'il n'y a aucun élément à retourner. Lève une InvalidOperationException s'il y a plusieurs éléments à retourner.	Non applicable.	Enumerable.SingleOrDefault Queryable.SingleOrDefault

Voir aussi

- System.Linq
- Présentation des opérateurs de requête standard (C#)
- Guide pratique pour rechercher le ou les fichiers les plus volumineux dans une arborescence de répertoires (LINQ) (C#)

Conversion des types de données (C#)

Article • 08/06/2023

Les méthodes de conversion changent le type d'objets en entrée.

Les opérations de conversion dans les requêtes LINQ sont utiles dans diverses applications. En voici quelques exemples :

- La méthode Enumerable.AsEnumerable peut être utilisée pour masquer l'implémentation personnalisée d'un type d'un opérateur de requête standard.
- La méthode Enumerable.OfType peut être utilisée pour activer des collections non paramétrables pour l'interrogation LINQ.
- Les méthodes Enumerable.ToArray, Enumerable.ToDictionary, Enumerable.ToList et Enumerable.ToLookup peuvent être utilisées pour forcer l'exécution immédiate de la requête au lieu de la différer jusqu'à ce que la requête soit énumérée.

Méthodes

Le tableau suivant répertorie les méthodes d'opérateur de requête standard qui effectuent des conversions de types de données.

Les méthodes de conversion de ce tableau dont les noms commencent par "As" modifient le type statique de la collection source mais ne l'énumèrent pas. Les méthodes dont les noms commencent par "To" énumèrent la collection source et placent les éléments dans le type de collection correspondant.

Nom de la méthode	Description	Syntaxe d'expression de requête C#	Informations complémentaires
AsEnumerable	Retourne l'entrée typée comme IEnumerable<T>.	Non applicable.	Enumerable.AsEnumerable
AsQueryable	Convertit un IEnumerable (générique) en IQueryable (générique).	Non applicable.	Queryable.AsQueryable

Nom de la méthode	Description	Syntaxe d'expression de requête C#	Informations complémentaires
Caster	Effectue un cast des éléments d'une collection en un type spécifié.	Utilisez une variable de portée explicitement typée. Par exemple : `from string str in words`	Enumerable.Cast Queryable.Cast
OfType	Filtre les valeurs en fonction de leur capacité à faire l'objet d'un cast en un type spécifié.	Non applicable.	Enumerable.OfType Queryable.OfType
ToArray	Convertit une collection en un tableau. Cette méthode force l'exécution de la requête.	Non applicable.	Enumerable.ToArray
ToDictionary	Place des éléments dans un Dictionary<TKey,TValue> basé sur une fonction de sélecteur de clés. Cette méthode force l'exécution de la requête.	Non applicable.	Enumerable.ToDictionary
ToList	Convertit une collection en List<T>. Cette méthode force l'exécution de la requête.	Non applicable.	Enumerable.ToList
ToLookup	Place des éléments dans un Lookup<TKey,TElement> (un dictionnaire de type un-à-plusieurs) basé sur une fonction de sélecteur de clés. Cette méthode force l'exécution de la requête.	Non applicable.	Enumerable.ToLookup

Exemple de syntaxe d'expression de requête

L'exemple de code suivant utilise une variable de portée explicitement typée pour effectuer un cast d'un type en un sous-type avant d'accéder à un membre qui est disponible uniquement sur le sous-type.

```
C#
```

```
class Plant
{
    public string Name { get; set; }
}

class CarnivorousPlant : Plant
{
    public string TrapType { get; set; }
}

static void Cast()
{
    Plant[] plants = new Plant[] {
        new CarnivorousPlant { Name = "Venus Fly Trap", TrapType = "Snap
Trap" },
        new CarnivorousPlant { Name = "Pitcher Plant", TrapType = "Pitfall
Trap" },
        new CarnivorousPlant { Name = "Sundew", TrapType = "Flypaper Trap"
},
        new CarnivorousPlant { Name = "Waterwheel Plant", TrapType = "Snap
Trap" }
    };

    var query = from CarnivorousPlant cPlant in plants
                where cPlant.TrapType == "Snap Trap"
                select cPlant;

    foreach (Plant plant in query)
        Console.WriteLine(plant.Name);

    /* This code produces the following output:

        Venus Fly Trap
        Waterwheel Plant
    */
}
```

Voir aussi

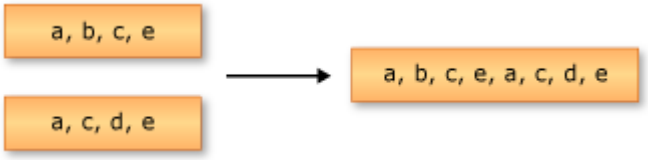
- System.Linq
- Présentation des opérateurs de requête standard (C#)
- from, clause
- Expressions de requête LINQ
- Guide pratique pour interroger un ArrayList avec LINQ (C#)

Opérations de concaténation (C#)

Article • 07/04/2023

La concaténation fait référence à l'opération d'ajout d'une séquence à une autre.

L'illustration suivante représente une opération de concaténation de deux séquences de caractères.



Les méthodes d'opérateurs de requête standard qui effectuent des opérations de concaténation sont répertoriées dans la section suivante.

Méthodes

Nom de la méthode	Description	Syntaxe d'expression de requête C#	Informations complémentaires
Concat	Concatène deux séquences pour en former une seule.	Non applicable.	Enumerable.Concat Queryable.Concat

Voir aussi

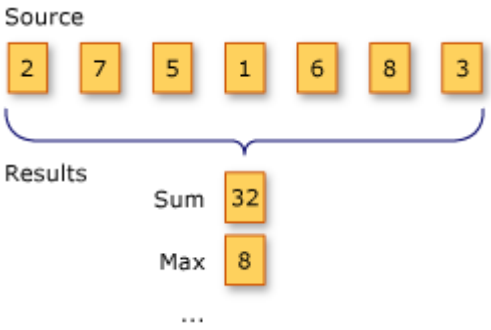
- System.Linq
- Présentation des opérateurs de requête standard (C#)
- Guide pratique pour combiner et comparer des collections de chaînes (LINQ) (C#)

Opérations d'agrégation (C#)

Article • 07/04/2023

Une opération d'agrégation calcule une valeur unique à partir d'une collection de valeurs. Par exemple, une opération d'agrégation peut être le calcul de la température quotidienne moyenne à partir des valeurs de température quotidiennes relevées sur un mois.

L'illustration suivante montre les résultats de deux opérations d'agrégation différentes sur une séquence de nombres. La première opération additionne les nombres. La deuxième opération retourne la valeur maximale dans la séquence.



Les méthodes d'opérateurs de requête standard qui effectuent des opérations d'agrégation sont répertoriées dans la section suivante.

Méthodes

Nom de la méthode	description	Syntaxe d'expression de requête C#	Informations complémentaires
Agrégat	Effectue une opération d'agrégation personnalisée sur les valeurs d'une collection.	Non applicable.	Enumerable.Aggregate Queryable.Aggregate
Moyenne	Calcule la valeur moyenne d'une collection de valeurs.	Non applicable.	Enumerable.Average Queryable.Average
Count	Compte tous les éléments d'une collection, ou uniquement les éléments basés sur une fonction de prédicat.	Non applicable.	Enumerable.Count Queryable.Count
LongCount	Compte tous les éléments d'une grande collection, ou uniquement les éléments basés sur une fonction de prédicat.	Non applicable.	Enumerable.LongCount Queryable.LongCount

Nom de la méthode	description	Syntaxe d'expression de requête C#	Informations complémentaires
Max ou MaxBy	Détermine la valeur maximale dans une collection.	Non applicable.	Enumerable.Max Enumerable.MaxBy Queryable.Max Queryable.MaxBy
Min ou MinBy	Détermine la valeur minimale dans une collection.	Non applicable.	Enumerable.Min Enumerable.MinBy Queryable.Min Queryable.MinBy
Sum	Calcule la somme des valeurs d'une collection.	Non applicable.	Enumerable.Sum Queryable.Sum

Voir aussi

- System.Linq
- Présentation des opérateurs de requête standard (C#)
- Guide pratique pour calculer des valeurs de colonnes dans un fichier texte CSV (LINQ) (C#)
- Guide pratique pour rechercher le ou les fichiers les plus volumineux dans une arborescence de répertoires (LINQ) (C#)
- Guide pratique pour rechercher le nombre total d'octets dans un ensemble de dossiers (LINQ) (C#)

Interroger une collection d'objets

Article • 20/07/2023

« LINQ to Objects » fait référence à l'utilisation directe de requêtes LINQ avec n'importe quelle collection IEnumerable ou IEnumerable<T>, sans utiliser de fournisseur LINQ ou d'API intermédiaire comme LINQ to SQL ou LINQ to XML. Vous pouvez utiliser LINQ pour interroger des collections énumérables telles que List<T>, Array ou Dictionary<TKey,TValue>. La collection peut être définie par l'utilisateur ou retournée par une API .NET. Avec l'approche LINQ, vous écrivez du code déclaratif qui décrit ce que vous voulez récupérer.

De plus, les requêtes LINQ offrent trois avantages principaux par rapport aux boucles `foreach` classiques :

- Elles sont plus concises et lisibles, en particulier durant le filtrage de plusieurs conditions.
- Elles fournissent de puissantes fonctionnalités de filtrage, de classement et de regroupement avec un minimum de code d'application.
- Elles peuvent être portées vers d'autres sources de données avec peu ou pas de changements.

En général, plus l'opération que vous voulez effectuer sur les données est complexe, plus vous avez intérêt à utiliser LINQ à la place des techniques d'itération classiques.

Cet exemple montre comment effectuer une requête simple sur une liste d'objets `Student`. Chaque objet `Student` contient des informations de base sur l'étudiant et une liste qui représente les notes de l'étudiant lors de quatre examens.

> **Notes**
>
> De nombreux autres exemples de cette section utilisent la même `Student` classe et `students` la même collection.

C#

```
class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int ID { get; set; }
    public GradeLevel? Year { get; set; }
```

```
    public List<int> ExamScores { get; set; }

    public Student(string FirstName, string LastName, int ID, GradeLevel
Year, List<int> ExamScores)
    {
        this.FirstName = FirstName;
        this.LastName = LastName;
        this.ID = ID;
        this.Year = Year;
        this.ExamScores = ExamScores;
    }

    public Student(string FirstName, string LastName, int StudentID,
List<int>? ExamScores = null)
    {
        this.FirstName = FirstName;
        this.LastName = LastName;
        ID = StudentID;
        this.ExamScores = ExamScores ?? Enumerable.Empty<int>().ToList();
    }

    public static List<Student> students = new()
    {
        new(
            FirstName: "Terry", LastName: "Adams", ID: 120,
            Year: GradeLevel.SecondYear,
            ExamScores: new() { 99, 82, 81, 79 }
        ),
        new(
            "Fadi", "Fakhouri", 116,
            GradeLevel.ThirdYear,
            new() { 99, 86, 90, 94 }
        ),
        new(
            "Hanying", "Feng", 117,
            GradeLevel.FirstYear,
            new() { 93, 92, 80, 87 }
        ),
        new(
            "Cesar", "Garcia", 114,
            GradeLevel.FourthYear,
            new() { 97, 89, 85, 82 }
        ),
        new(
            "Debra", "Garcia", 115,
            GradeLevel.ThirdYear,
            new() { 35, 72, 91, 70 }
        ),
        new(
            "Hugo", "Garcia", 118,
            GradeLevel.SecondYear,
            new() { 92, 90, 83, 78 }
        ),
        new(
            "Sven", "Mortensen", 113,
```

```
            GradeLevel.FirstYear,
            new() { 88, 94, 65, 91 }
        ),
        new(
            "Claire", "O'Donnell", 112,
            GradeLevel.FourthYear,
            new() { 75, 84, 91, 39 }
        ),
        new(
            "Svetlana", "Omelchenko", 111,
            GradeLevel.SecondYear,
            new() { 97, 92, 81, 60 }
        ),
        new(
            "Lance", "Tucker", 119,
            GradeLevel.ThirdYear,
            new() { 68, 79, 88, 92 }
        ),
        new(
            "Michael", "Tucker", 122,
            GradeLevel.FirstYear,
            new() { 94, 92, 91, 91 }
        ),
        new(
            "Eugene", "Zabokritski", 121,
            GradeLevel.FourthYear,
            new() { 96, 85, 91, 60 }
        )
    };
}

enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};
```

Exemple

La requête suivante retourne les étudiants qui ont reçu une note supérieure ou égale à 90 lors de leur premier examen.

C#

```
void QueryHighScores(int exam, int score)
{
    var highScores =
        from student in students
        where student.ExamScores[exam] > score
```

```
        select new
        {
            Name = student.FirstName,
            Score = student.ExamScores[exam]
        };

    foreach (var item in highScores)
    {
        Console.WriteLine($"{item.Name,-15}{item.Score}");
    }
}

QueryHighScores(1, 90);
```

Cette requête est volontairement simple pour vous permettre de faire des essais. Par exemple, vous pouvez essayer plus de conditions dans la clause `where` ou utiliser une clause `orderby` pour trier les résultats.

Classification des opérateurs de requête standard en fonction de leur mode d'exécution

Les implémentations LINQ to Objects des méthodes d'opérateur de requête standard s'exécutent de deux manières principales : immédiate ou différée. En outre, les opérateurs de requête qui utilisent l'exécution différée peuvent être divisés en deux catégories : ceux prenant en charge la diffusion en continu et ceux ne la prenant pas en charge. Le fait de connaître le mode d'exécution des différents opérateurs de requête peut vous aider à comprendre les résultats obtenus à partir d'une requête donnée. Ceci est particulièrement vrai si la source de données change ou si vous générez une requête sur une autre requête. Cette rubrique classe les opérateurs de requête standard selon leur mode d'exécution.

Immédiat

L'exécution immédiate signifie que la source de données est lue et que l'opération est effectuée une seule fois. Tous les opérateurs de requête standard qui retournent un résultat scalaire s'exécutent immédiatement. Vous pouvez forcer l'exécution d'une requête immédiatement à l'aide des méthodes Enumerable.ToList ou Enumerable.ToArray. L'exécution immédiate permet la réutilisation des résultats de la requête, et non la déclaration de requête. Les résultats sont récupérés une fois, puis stockés pour une utilisation ultérieure.

Différé

L'exécution différée signifie que l'opération n'est pas effectuée au point où la requête est déclarée dans le code. L'opération est effectuée uniquement quand la variable de requête est énumérée, par exemple à l'aide d'une instruction `foreach`. Cela signifie que les résultats de l'exécution de la requête dépendent du contenu de la source de données au moment de l'exécution de la requête, et non au moment de sa définition. Si la variable de requête est énumérée plusieurs fois, les résultats peuvent différer chaque fois. Presque tous les opérateurs de requête standard dont le type de retour est IEnumerable<T> ou IOrderedEnumerable<TElement> s'exécutent de manière différée. L'exécution différée permet la réutilisation des requêtes, car la requête extrait les données mises à jour de la source de données chaque fois que les résultats de la requête sont itérés.

Les opérateurs de requête qui utilisent l'exécution différée peuvent également être classés comme prenant en charge la diffusion en continu ou non.

Diffusion en continu

Les opérateurs prenant en charge la diffusion en continu n'ont pas à lire toutes les données sources avant de générer des éléments. Au moment de l'exécution, un opérateur prenant en charge la diffusion en continu effectue son opération sur chaque élément source à mesure qu'il est lu et génère l'élément si nécessaire. Ce type d'opérateur continue à lire des éléments source jusqu'à ce qu'un élément de résultat puisse être produit. Cela signifie que plusieurs éléments source peuvent être lus pour produire un élément de résultat.

Requêtes qui ne sont pas des requêtes de streaming

Les opérateurs ne prenant pas en charge la diffusion en continu doivent lire toutes les données source avant de générer un élément de résultat. Les opérations telles que le tri ou le regroupement font partie de cette catégorie. Au moment de l'exécution, les opérateurs de requête ne prenant pas en charge la diffusion en continu lisent toutes les données source, les placent dans une structure de données, puis effectuent l'opération et génèrent les éléments de résultat.

Tableau de classification

Le tableau suivant classe chaque méthode d'opérateur de requête standard en fonction de son mode d'exécution.

> ⓘ **Notes**

Si un opérateur est présent dans deux colonnes, deux séquences d'entrée sont impliquées dans l'opération et chacune d'elles est évaluée différemment. Dans ces cas, la première séquence dans la liste de paramètres est toujours évaluée de façon différée, avec diffusion en continu.

Opérateur de requête standard	Type de retour	Exécution immédiate	Exécution différée avec diffusion en continu	Exécution différée sans diffusion en continu
Aggregate	`TSource`	X		
All	Boolean	X		
Any	Boolean	X		
AsEnumerable	IEnumerable<T>		X	
Average	Valeur numérique unique	X		
Cast	IEnumerable<T>		X	
Concat	IEnumerable<T>		X	
Contains	Boolean	X		
Count	Int32	X		
DefaultIfEmpty	IEnumerable<T>		X	
Distinct	IEnumerable<T>		X	
ElementAt	`TSource`	X		
ElementAtOrDefault	`TSource?`	X		
Empty	IEnumerable<T>	X		
Except	IEnumerable<T>		X	X
First	`TSource`	X		
FirstOrDefault	`TSource?`	X		
GroupBy	IEnumerable<T>			X
GroupJoin	IEnumerable<T>		X	X
Intersect	IEnumerable<T>		X	X

Opérateur de requête standard	Type de retour	Exécution immédiate	Exécution différée avec diffusion en continu	Exécution différée sans diffusion en continu
Join	IEnumerable<T>		X	X
Last	`TSource`	X		
LastOrDefault	`TSource?`	X		
LongCount	Int64	X		
Max	Valeur numérique unique, `TSource`, ou `TResult?`	X		
Min	Valeur numérique unique, `TSource`, ou `TResult?`	X		
OfType	IEnumerable<T>		X	
OrderBy	IOrderedEnumerable<TElement>			X
OrderByDescending	IOrderedEnumerable<TElement>			X
Range	IEnumerable<T>		X	
Repeat	IEnumerable<T>		X	
Reverse	IEnumerable<T>			X
Select	IEnumerable<T>		X	
SelectMany	IEnumerable<T>		X	
SequenceEqual	Boolean	X		
Single	`TSource`	X		
SingleOrDefault	`TSource?`	X		
Skip	IEnumerable<T>		X	
SkipWhile	IEnumerable<T>		X	
Sum	Valeur numérique unique	X		
Take	IEnumerable<T>		X	
TakeWhile	IEnumerable<T>		X	

Opérateur de requête standard	Type de retour	Exécution immédiate	Exécution différée avec diffusion en continu	Exécution différée sans diffusion en continu
ThenBy	IOrderedEnumerable<TElement>			X
ThenByDescending	IOrderedEnumerable<TElement>			X
ToArray	Tableau `TSource[]`	X		
ToDictionary	Dictionary<TKey,TValue>	X		
ToList	IList<T>	X		
ToLookup	ILookup<TKey,TElement>	X		
Union	IEnumerable<T>		X	
Where	IEnumerable<T>		X	

Voir aussi

- LINQ (Language-Integrated Query)
- Interpolation de chaîne

LINQ (Language-Integrated Query)

Article • 24/02/2023

LINQ (Language-Integrated Query) est le nom d'un ensemble de technologies basé sur l'intégration de fonctions de requête directement dans le langage C#. En règle générale, les requêtes de données sont exprimées comme de simples chaînes, sans vérification de type au moment de l'exécution, ni prise en charge IntelliSense. En outre, vous devez apprendre un langage de requête différent pour chaque type de source de données : bases de données SQL, documents XML, services web, etc. Avec LINQ, une requête est une construction de langage de premier ordre, comme les classes, les méthodes et les événements.

Pour un développeur qui écrit des requêtes, la partie « intégrée au langage » la plus visible de LINQ est l'expression de requête. Les expressions de requête sont écrites selon une *syntaxe de requête* déclarative. En utilisant la syntaxe de requête, vous pouvez effectuer des opérations de filtrage, de classement et de regroupement sur des sources de données avec un minimum de code. Vous utilisez les mêmes modèles d'expression de requête de base pour interroger et transformer des données dans les bases de données SQL, les jeux de données ADO .NET, les flux et documents XML et les collections .NET.

L'exemple suivant montre l'opération de requête complète. L'opération complète comprend la création d'une source de données, la définition de l'expression de requête et l'exécution de la requête dans une instruction `foreach`.

C#

```
// Specify the data source.
int[] scores = { 97, 92, 81, 60 };

// Define the query expression.
IEnumerable<int> scoreQuery =
    from score in scores
    where score > 80
    select score;

// Execute the query.
foreach (int i in scoreQuery)
{
    Console.Write(i + " ");
}

// Output: 97 92 81
```

Vue d'ensemble des expressions de requête

- Les expressions de requête peuvent être utilisées pour interroger et transformer des données à partir de n'importe quelle source de données compatible LINQ. Par exemple, une même requête peut récupérer des données d'une base de données SQL et générer un flux XML en sortie.

- Les expressions de requête sont faciles à saisir car elles utilisent de nombreuses constructions familières du langage C#.

- Les variables d'une expression de requête sont toutes fortement typées, même si, dans de nombreux cas, il n'est pas nécessaire de fournir le type explicitement, car le compilateur peut le déduire. Pour plus d'informations, consultez la page Relations entre les types dans les opérations de requête LINQ.

- Une requête ne s'exécute pas tant que vous n'avez pas itéré la variable de requête, par exemple dans une instruction `foreach`. Pour plus d'informations, consultez Introduction aux requêtes LINQ.

- Lors de la compilation, les expressions de requête sont converties en appels de méthode d'opérateur de requête standard en fonction des règles définies dans la spécification du langage C#. Toute requête exprimable avec la syntaxe de requête peut également être exprimée avec la syntaxe de méthode. Toutefois, dans la plupart des cas, la syntaxe de requête est plus lisible et plus concise. Pour plus d'informations, consultez les pages Spécification du langage C# et Vue d'ensemble des opérateurs de requête standard.

- En règle générale, lorsque vous écrivez des requêtes LINQ, nous vous recommandons d'utiliser la syntaxe de requête dans la mesure du possible et la syntaxe de méthode si nécessaire. Il n'y a aucune différence de sémantique ou de performances entre les deux formats. Les expressions de requête sont souvent plus lisibles que les expressions équivalentes écrites avec la syntaxe de méthode.

- Certaines opérations de requête, comme Count ou Max, n'ont pas d'expression de requête équivalente et doivent par conséquent être exprimées sous la forme d'un appel de méthode. La syntaxe de méthode peut être combinée avec la syntaxe de requête de différentes manières. Pour plus d'informations, consultez la page Syntaxe de requête et syntaxe de méthode dans LINQ.

- Les expressions de requête peuvent être compilées dans des arborescences d'expressions ou des délégués, selon le type auquel la requête est appliquée. Les requêtes IEnumerable<T> sont compilées en délégués. Les requêtes IQueryable et

IQueryable<T> sont compilées en arborescences d'expression. Pour plus d'informations, consultez la page Arborescences d'expressions.

Étapes suivantes

Pour approfondir votre connaissance de LINQ, commencez par vous familiariser avec certains concepts de base expliqués sur la page Principes de base des expressions de requête, puis lisez la documentation relative à la technologie LINQ qui vous intéresse :

- Documents XML : LINQ to XML
- ADO.NET Entity Framework : LINQ to Entities
- Collections, fichiers, chaînes, etc. .NET : LINQ to Objects

Pour approfondir votre compréhension de LINQ en général, consultez la page LINQ en C#.

Pour commencer à travailler avec LINQ en C#, consultez le didacticiel Travailler avec LINQ.

Comment compter les occurrences d'un mot dans une chaîne (LINQ) (C#)

Article • 20/07/2023

Cet exemple montre comment utiliser une requête LINQ pour compter les occurrences d'un mot spécifié dans une chaîne. Notez que, pour effectuer le décompte, la méthode Split est d'abord appelée pour créer un tableau de mots. La méthode Split a un coût en matière de performances. Si la seule opération sur la chaîne consiste à compter les mots, il est préférable d'utiliser les méthodes Matches ou IndexOf. Toutefois, si les performances ne sont pas un facteur critique, ou si vous avez déjà fractionné la phrase pour effectuer d'autres types de requêtes sur elle, il est judicieux d'utiliser LINQ pour compter les mots ou les expressions.

Exemple

C#

```
class CountWords
{
    static void Main()
    {
        string text = @"Historically, the world of data and the world of
objects" +
          @" have not been well integrated. Programmers work in C# or Visual
Basic" +
          @" and also in SQL or XQuery. On the one side are concepts such as
classes," +
          @" objects, fields, inheritance, and .NET APIs. On the other side"
+
          @" are tables, columns, rows, nodes, and separate languages for
dealing with" +
          @" them. Data types often require translation between the two
worlds; there are" +
          @" different standard functions. Because the object world has no
notion of query, a" +
          @" query can only be represented as a string without compile-time
type checking or" +
          @" IntelliSense support in the IDE. Transferring data from SQL
tables or XML trees to" +
          @" objects in memory is often tedious and error-prone.";

        string searchTerm = "data";

        //Convert the string into an array of words
        string[] source = text.Split(new char[] { '.', '?', '!', ' ', ';',
':', ',' }, StringSplitOptions.RemoveEmptyEntries);
```

```
        // Create the query.  Use the InvariantCultureIgnoreCase comparision
to match "data" and "Data"
        var matchQuery = from word in source
                         where word.Equals(searchTerm,
StringComparison.InvariantCultureIgnoreCase)
                         select word;

        // Count the matches, which executes the query.
        int wordCount = matchQuery.Count();
        Console.WriteLine("{0} occurrences(s) of the search term \"{1}\"
were found.", wordCount, searchTerm);

        // Keep console window open in debug mode
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/* Output:
   3 occurrences(s) of the search term "data" were found.
*/
```

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Comment rechercher des phrases qui contiennent un ensemble de mots spécifié (LINQ) (C#)

Article • 20/07/2023

Cet exemple montre comment rechercher dans un fichier texte les phrases qui contiennent des correspondances pour chaque ensemble de mots spécifié. Même si le tableau des termes de recherche est codé en dur dans cet exemple, il pourrait aussi être rempli dynamiquement lors de l'exécution. Dans cet exemple, la requête retourne les phrases qui contiennent les mots « Historically », « data » et « integrated ».

Exemple

C#

```
class FindSentences
{
    static void Main()
    {
        string text = @"Historically, the world of data and the world of
objects " +
        @"have not been well integrated. Programmers work in C# or Visual
Basic " +
        @"and also in SQL or XQuery. On the one side are concepts such as
classes, " +
        @"objects, fields, inheritance, and .NET APIs. On the other side " +
        @"are tables, columns, rows, nodes, and separate languages for
dealing with " +
        @"them. Data types often require translation between the two worlds;
there are " +
        @"different standard functions. Because the object world has no
notion of query, a " +
        @"query can only be represented as a string without compile-time
type checking or " +
        @"IntelliSense support in the IDE. Transferring data from SQL tables
or XML trees to " +
        @"objects in memory is often tedious and error-prone.";

        // Split the text block into an array of sentences.
        string[] sentences = text.Split(new char[] { '.', '?', '!' });

        // Define the search terms. This list could also be dynamically
populated at run time.
        string[] wordsToMatch = { "Historically", "data", "integrated" };

        // Find sentences that contain all the terms in the wordsToMatch
```

```
array.
        // Note that the number of terms to match is not specified at
compile time.
        var sentenceQuery = from sentence in sentences
                            let w = sentence.Split(new char[] { '.', '?',
'!', ' ', ';', ':', ',' },

StringSplitOptions.RemoveEmptyEntries)
                            where
w.Distinct().Intersect(wordsToMatch).Count() == wordsToMatch.Count()
                            select sentence;

        // Execute the query. Note that you can explicitly type
        // the iteration variable here even though sentenceQuery
        // was implicitly typed.
        foreach (string str in sentenceQuery)
        {
            Console.WriteLine(str);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/* Output:
Historically, the world of data and the world of objects have not been well
integrated
*/
```

La requête fractionne d'abord le texte en phrases, puis fractionne les phrases en tableaux de chaînes contenant chacun des mots. Pour chacun de ces tableaux, la méthode Distinct supprime tous les mots en double, puis la requête effectue une opération Intersect sur le tableau de mots et le tableau `wordsToMatch`. Si le nombre de l'intersection est identique à celui du tableau `wordsToMatch`, cela signifie que tous les mots ont été trouvés et la phrase d'origine est donc retournée.

Dans l'appel à Split, les signes de ponctuation sont utilisés comme séparateurs pour être supprimés de la chaîne. Si vous ne l'avez pas fait, vous pourriez avoir, par exemple, la chaîne « Historically, », qui ne correspondrait pas au mot « Historically » du tableau `wordsToMatch`. Vous devrez peut-être utiliser des séparateurs supplémentaires, selon le type des signes de ponctuation qui se trouvent dans le texte source.

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Guide pratique pour interroger des caractères dans une chaîne (LINQ) (C#)

Article • 20/07/2023

La classe String implémente l'interface IEnumerable<T> générique. De ce fait, il est possible d'interroger n'importe quelle chaîne comme une séquence de caractères. Toutefois, ceci n'est pas une utilisation courante de LINQ. Pour les opérations de critères spéciaux complexes, utilisez la classe Regex.

Exemple

L'exemple suivant interroge une chaîne pour déterminer le nombre de chiffres qu'elle contient. Notez que la requête est « réutilisée » après sa première exécution. Ceci est possible car la requête proprement dite ne stocke pas de résultats réels.

C#

```
class QueryAString
{
    static void Main()
    {
        string aString = "ABCDE99F-J74-12-89A";

        // Select only those characters that are numbers
        IEnumerable<char> stringQuery =
          from ch in aString
          where Char.IsDigit(ch)
          select ch;

        // Execute the query
        foreach (char c in stringQuery)
            Console.Write(c + " ");

        // Call the Count method on the existing query.
        int count = stringQuery.Count();
        Console.WriteLine("Count = {0}", count);

        // Select all characters before the first '-'
        IEnumerable<char> stringQuery2 = aString.TakeWhile(c => c != '-');

        // Execute the second query
        foreach (char c in stringQuery2)
            Console.Write(c);

        Console.WriteLine(System.Environment.NewLine + "Press any key to
exit");
        Console.ReadKey();
```

```
    }
}
/* Output:
  Output: 9 9 7 4 1 2 8 9
  Count = 8
  ABCDE99F
*/
```

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Voir aussi

- Guide pratique pour combiner des requêtes LINQ avec des expressions régulières (C#)

Guide pratique pour combiner des requêtes LINQ avec des expressions régulières (C#)

Article • 20/07/2023

Cet exemple montre comment utiliser la classe Regex afin de créer une expression régulière pour les correspondances plus complexes des chaînes de texte. La requête LINQ facilite le filtrage des fichiers à parcourir à l'aide de l'expression régulière, et facilite également la personnalisation des résultats.

Exemple

C#

```
class QueryWithRegEx
{
    public static void Main()
    {
        // Modify this path as necessary so that it accesses your version of
Visual Studio.
        string startFolder = @"C:\Program Files (x86)\Microsoft Visual
Studio 14.0\";
        // One of the following paths may be more appropriate on your
computer.
        //string startFolder = @"C:\Program Files (x86)\Microsoft Visual
Studio\2017\";

        // Take a snapshot of the file system.
        IEnumerable<System.IO.FileInfo> fileList = GetFiles(startFolder);

        // Create the regular expression to find all things "Visual".
        System.Text.RegularExpressions.Regex searchTerm =
            new System.Text.RegularExpressions.Regex(@"Visual
(Basic|C#|C\+\+|Studio)");

        // Search the contents of each .htm file.
        // Remove the where clause to find even more matchedValues!
        // This query produces a list of files where a match
        // was found, and a list of the matchedValues in that file.
        // Note: Explicit typing of "Match" in select clause.
        // This is required because MatchCollection is not a
        // generic IEnumerable collection.
        var queryMatchingFiles =
            from file in fileList
            where file.Extension == ".htm"
            let fileText = System.IO.File.ReadAllText(file.FullName)
```

```
            let matches = searchTerm.Matches(fileText)
            where matches.Count > 0
            select new
            {
                name = file.FullName,
                matchedValues = from System.Text.RegularExpressions.Match
match in matches
                                select match.Value
            };

        // Execute the query.
        Console.WriteLine("The term \"{0}\" was found in:",
searchTerm.ToString());

        foreach (var v in queryMatchingFiles)
        {
            // Trim the path a bit, then write
            // the file name in which a match was found.
            string s = v.name.Substring(startFolder.Length - 1);
            Console.WriteLine(s);

            // For this file, write out all the matching strings
            foreach (var v2 in v.matchedValues)
            {
                Console.WriteLine("   " + v2);
            }
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // This method assumes that the application has discovery
    // permissions for all folders under the specified path.
    static IEnumerable<System.IO.FileInfo> GetFiles(string path)
    {
        if (!System.IO.Directory.Exists(path))
            throw new System.IO.DirectoryNotFoundException();

        string[] fileNames = null;
        List<System.IO.FileInfo> files = new List<System.IO.FileInfo>();

        fileNames = System.IO.Directory.GetFiles(path, "*.*",
System.IO.SearchOption.AllDirectories);
        foreach (string name in fileNames)
        {
            files.Add(new System.IO.FileInfo(name));
        }
        return files;
    }
}
```

Notez que vous pouvez également interroger l'objet MatchCollection qui est retourné par une recherche `RegEx`. Dans cet exemple, seule la valeur de chaque correspondance est générée dans les résultats. Toutefois, il est également possible d'utiliser LINQ pour effectuer toutes sortes de filtrage, de tri et de regroupement au sein de cette collection. Étant donné que MatchCollection est une collection IEnumerable non générique, vous devez déclarer explicitement le type de la variable de portée dans la requête.

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Guide pratique pour rechercher la différence ensembliste entre deux listes (LINQ) (C#)

Article • 20/07/2023

Cet exemple montre comment utiliser LINQ pour comparer deux listes de chaînes et sortir les lignes qui sont présentes dans names1.txt, mais pas dans names2.txt.

Pour créer le fichier de données

1. Copiez names1.txt et names2.txt dans votre dossier de solution comme indiqué dans Guide pratique pour combiner et comparer des collections de chaînes (LINQ) (C#).

Exemple

C#

```
class CompareLists
{
    static void Main()
    {
        // Create the IEnumerable data sources.
        string[] names1 =
System.IO.File.ReadAllLines(@"../../../names1.txt");
        string[] names2 =
System.IO.File.ReadAllLines(@"../../../names2.txt");

        // Create the query. Note that method syntax must be used here.
        IEnumerable<string> differenceQuery =
          names1.Except(names2);

        // Execute the query.
        Console.WriteLine("The following lines are in names1.txt but not
names2.txt");
        foreach (string s in differenceQuery)
            Console.WriteLine(s);

        // Keep the console window open until the user presses a key.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/* Output:
     The following lines are in names1.txt but not names2.txt
```

```
    Potra, Cristina
    Noriega, Fabricio
    Aw, Kam Foo
    Toyoshima, Tim
    Guy, Wey Yuan
    Garcia, Debra
     */
```

Certains types d'opérations de requête en C#, tels que Except, Distinct, Union et Concat, peuvent uniquement être exprimés dans une syntaxe basée sur une méthode.

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Guide pratique pour trier ou filtrer des données texte par mot ou par champ (LINQ) (C#)

Article • 20/07/2023

L'exemple suivant montre comment trier les lignes d'un texte structuré, telles que des valeurs séparées par des virgules, à l'aide d'un champ de la ligne. Le champ peut être spécifié de manière dynamique lors de l'exécution. Supposons que les champs du fichier scores.csv correspondent au numéro d'identification d'un étudiant, suivi d'une série de quatre notes d'examen.

Pour créer un fichier contenant des données

1. Copiez les données du fichier scores.csv disponible dans la rubrique Guide pratique pour joindre du contenu issu de fichiers non similaires (LINQ) (C#), puis enregistrez-les dans le dossier de votre solution.

Exemple

C#

```
public class SortLines
{
    static void Main()
    {
        // Create an IEnumerable data source
        string[] scores =
System.IO.File.ReadAllLines(@"../../../scores.csv");

        // Change this to any value from 0 to 4.
        int sortField = 1;

        Console.WriteLine("Sorted highest to lowest by field [{0}]:",
sortField);

        // Demonstrates how to return query from a method.
        // The query is executed here.
        foreach (string str in RunQuery(scores, sortField))
        {
            Console.WriteLine(str);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
```

```
        Console.ReadKey();
    }

    // Returns the query variable, not query results!
    static IEnumerable<string> RunQuery(IEnumerable<string> source, int num)
    {
        // Split the string and sort on field[num]
        var scoreQuery = from line in source
                         let fields = line.Split(',')
                         orderby fields[num] descending
                         select line;

        return scoreQuery;
    }
}
/* Output (if sortField == 1):
   Sorted highest to lowest by field [1]:
    116, 99, 86, 90, 94
    120, 99, 82, 81, 79
    111, 97, 92, 81, 60
    114, 97, 89, 85, 82
    121, 96, 85, 91, 60
    122, 94, 92, 91, 91
    117, 93, 92, 80, 87
    118, 92, 90, 83, 78
    113, 88, 94, 65, 91
    112, 75, 84, 91, 39
    119, 68, 79, 88, 92
    115, 35, 72, 91, 70
 */
```

Cet exemple montre également comment retourner une variable de requête à partir d'une méthode.

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Guide pratique pour réorganiser les champs d'un fichier délimité (LINQ) (C#)

Article • 20/07/2023

Un fichier de valeurs séparées par des virgules (CSV) est un fichier texte qui est souvent utilisé pour stocker des données de feuille de calcul ou autres données tabulaires qui sont représentées sous forme de lignes et de colonnes. En utilisant la méthode Split pour séparer les champs, il est très facile d'interroger et de manipuler des fichiers CSV à l'aide de LINQ. En fait, la même technique peut servir à réorganiser les sections de n'importe quelle ligne de texte structurée et ne se limite donc pas aux fichiers CSV.

Dans l'exemple suivant, supposons que les trois colonnes représentent le « nom », « prénom » et « ID » des étudiants. Les champs sont dans l'ordre alphabétique en fonction des noms des étudiants. La requête produit une séquence dans laquelle la colonne ID apparaît en premier, suivie d'une deuxième colonne qui comprend à la fois le prénom et le nom de l'étudiant. Les lignes sont réorganisées d'après le champ ID. Les résultats sont enregistrés dans un nouveau fichier et les données d'origine ne sont pas modifiées.

Pour créer le fichier de données

1. Copiez les lignes suivantes dans un fichier texte nommé spreadsheet1.csv. Enregistrez le fichier dans votre dossier de projet.

```csv
Adams,Terry,120
Fakhouri,Fadi,116
Feng,Hanying,117
Garcia,Cesar,114
Garcia,Debra,115
Garcia,Hugo,118
Mortensen,Sven,113
O'Donnell,Claire,112
Omelchenko,Svetlana,111
Tucker,Lance,119
Tucker,Michael,122
Zabokritski,Eugene,121
```

Exemple

C#

```
class CSVFiles
{
    static void Main(string[] args)
    {
        // Create the IEnumerable data source
        string[] lines =
System.IO.File.ReadAllLines(@"../../../spreadsheet1.csv");

        // Create the query. Put field 2 first, then
        // reverse and combine fields 0 and 1 from the old field
        IEnumerable<string> query =
            from line in lines
            let x = line.Split(',')
            orderby x[2]
            select x[2] + ", " + (x[1] + " " + x[0]);

        // Execute the query and write out the new file. Note that
WriteAllLines
        // takes a string[], so ToArray is called on the query.
        System.IO.File.WriteAllLines(@"../../../spreadsheet2.csv",
query.ToArray());

        Console.WriteLine("Spreadsheet2.csv written to disk. Press any key
to exit");
        Console.ReadKey();
    }
}
/* Output to spreadsheet2.csv:
111, Svetlana Omelchenko
112, Claire O'Donnell
113, Sven Mortensen
114, Cesar Garcia
115, Debra Garcia
116, Fadi Fakhouri
117, Hanying Feng
118, Hugo Garcia
119, Lance Tucker
120, Terry Adams
121, Eugene Zabokritski
122, Michael Tucker
 */
```

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Comment combiner et comparer des collections de chaînes (LINQ) (C#)

Article • 20/07/2023

Cet exemple montre comment fusionner des fichiers qui contiennent des lignes de texte, puis comment trier les résultats. Il montre plus précisément comment effectuer une concaténation simple, une union et une intersection avec les deux ensembles de lignes de texte.

Pour configurer le projet et les fichiers texte

1. Copiez ces noms dans un fichier texte nommé names1.txt, puis enregistrez-le dans votre dossier de projet :

   ```text
   Bankov, Peter
   Holm, Michael
   Garcia, Hugo
   Potra, Cristina
   Noriega, Fabricio
   Aw, Kam Foo
   Beebe, Ann
   Toyoshima, Tim
   Guy, Wey Yuan
   Garcia, Debra
   ```

2. Copiez ces noms dans un fichier texte nommé names2.txt, puis enregistrez-le dans votre dossier de projet : Notez que les deux fichiers ont des noms en commun.

   ```text
   Liu, Jinghao
   Bankov, Peter
   Holm, Michael
   Garcia, Hugo
   Beebe, Ann
   Gilchrist, Beth
   Myrcha, Jacek
   Giakoumakis, Leo
   McLin, Nkenge
   El Yassir, Mehdi
   ```

Exemple

C#

```
class MergeStrings
    {
        static void Main(string[] args)
        {
            //Put text files in your solution folder
            string[] fileA =
System.IO.File.ReadAllLines(@"../../../names1.txt");
            string[] fileB =
System.IO.File.ReadAllLines(@"../../../names2.txt");

            //Simple concatenation and sort. Duplicates are preserved.
            IEnumerable<string> concatQuery =
                fileA.Concat(fileB).OrderBy(s => s);

            // Pass the query variable to another function for execution.
            OutputQueryResults(concatQuery, "Simple concatenate and sort.
Duplicates are preserved:");

            // Concatenate and remove duplicate names based on
            // default string comparer.
            IEnumerable<string> uniqueNamesQuery =
                fileA.Union(fileB).OrderBy(s => s);
            OutputQueryResults(uniqueNamesQuery, "Union removes duplicate
names:");

            // Find the names that occur in both files (based on
            // default string comparer).
            IEnumerable<string> commonNamesQuery =
                fileA.Intersect(fileB);
            OutputQueryResults(commonNamesQuery, "Merge based on
intersect:");

            // Find the matching fields in each list. Merge the two
            // results by using Concat, and then
            // sort using the default string comparer.
            string nameMatch = "Garcia";

            IEnumerable<String> tempQuery1 =
                from name in fileA
                let n = name.Split(',')
                where n[0] == nameMatch
                select name;

            IEnumerable<string> tempQuery2 =
                from name2 in fileB
                let n2 = name2.Split(',')
                where n2[0] == nameMatch
                select name2;
```

```
            IEnumerable<string> nameMatchQuery =
                tempQuery1.Concat(tempQuery2).OrderBy(s => s);
            OutputQueryResults(nameMatchQuery, $"Concat based on partial
name match \"{nameMatch}\":");

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to exit");
            Console.ReadKey();
        }

        static void OutputQueryResults(IEnumerable<string> query, string
message)
        {
            Console.WriteLine(System.Environment.NewLine + message);
            foreach (string item in query)
            {
                Console.WriteLine(item);
            }
            Console.WriteLine("{0} total names in list", query.Count());
        }
    }
    /* Output:
        Simple concatenate and sort. Duplicates are preserved:
        Aw, Kam Foo
        Bankov, Peter
        Bankov, Peter
        Beebe, Ann
        Beebe, Ann
        El Yassir, Mehdi
        Garcia, Debra
        Garcia, Hugo
        Garcia, Hugo
        Giakoumakis, Leo
        Gilchrist, Beth
        Guy, Wey Yuan
        Holm, Michael
        Holm, Michael
        Liu, Jinghao
        McLin, Nkenge
        Myrcha, Jacek
        Noriega, Fabricio
        Potra, Cristina
        Toyoshima, Tim
        20 total names in list

        Union removes duplicate names:
        Aw, Kam Foo
        Bankov, Peter
        Beebe, Ann
        El Yassir, Mehdi
        Garcia, Debra
        Garcia, Hugo
        Giakoumakis, Leo
        Gilchrist, Beth
        Guy, Wey Yuan
```

```
        Holm, Michael
        Liu, Jinghao
        McLin, Nkenge
        Myrcha, Jacek
        Noriega, Fabricio
        Potra, Cristina
        Toyoshima, Tim
        16 total names in list

        Merge based on intersect:
        Bankov, Peter
        Holm, Michael
        Garcia, Hugo
        Beebe, Ann
        4 total names in list

        Concat based on partial name match "Garcia":
        Garcia, Debra
        Garcia, Hugo
        Garcia, Hugo
        3 total names in list
*/
```

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Comment remplir des collections d'objets issues de plusieurs sources (LINQ) (C#)

Article • 07/04/2023

Cet exemple montre comment fusionner des données de différentes sources en une séquence de nouveaux types.

> **Notes**
>
> N'essayez pas de joindre des données du système de fichiers ou des données en mémoire à des données qui se trouvent encore dans une base de données. Ces jointures interdomaines peuvent générer des résultats indéfinis, en raison des différentes façons par lesquelles les opérations de jointure peuvent être définies pour les requêtes de base de données et autres types de sources. En outre, une telle opération risque de provoquer une exception de mémoire insuffisante si la quantité de données dans la base de données est assez élevée. Pour joindre des données d'une base de données à des données en mémoire, appelez d'abord `ToList` ou `ToArray` sur la requête de base de données, puis effectuez la jointure sur la collection retournée.

Pour créer le fichier de données

Copiez les fichiers names.csv et scores.csv dans votre dossier de projet, comme décrit dans Comment joindre du contenu issu de différents fichiers (LINQ) (C#).

Exemple

L'exemple suivant montre comment utiliser un type nommé `Student` pour stocker des données fusionnées de deux collections en mémoire de chaînes qui simulent des données de feuille de calcul au format .csv. La première collection de chaînes représente les noms et les ID des étudiants, et la deuxième collection représente l'ID d'étudiant (dans la première colonne) et quatre notes d'examen. L'ID est utilisé comme clé étrangère.

```
C#
```

```
using System;
using System.Collections.Generic;
using System.Linq;

class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int ID { get; set; }
    public List<int> ExamScores { get; set; }
}

class PopulateCollection
{
    static void Main()
    {
        // These data files are defined in How to join content from
        // dissimilar files (LINQ).

        // Each line of names.csv consists of a last name, a first name, and
an
        // ID number, separated by commas. For example,
Omelchenko,Svetlana,111
        string[] names = System.IO.File.ReadAllLines(@"../../../names.csv");

        // Each line of scores.csv consists of an ID number and four test
        // scores, separated by commas. For example, 111, 97, 92, 81, 60
        string[] scores =
System.IO.File.ReadAllLines(@"../../../scores.csv");

        // Merge the data sources using a named type.
        // var could be used instead of an explicit type. Note the dynamic
        // creation of a list of ints for the ExamScores member. The first
item
        // is skipped in the split string because it is the student ID,
        // not an exam score.
        IEnumerable<Student> queryNamesScores =
            from nameLine in names
            let splitName = nameLine.Split(',')
            from scoreLine in scores
            let splitScoreLine = scoreLine.Split(',')
            where Convert.ToInt32(splitName[2]) ==
Convert.ToInt32(splitScoreLine[0])
            select new Student()
            {
                FirstName = splitName[0],
                LastName = splitName[1],
                ID = Convert.ToInt32(splitName[2]),
                ExamScores = (from scoreAsText in splitScoreLine.Skip(1)
                              select Convert.ToInt32(scoreAsText)).
                              ToList()
            };

        // Optional. Store the newly created student objects in memory
```

```
        // for faster access in future queries. This could be useful with
        // very large data files.
        List<Student> students = queryNamesScores.ToList();

        // Display each student's name and exam score average.
        foreach (var student in students)
        {
            Console.WriteLine("The average score of {0} {1} is {2}.",
                student.FirstName, student.LastName,
                student.ExamScores.Average());
        }

        //Keep console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    The average score of Omelchenko Svetlana is 82.5.
    The average score of O'Donnell Claire is 72.25.
    The average score of Mortensen Sven is 84.5.
    The average score of Garcia Cesar is 88.25.
    The average score of Garcia Debra is 67.
    The average score of Fakhouri Fadi is 92.25.
    The average score of Feng Hanying is 88.
    The average score of Garcia Hugo is 85.75.
    The average score of Tucker Lance is 81.75.
    The average score of Adams Terry is 85.25.
    The average score of Zabokritski Eugene is 83.
    The average score of Tucker Michael is 92.
 */
```

Dans la clause select, un initialiseur d'objet est utilisé pour instancier chaque nouvel objet `Student` à l'aide des données des deux sources.

Si vous n'êtes pas obligé de stocker les résultats d'une requête, les types anonymes peuvent être plus pratiques que les types nommés. Les types nommés sont nécessaires si vous passez les résultats de requête en dehors de la méthode dans laquelle la requête est exécutée. L'exemple suivant exécute la même tâche que l'exemple précédent, mais utilise des types anonymes plutôt que des types nommés :

C#

```
// Merge the data sources by using an anonymous type.
// Note the dynamic creation of a list of ints for the
// ExamScores member. We skip 1 because the first string
// in the array is the student ID, not an exam score.
var queryNamesScores2 =
    from nameLine in names
    let splitName = nameLine.Split(',')
    from scoreLine in scores
```

```
    let splitScoreLine = scoreLine.Split(',')
    where Convert.ToInt32(splitName[2]) ==
Convert.ToInt32(splitScoreLine[0])
    select new
    {
        First = splitName[0],
        Last = splitName[1],
        ExamScores = (from scoreAsText in splitScoreLine.Skip(1)
                      select Convert.ToInt32(scoreAsText))
                      .ToList()
    };

// Display each student's name and exam score average.
foreach (var student in queryNamesScores2)
{
    Console.WriteLine("The average score of {0} {1} is {2}.",
        student.First, student.Last, student.ExamScores.Average());
}
```

Voir aussi

- LINQ et chaînes (C#)
- Initialiseurs d’objets et de collections
- Types anonymes

Guide pratique pour fractionner un fichier en plusieurs fichiers à l'aide de groupes (LINQ) (C#)

Article • 20/07/2023

Cet exemple montre comment fusionner le contenu de deux fichiers, puis créer un ensemble de fichiers qui organisent les données d'une nouvelle façon.

Pour créer le fichier de données

1. Copiez ces noms dans un fichier texte nommé names1.txt, puis enregistrez-le dans votre dossier de projet :

   ```text
   Bankov, Peter
   Holm, Michael
   Garcia, Hugo
   Potra, Cristina
   Noriega, Fabricio
   Aw, Kam Foo
   Beebe, Ann
   Toyoshima, Tim
   Guy, Wey Yuan
   Garcia, Debra
   ```

2. Copiez ces noms dans un fichier texte nommé names2.txt, puis enregistrez-le dans votre dossier de projet. Notez que les deux fichiers ont certains noms en commun.

   ```text
   Liu, Jinghao
   Bankov, Peter
   Holm, Michael
   Garcia, Hugo
   Beebe, Ann
   Gilchrist, Beth
   Myrcha, Jacek
   Giakoumakis, Leo
   McLin, Nkenge
   El Yassir, Mehdi
   ```

Exemple

C#

```
class SplitWithGroups
{
    static void Main()
    {
        string[] fileA =
System.IO.File.ReadAllLines(@"../../../names1.txt");
        string[] fileB =
System.IO.File.ReadAllLines(@"../../../names2.txt");

        // Concatenate and remove duplicate names based on
        // default string comparer
        var mergeQuery = fileA.Union(fileB);

        // Group the names by the first letter in the last name.
        var groupQuery = from name in mergeQuery
                         let n = name.Split(',')
                         group name by n[0][0] into g
                         orderby g.Key
                         select g;

        // Create a new file for each group that was created
        // Note that nested foreach loops are required to access
        // individual items with each group.
        foreach (var g in groupQuery)
        {
            // Create the new file name.
            string fileName = @"../../../testFile_" + g.Key + ".txt";

            // Output to display.
            Console.WriteLine(g.Key);

            // Write file.
            using (System.IO.StreamWriter sw = new
System.IO.StreamWriter(fileName))
            {
                foreach (var item in g)
                {
                    sw.WriteLine(item);
                    // Output to console for example purposes.
                    Console.WriteLine("   {0}", item);
                }
            }
        }
        // Keep console window open in debug mode.
        Console.WriteLine("Files have been written. Press any key to exit");
        Console.ReadKey();
    }
}
/* Output:
```

```
    A
       Aw, Kam Foo
    B
       Bankov, Peter
       Beebe, Ann
    E
       El Yassir, Mehdi
    G
       Garcia, Hugo
       Guy, Wey Yuan
       Garcia, Debra
       Gilchrist, Beth
       Giakoumakis, Leo
    H
       Holm, Michael
    L
       Liu, Jinghao
    M
       Myrcha, Jacek
       McLin, Nkenge
    N
       Noriega, Fabricio
    P
       Potra, Cristina
    T
       Toyoshima, Tim
*/
```

Le programme écrit un fichier distinct pour chaque groupe, dans le même dossier que les fichiers de données.

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Guide pratique pour joindre du contenu issu de fichiers dissemblables (LINQ) (C#)

Article • 20/07/2023

Cet exemple montre comment joindre des données de deux fichiers CSV qui partagent une valeur commune utilisée comme clé correspondante. Cette technique peut être utile si vous devez combiner les données de deux feuilles de calcul, ou d'une feuille de calcul et d'un fichier qui a un autre format, dans un nouveau fichier. Vous pouvez modifier l'exemple pour qu'il fonctionne avec tout type de texte structuré.

Pour créer le fichier de données

1. Copiez les lignes suivantes dans un fichier nommé *scores.csv*, puis enregistrez-le dans votre dossier de projet. Le fichier représente les données de feuille de calcul. La colonne 1 est l'ID de l'étudiant et les colonnes 2 à 5 correspondent à des résultats d'examens.

   csv

   ```
   111, 97, 92, 81, 60
   112, 75, 84, 91, 39
   113, 88, 94, 65, 91
   114, 97, 89, 85, 82
   115, 35, 72, 91, 70
   116, 99, 86, 90, 94
   117, 93, 92, 80, 87
   118, 92, 90, 83, 78
   119, 68, 79, 88, 92
   120, 99, 82, 81, 79
   121, 96, 85, 91, 60
   122, 94, 92, 91, 91
   ```

2. Copiez les lignes suivantes dans un fichier nommé *names.csv*, puis enregistrez-le dans votre dossier de projet. Le fichier représente une feuille de calcul qui contient le nom, le prénom et l'ID de l'étudiant.

   csv

   ```
   Omelchenko,Svetlana,111
   O'Donnell,Claire,112
   Mortensen,Sven,113
   ```

```
Garcia,Cesar,114
Garcia,Debra,115
Fakhouri,Fadi,116
Feng,Hanying,117
Garcia,Hugo,118
Tucker,Lance,119
Adams,Terry,120
Zabokritski,Eugene,121
Tucker,Michael,122
```

Exemple

C#

```
using System;
using System.Collections.Generic;
using System.Linq;

class JoinStrings
{
    static void Main()
    {
        // Join content from dissimilar files that contain
        // related information. File names.csv contains the student
        // name plus an ID number. File scores.csv contains the ID
        // and a set of four test scores. The following query joins
        // the scores to the student names by using ID as a
        // matching key.

        string[] names = System.IO.File.ReadAllLines(@"../../../names.csv");
        string[] scores =
System.IO.File.ReadAllLines(@"../../../scores.csv");

        // Name:    Last[0],       First[1],  ID[2]
        //          Omelchenko,    Svetlana,  11
        // Score:   StudentID[0],  Exam1[1]   Exam2[2],  Exam3[3],  Exam4[4]
        //          111,           97,        92,        81,        60

        // This query joins two dissimilar spreadsheets based on common ID
value.
        // Multiple from clauses are used instead of a join clause
        // in order to store results of id.Split.
        IEnumerable<string> scoreQuery1 =
            from name in names
            let nameFields = name.Split(',')
            from id in scores
            let scoreFields = id.Split(',')
            where Convert.ToInt32(nameFields[2]) ==
Convert.ToInt32(scoreFields[0])
            select nameFields[0] + "," + scoreFields[1] + "," +
scoreFields[2]
                   + "," + scoreFields[3] + "," + scoreFields[4];
```

```
        // Pass a query variable to a method and execute it
        // in the method. The query itself is unchanged.
        OutputQueryResults(scoreQuery1, "Merge two spreadsheets:");

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    static void OutputQueryResults(IEnumerable<string> query, string
message)
    {
        Console.WriteLine(System.Environment.NewLine + message);
        foreach (string item in query)
        {
            Console.WriteLine(item);
        }
        Console.WriteLine("{0} total names in list", query.Count());
    }
}
/* Output:
Merge two spreadsheets:
Omelchenko, 97, 92, 81, 60
O'Donnell, 75, 84, 91, 39
Mortensen, 88, 94, 65, 91
Garcia, 97, 89, 85, 82
Garcia, 35, 72, 91, 70
Fakhouri, 99, 86, 90, 94
Feng, 93, 92, 80, 87
Garcia, 92, 90, 83, 78
Tucker, 68, 79, 88, 92
Adams, 99, 82, 81, 79
Zabokritski, 96, 85, 91, 60
Tucker, 94, 92, 91, 91
12 total names in list
 */
```

Comment calculer des valeurs de colonnes dans un fichier texte CSV (LINQ) (C#)

Article • 20/07/2023

Cet exemple montre comment effectuer des calculs d'agrégation tels que Sum, Average, Min et Max, avec les colonnes d'un fichier .csv. Les exemples de principes montrés ici sont applicables à d'autres types de texte structuré.

Pour créer le fichier source

1. Copiez les lignes suivantes dans un fichier nommé scores.csv, puis enregistrez-le dans votre dossier de projet. Supposons que la première colonne contienne un ID d'étudiant et que les colonnes suivantes contiennent les résultats de quatre examens.

```csv
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

Exemple

```C#
class SumColumns
{
    static void Main(string[] args)
    {
        string[] lines =
System.IO.File.ReadAllLines(@"../../../scores.csv");
```

```
        // Specifies the column to compute.
        int exam = 3;

        // Spreadsheet format:
        // Student ID    Exam#1  Exam#2  Exam#3  Exam#4
        // 111,          97,     92,     81,     60

        // Add one to exam to skip over the first column,
        // which holds the student ID.
        SingleColumn(lines, exam + 1);
        Console.WriteLine();
        MultiColumns(lines);

        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    static void SingleColumn(IEnumerable<string> strs, int examNum)
    {
        Console.WriteLine("Single Column Query:");

        // Parameter examNum specifies the column to
        // run the calculations on. This value could be
        // passed in dynamically at run time.

        // Variable columnQuery is an IEnumerable<int>.
        // The following query performs two steps:
        // 1) use Split to break each row (a string) into an array
        //    of strings,
        // 2) convert the element at position examNum to an int
        //    and select it.
        var columnQuery =
            from line in strs
            let elements = line.Split(',')
            select Convert.ToInt32(elements[examNum]);

        // Execute the query and cache the results to improve
        // performance. This is helpful only with very large files.
        var results = columnQuery.ToList();

        // Perform aggregate calculations Average, Max, and
        // Min on the column specified by examNum.
        double average = results.Average();
        int max = results.Max();
        int min = results.Min();

        Console.WriteLine("Exam #{0}: Average:{1:##.##} High Score:{2} Low
Score:{3}",
                    examNum, average, max, min);
    }

    static void MultiColumns(IEnumerable<string> strs)
    {
        Console.WriteLine("Multi Column Query:");
```

```
        // Create a query, multiColQuery. Explicit typing is used
        // to make clear that, when executed, multiColQuery produces
        // nested sequences. However, you get the same results by
        // using 'var'.

        // The multiColQuery query performs the following steps:
        // 1) use Split to break each row (a string) into an array
        //    of strings,
        // 2) use Skip to skip the "Student ID" column, and store the
        //    rest of the row in scores.
        // 3) convert each score in the current row from a string to
        //    an int, and select that entire sequence as one row
        //    in the results.
        IEnumerable<IEnumerable<int>> multiColQuery =
            from line in strs
            let elements = line.Split(',')
            let scores = elements.Skip(1)
            select (from str in scores
                    select Convert.ToInt32(str));

        // Execute the query and cache the results to improve
        // performance.
        // ToArray could be used instead of ToList.
        var results = multiColQuery.ToList();

        // Find out how many columns you have in results.
        int columnCount = results[0].Count();

        // Perform aggregate calculations Average, Max, and
        // Min on each column.
        // Perform one iteration of the loop for each column
        // of scores.
        // You can use a for loop instead of a foreach loop
        // because you already executed the multiColQuery
        // query by calling ToList.
        for (int column = 0; column < columnCount; column++)
        {
            var results2 = from row in results
                           select row.ElementAt(column);
            double average = results2.Average();
            int max = results2.Max();
            int min = results2.Min();

            // Add one to column because the first exam is Exam #1,
            // not Exam #0.
            Console.WriteLine("Exam #{0} Average: {1:##.##} High Score: {2}
Low Score: {3}",
                              column + 1, average, max, min);
        }
    }
}
/* Output:
    Single Column Query:
    Exam #4: Average:76.92 High Score:94 Low Score:39
```

```
    Multi Column Query:
    Exam #1 Average: 86.08 High Score: 99 Low Score: 35
    Exam #2 Average: 86.42 High Score: 94 Low Score: 72
    Exam #3 Average: 84.75 High Score: 91 Low Score: 65
    Exam #4 Average: 76.92 High Score: 94 Low Score: 39
*/
```

La requête fonctionne à l'aide de la méthode Split pour convertir chaque ligne de texte en un tableau. Chaque élément du tableau représente une colonne. Enfin, le texte de chaque colonne est converti en sa représentation numérique. Si votre fichier est un fichier séparé par des tabulations, remplacez l'argument de la méthode `Split` par `\t`.

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Comment interroger les métadonnées d'un assembly avec la réflexion (LINQ)

Article • 20/03/2023

Vous utilisez les API de réflexion .NET pour examiner les métadonnées dans un assembly .NET et pour créer des collections de types, de membres de type et de paramètres qui se trouvent dans cet assembly. Comme ces collections prennent en charge l'interface générique IEnumerable<T>, elles peuvent être interrogées à l'aide de LINQ.

L'exemple suivant montre comment utiliser LINQ avec la réflexion pour récupérer des métadonnées spécifiques concernant des méthodes qui correspondent à un critère de recherche spécifié. Ici, la requête recherche les noms de toutes les méthodes dans l'assembly qui retournent des types énumérables tels que des tableaux.

C#

```
Assembly assembly = Assembly.Load("System.Private.CoreLib, Version=7.0.0.0,
Culture=neutral, PublicKeyToken=7cec85d7bea7798e");
var pubTypesQuery = from type in assembly.GetTypes()
                    where type.IsPublic
                    from method in type.GetMethods()
                    where method.ReturnType.IsArray == true
                        || (method.ReturnType.GetInterface(

typeof(System.Collections.Generic.IEnumerable<>).FullName!) != null
                        && method.ReturnType.FullName != "System.String")
                    group method.ToString() by type.ToString();

foreach (var groupOfMethods in pubTypesQuery)
{
    Console.WriteLine("Type: {0}", groupOfMethods.Key);
    foreach (var method in groupOfMethods)
    {
        Console.WriteLine("  {0}", method);
    }
}
```

L'exemple utilise la méthode Assembly.GetTypes pour retourner un tableau de types de l'assembly spécifié. Le filtre where est appliqué pour que seuls des types publics soient retournés. Pour chaque type public, une sous-requête est générée en utilisant le tableau MethodInfo qui est retourné à partir de l'appel Type.GetMethods. Ces résultats sont filtrés pour retourner uniquement les méthodes dont le type de retour est un tableau ou

un type qui implémente IEnumerable<T>. Pour finir, ces résultats sont regroupés en utilisant le nom de type comme clé.

Comment interroger les métadonnées d'un assembly avec la réflexion (LINQ)

Article • 20/03/2023

Vous utilisez les API de réflexion .NET pour examiner les métadonnées dans un assembly .NET et pour créer des collections de types, de membres de type et de paramètres qui se trouvent dans cet assembly. Comme ces collections prennent en charge l'interface générique IEnumerable<T>, elles peuvent être interrogées à l'aide de LINQ.

L'exemple suivant montre comment utiliser LINQ avec la réflexion pour récupérer des métadonnées spécifiques concernant des méthodes qui correspondent à un critère de recherche spécifié. Ici, la requête recherche les noms de toutes les méthodes dans l'assembly qui retournent des types énumérables tels que des tableaux.

C#

```
Assembly assembly = Assembly.Load("System.Private.CoreLib, Version=7.0.0.0,
Culture=neutral, PublicKeyToken=7cec85d7bea7798e");
var pubTypesQuery = from type in assembly.GetTypes()
                    where type.IsPublic
                    from method in type.GetMethods()
                    where method.ReturnType.IsArray == true
                        || (method.ReturnType.GetInterface(

typeof(System.Collections.Generic.IEnumerable<>).FullName!) != null
                        && method.ReturnType.FullName != "System.String")
                    group method.ToString() by type.ToString();

foreach (var groupOfMethods in pubTypesQuery)
{
    Console.WriteLine("Type: {0}", groupOfMethods.Key);
    foreach (var method in groupOfMethods)
    {
        Console.WriteLine("  {0}", method);
    }
}
```

L'exemple utilise la méthode Assembly.GetTypes pour retourner un tableau de types de l'assembly spécifié. Le filtre where est appliqué pour que seuls des types publics soient retournés. Pour chaque type public, une sous-requête est générée en utilisant le tableau MethodInfo qui est retourné à partir de l'appel Type.GetMethods. Ces résultats sont filtrés pour retourner uniquement les méthodes dont le type de retour est un tableau ou

un type qui implémente IEnumerable<T>. Pour finir, ces résultats sont regroupés en utilisant le nom de type comme clé.

LINQ et répertoires de fichiers (C#)

Article • 07/04/2023

De nombreuses opérations du système de fichiers sont des requêtes et l'approche de LINQ leur convient donc bien.

Les requêtes de cette section sont non destructives. Elles ne sont pas utilisées pour modifier le contenu des fichiers ou des dossiers d'origine. La règle suivie est que les requêtes ne doivent pas provoquer des effets secondaires. En règle générale, tout code (y compris les requêtes qui utilisent des opérateurs de création / mise à jour / suppression) qui modifie les données sources doit rester distinct du code qui interroge simplement les données.

Cette section contient les rubriques suivantes :

Guide pratique pour interroger des fichiers ayant un attribut ou un nom donné (C#)
Montre comment rechercher des fichiers en examinant une ou plusieurs propriétés de son objet FileInfo.

Guide pratique pour regrouper des fichiers par extension (LINQ) (C#)
Montre comment retourner des groupes d'objets FileInfo en fonction de leur extension de nom de fichier.

Guide pratique pour rechercher le nombre total d'octets dans un ensemble de dossiers (LINQ) (C#)
Montre comment retourner le nombre total d'octets dans tous les fichiers d'une arborescence de répertoires spécifiée.

Guide pratique pour comparer le contenu de deux dossiers (LINQ) (C#)
Montre comment retourner tous les fichiers qui sont présents dans deux dossiers spécifiés, ainsi que tous les fichiers qui sont présents dans un dossier mais pas dans l'autre.

Guide pratique pour rechercher le ou les fichiers les plus volumineux dans une arborescence de répertoires (LINQ) (C#)
Montre comment retourner le fichier le plus grand ou le plus petit, ou un nombre spécifié de fichiers, dans une arborescence de répertoires.

Guide pratique pour interroger des fichiers dupliqués dans une arborescence de répertoires (LINQ) (C#)
Montre comment regrouper tous les noms de fichier qui se trouvent dans plusieurs emplacements d'une arborescence de répertoires spécifiée. Montre aussi comment effectuer des comparaisons plus complexes avec un comparateur personnalisé.

Guide pratique pour interroger le contenu de fichiers dans un dossier (LINQ) (C#)
Montre comment itérer au sein des dossiers d'une arborescence, ouvrir chaque fichier et interroger le contenu du fichier.

Commentaires

La création d'une source de données représentant avec précision le contenu du système de fichiers et la gestion correcte des exceptions induisent une certaine complexité. Les exemples de cette section créent une collection d'instantanés d'objets FileInfo qui représente tous les fichiers d'un dossier racine spécifié et de tous ses sous-dossiers. L'état réel de chaque FileInfo peut changer dans le temps entre le début et la fin de l'exécution d'une requête. Par exemple, vous pouvez créer une liste d'objets FileInfo à utiliser comme source de données. Si vous essayez d'accéder à la propriété `Length` dans une requête, l'objet FileInfo tente d'accéder au système de fichiers pour mettre à jour la valeur de `Length`. Si le fichier n'existe plus, vous obtenez une exception FileNotFoundException dans votre requête, même si vous n'interrogez pas directement le système de fichiers. Certaines requêtes de cette section utilisent une méthode distincte qui consomme ces exceptions particulières dans certains cas. Une autre option consiste à conserver votre source de données à jour dynamiquement en utilisant FileSystemWatcher.

Voir aussi

- LINQ to Objects (C#)

Guide pratique pour interroger des fichiers ayant un attribut ou un nom donné (C#)

Article • 20/07/2023

Cet exemple montre comment rechercher tous les fichiers ayant une extension de nom de fichier spécifiée (par exemple « .txt ») dans une arborescence de répertoires spécifiée. Il montre également comment retourner le fichier le plus récent ou le plus ancien dans l'arborescence en fonction de l'heure de création.

Exemple

C#

```
class FindFileByExtension
{
    // This query will produce the full path for all .txt files
    // under the specified folder including subfolders.
    // It orders the list according to the file name.
    static void Main()
    {
        string startFolder = @"c:\program files\Microsoft Visual Studio
9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new
System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery
permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

        //Create the query
        IEnumerable<System.IO.FileInfo> fileQuery =
            from file in fileList
            where file.Extension == ".txt"
            orderby file.Name
            select file;

        //Execute the query. This might write out a lot of files!
        foreach (System.IO.FileInfo fi in fileQuery)
        {
            Console.WriteLine(fi.FullName);
        }
```

```
        // Create and execute a new query by using the previous
        // query as a starting point. fileQuery is not
        // executed again until the call to Last()
        var newestFile =
            (from file in fileQuery
             orderby file.CreationTime
             select new { file.FullName, file.CreationTime })
            .Last();

        Console.WriteLine("\r\nThe newest .txt file is {0}. Creation time:
{1}",
            newestFile.FullName, newestFile.CreationTime);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
```

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Guide pratique pour regrouper des fichiers par extension (LINQ) (C#)

Article • 20/07/2023

Cet exemple montre comment utiliser LINQ pour effectuer des opérations de regroupement et de tri avancées sur des listes de fichiers ou de dossiers. Il montre également comment parcourir les résultats dans la fenêtre de console à l'aide des méthodes Skip et Take.

Exemple

La requête suivante montre comment regrouper le contenu d'une arborescence de répertoires spécifiée en fonction de l'extension de nom de fichier.

C#

```
class GroupByExtension
{
    // This query will sort all the files under the specified folder
    //  and subfolder into groups keyed by the file extension.
    private static void Main()
    {
        // Take a snapshot of the file system.
        string startFolder = @"c:\program files\Microsoft Visual Studio
9.0\Common7";

        // Used in WriteLine to trim output lines.
        int trimLength = startFolder.Length;

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new
System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery
permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

        // Create the query.
        var queryGroupByExt =
            from file in fileList
            group file by file.Extension.ToLower() into fileGroup
            orderby fileGroup.Key
            select fileGroup;

        // Display one group at a time. If the number of
```

```
        // entries is greater than the number of lines
        // in the console window, then page the output.
        PageOutput(trimLength, queryGroupByExt);
    }

    // This method specifically handles group queries of FileInfo objects
with string keys.
    // It can be modified to work for any long listings of data. Note that
explicit typing
    // must be used in method signatures. The groupbyExtList parameter is a
query that produces
    // groups of FileInfo objects with string keys.
    private static void PageOutput(int rootLength,

IEnumerable<System.Linq.IGrouping<string, System.IO.FileInfo>>
groupByExtList)
    {
        // Flag to break out of paging loop.
        bool goAgain = true;

        // "3" = 1 line for extension + 1 for "Press any key" + 1 for input
cursor.
        int numLines = Console.WindowHeight - 3;

        // Iterate through the outer collection of groups.
        foreach (var filegroup in groupByExtList)
        {
            // Start a new extension at the top of a page.
            int currentLine = 0;

            // Output only as many lines of the current group as will fit in
the window.
            do
            {
                Console.Clear();
                Console.WriteLine(filegroup.Key == String.Empty ? "[none]" :
filegroup.Key);

                // Get 'numLines' number of items starting at number
'currentLine'.
                var resultPage = filegroup.Skip(currentLine).Take(numLines);

                //Execute the resultPage query
                foreach (var f in resultPage)
                {
                    Console.WriteLine("\t{0}",
f.FullName.Substring(rootLength));
                }

                // Increment the line counter.
                currentLine += numLines;

                // Give the user a chance to escape.
                Console.WriteLine("Press any key to continue or the 'End'
key to break...");
```

```
                ConsoleKey key = Console.ReadKey().Key;
                if (key == ConsoleKey.End)
                {
                    goAgain = false;
                    break;
                }
            } while (currentLine < filegroup.Count());

            if (goAgain == false)
                break;
        }
    }
}
```

La sortie de ce programme peut être longue, en fonction des détails du système de fichiers local et de la valeur de `startFolder`. Pour permettre l'affichage de tous les résultats, cet exemple montre comment parcourir les résultats. Les mêmes techniques sont applicables aux applications Windows et web. Notez qu'étant donné que le code organise les éléments dans un groupe, une boucle `foreach` imbriquée est nécessaire. Il existe également une logique supplémentaire pour calculer la position actuelle dans la liste et pour permettre aux utilisateurs d'arrêter la pagination et de quitter le programme. Dans ce cas particulier, la requête de pagination est exécutée par rapport aux résultats de la requête d'origine mis en cache. Dans d'autres contextes, tels que LINQ to SQL, ce type de mise en cache n'est pas nécessaire.

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Guide pratique pour rechercher le nombre total d'octets dans un ensemble de dossiers (LINQ) (C#)

Article • 20/07/2023

Cet exemple montre comment récupérer le nombre total d'octets utilisés par tous les fichiers d'un dossier spécifié, ainsi que par tous ses sous-dossiers.

Exemple

La méthode Sum ajoute les valeurs de tous les éléments sélectionnés dans la clause `select`. Vous pouvez facilement modifier cette requête pour récupérer le fichier le plus grand ou le plus petit de l'arborescence du répertoire spécifié en appelant la méthode Min ou Max plutôt que Sum.

C#

```
class QuerySize
{
    public static void Main()
    {
        string startFolder = @"c:\program files\Microsoft Visual Studio
9.0\VC#";

        // Take a snapshot of the file system.
        // This method assumes that the application has discovery
permissions
        // for all folders under the specified path.
        IEnumerable<string> fileList =
System.IO.Directory.GetFiles(startFolder, "*.*",
System.IO.SearchOption.AllDirectories);

        var fileQuery = from file in fileList
                        select GetFileLength(file);

        // Cache the results to avoid multiple trips to the file system.
        long[] fileLengths = fileQuery.ToArray();

        // Return the size of the largest file
        long largestFile = fileLengths.Max();

        // Return the total number of bytes in all the files under the
specified folder.
        long totalBytes = fileLengths.Sum();

        Console.WriteLine("There are {0} bytes in {1} files under {2}",
```

```
            totalBytes, fileList.Count(), startFolder);
        Console.WriteLine("The largest files is {0} bytes.", largestFile);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    // This method is used to swallow the possible exception
    // that can be raised when accessing the System.IO.FileInfo.Length
property.
    static long GetFileLength(string filename)
    {
        long retval;
        try
        {
            System.IO.FileInfo fi = new System.IO.FileInfo(filename);
            retval = fi.Length;
        }
        catch (System.IO.FileNotFoundException)
        {
            // If a file is no longer present,
            // just add zero bytes to the total.
            retval = 0;
        }
        return retval;
    }
}
```

Si vous devez seulement compter le nombre d'octets dans une arborescence de répertoires spécifiée, pour être plus efficace, ne créez pas de requête LINQ, car cela induit la surcharge liée à la création de la collection de listes comme source de données. Plus la requête est complexe, plus l'approche LINQ devient utile. C'est également valable lorsque vous devez exécuter plusieurs requêtes sur une même source de données.

La requête appelle une méthode distincte pour obtenir la longueur du fichier. Elle procède ainsi pour utiliser l'exception qui sera éventuellement levée si le fichier a été supprimé sur un autre thread après la création de l'objet FileInfo dans l'appel à `GetFiles`. Même si l'objet FileInfo a déjà été créé, l'exception peut être levée, car un objet FileInfo essaiera d'actualiser sa propriété Length avec la longueur la plus récente lors du premier accès à la propriété. En plaçant cette opération dans un bloc try-catch en dehors de la requête, le code respecte la règle qui consiste à éviter les opérations dans les requêtes qui peuvent avoir des effets secondaires. En règle générale, il faut faire très attention lors de l'utilisation d'exceptions et s'assurer que l'application ne reste pas dans un état inconnu.

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Guide pratique pour comparer le contenu de deux dossiers (LINQ) (C#)

Article • 20/07/2023

Cet exemple montre trois façons de comparer deux listes de fichiers :

- En recherchant une valeur booléenne qui spécifie si les deux listes sont identiques
- En recherchant l'intersection pour récupérer les fichiers qui se trouvent dans les deux dossiers
- En recherchant la différence ensembliste pour récupérer les fichiers qui se trouvent dans un dossier, mais pas dans l'autre

> **Notes**
>
> Les techniques présentées ici peuvent être adaptées pour comparer des séquences d'objets de tout type.

La classe `FileComparer` présentée ici montre comment utiliser une classe de comparateur personnalisée avec les opérateurs de requête standard. La classe n'est pas destinée à être utilisée dans des scénarios réels. Elle utilise simplement le nom et la longueur en octets de chaque fichier pour déterminer si les dossiers ont un contenu identique. Dans un scénario réel, vous devriez modifier ce comparateur pour effectuer une vérification d'égalité plus rigoureuse.

Exemple

C#

```
namespace QueryCompareTwoDirs
{
    class CompareDirs
    {

        static void Main(string[] args)
        {

            // Create two identical or different temporary folders
            // on a local drive and change these file paths.
            string pathA = @"C:\TestDir";
            string pathB = @"C:\TestDir2";
```

```
            System.IO.DirectoryInfo dir1 = new
System.IO.DirectoryInfo(pathA);
            System.IO.DirectoryInfo dir2 = new
System.IO.DirectoryInfo(pathB);

            // Take a snapshot of the file system.
            IEnumerable<System.IO.FileInfo> list1 = dir1.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);
            IEnumerable<System.IO.FileInfo> list2 = dir2.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

            //A custom file comparer defined below
            FileCompare myFileCompare = new FileCompare();

            // This query determines whether the two folders contain
            // identical file lists, based on the custom file comparer
            // that is defined in the FileCompare class.
            // The query executes immediately because it returns a bool.
            bool areIdentical = list1.SequenceEqual(list2, myFileCompare);

            if (areIdentical == true)
            {
                Console.WriteLine("the two folders are the same");
            }
            else
            {
                Console.WriteLine("The two folders are not the same");
            }

            // Find the common files. It produces a sequence and doesn't
            // execute until the foreach statement.
            var queryCommonFiles = list1.Intersect(list2, myFileCompare);

            if (queryCommonFiles.Any())
            {
                Console.WriteLine("The following files are in both
folders:");
                foreach (var v in queryCommonFiles)
                {
                    Console.WriteLine(v.FullName); //shows which items end
up in result list
                }
            }
            else
            {
                Console.WriteLine("There are no common files in the two
folders.");
            }

            // Find the set difference between the two folders.
            // For this example we only check one way.
            var queryList1Only = (from file in list1
                                  select file).Except(list2, myFileCompare);

            Console.WriteLine("The following files are in list1 but not
```

```
list2:");
            foreach (var v in queryList1Only)
            {
                Console.WriteLine(v.FullName);
            }

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }

    // This implementation defines a very simple comparison
    // between two FileInfo objects. It only compares the name
    // of the files being compared and their length in bytes.
    class FileCompare :
System.Collections.Generic.IEqualityComparer<System.IO.FileInfo>
    {
        public FileCompare() { }

        public bool Equals(System.IO.FileInfo f1, System.IO.FileInfo f2)
        {
            return (f1.Name == f2.Name &&
                    f1.Length == f2.Length);
        }

        // Return a hash that reflects the comparison criteria. According to
the
        // rules for IEqualityComparer<T>, if Equals is true, then the hash
codes must
        // also be equal. Because equality as defined here is a simple value
equality, not
        // reference identity, it is possible that two or more objects will
produce the same
        // hash code.
        public int GetHashCode(System.IO.FileInfo fi)
        {
            string s = $"{fi.Name}{fi.Length}";
            return s.GetHashCode();
        }
    }
}
```

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Guide pratique pour rechercher le ou les fichiers les plus volumineux dans une arborescence de répertoires (LINQ) (C#)

Article • 20/07/2023

Cet exemple montre cinq requêtes liées à la taille des fichiers en octets :

- Comment récupérer la taille en octets du plus grand fichier.
- Comment récupérer la taille en octets du plus petit fichier.
- Comment récupérer le plus grand ou le plus petit fichier de l'objet FileInfo dans un ou plusieurs dossiers situés dans le dossier racine spécifié.
- Comment récupérer une séquence, telle que les 10 fichiers les plus volumineux.
- Comment regrouper des fichiers selon leur taille en octets, en ignorant les fichiers qui sont inférieurs à une taille spécifiée.

Exemple

L'exemple suivant contient cinq requêtes distinctes qui montrent comment interroger et regrouper des fichiers selon leur taille en octets. Vous pouvez facilement modifier ces exemples pour baser la requête sur une autre propriété de l'objet FileInfo.

C#

```
class QueryBySize
{
    static void Main(string[] args)
    {
        QueryFilesBySize();
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    private static void QueryFilesBySize()
    {
        string startFolder = @"c:\program files\Microsoft Visual Studio
9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new
System.IO.DirectoryInfo(startFolder);
```

```
        // This method assumes that the application has discovery
permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

        //Return the size of the largest file
        long maxSize =
            (from file in fileList
             let len = GetFileLength(file)
             select len)
             .Max();

        Console.WriteLine("The length of the largest file under {0} is {1}",
            startFolder, maxSize);

        // Return the FileInfo object for the largest file
        // by sorting and selecting from beginning of list
        System.IO.FileInfo longestFile =
            (from file in fileList
             let len = GetFileLength(file)
             where len > 0
             orderby len descending
             select file)
            .First();

        Console.WriteLine("The largest file under {0} is {1} with a length
of {2} bytes",
                          startFolder, longestFile.FullName,
longestFile.Length);

        //Return the FileInfo of the smallest file
        System.IO.FileInfo smallestFile =
            (from file in fileList
             let len = GetFileLength(file)
             where len > 0
             orderby len ascending
             select file).First();

        Console.WriteLine("The smallest file under {0} is {1} with a length
of {2} bytes",
                          startFolder, smallestFile.FullName,
smallestFile.Length);

        //Return the FileInfos for the 10 largest files
        // queryTenLargest is an IEnumerable<System.IO.FileInfo>
        var queryTenLargest =
            (from file in fileList
             let len = GetFileLength(file)
             orderby len descending
             select file).Take(10);

        Console.WriteLine("The 10 largest files under {0} are:",
startFolder);
```

```
        foreach (var v in queryTenLargest)
        {
            Console.WriteLine("{0}: {1} bytes", v.FullName, v.Length);
        }

        // Group the files according to their size, leaving out
        // files that are less than 200000 bytes.
        var querySizeGroups =
            from file in fileList
            let len = GetFileLength(file)
            where len > 0
            group file by (len / 100000) into fileGroup
            where fileGroup.Key >= 2
            orderby fileGroup.Key descending
            select fileGroup;

        foreach (var filegroup in querySizeGroups)
        {
            Console.WriteLine(filegroup.Key.ToString() + "00000");
            foreach (var item in filegroup)
            {
                Console.WriteLine("\t{0}: {1}", item.Name, item.Length);
            }
        }
    }

    // This method is used to swallow the possible exception
    // that can be raised when accessing the FileInfo.Length property.
    // In this particular case, it is safe to swallow the exception.
    static long GetFileLength(System.IO.FileInfo fi)
    {
        long retval;
        try
        {
            retval = fi.Length;
        }
        catch (System.IO.FileNotFoundException)
        {
            // If a file is no longer present,
            // just add zero bytes to the total.
            retval = 0;
        }
        return retval;
    }

}
```

Pour retourner un ou plusieurs objets FileInfo complets, la requête doit d'abord examiner chacun d'eux dans la source de données, puis les trier selon la valeur de leur propriété Length. Elle peut ensuite retourner l'objet ou la séquence d'objets avec les plus grandes valeurs de longueur. Utilisez First pour retourner le premier élément d'une

liste. Utilisez Take pour retourner les n premiers éléments. Spécifiez un ordre de tri décroissant pour placer les plus petits éléments au début de la liste.

La requête appelle une méthode distincte pour obtenir la taille du fichier en octets et ainsi permettre l'utilisation de l'exception éventuellement levée si un fichier a été supprimé sur un autre thread depuis la création de l'objet FileInfo dans l'appel à `GetFiles`. Même si l'objet FileInfo a déjà été créé, l'exception peut être levée, car un objet FileInfo essaiera d'actualiser sa propriété Length en utilisant la taille en octets la plus récente lors du premier accès à la propriété. En plaçant cette opération dans un bloc try-catch en dehors de la requête, nous respectons la règle qui consiste à éviter les opérations dans les requêtes qui peuvent avoir des effets secondaires. En règle générale, il faut faire très attention lors de l'utilisation d'exceptions et s'assurer que l'application ne reste pas dans un état inconnu.

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Guide pratique pour interroger des fichiers dupliqués dans une arborescence de répertoires (LINQ) (C#)

Article • 20/07/2023

Parfois, plusieurs dossiers peuvent contenir des fichiers ayant le même nom. Par exemple, sous le dossier d'installation de Visual Studio, plusieurs dossiers ont un fichier readme.htm. Cet exemple montre comment rechercher ces noms de fichiers dupliqués sous un dossier racine spécifié. Le deuxième exemple montre comment rechercher des fichiers dont la taille et l'heure de création correspondent également.

Exemple

C#

```
class QueryDuplicateFileNames
{
    static void Main(string[] args)
    {
        // Uncomment QueryDuplicates2 to run that query.
        QueryDuplicates();
        // QueryDuplicates2();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    static void QueryDuplicates()
    {
        // Change the root drive or folder if necessary
        string startFolder = @"c:\program files\Microsoft Visual Studio
9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new
System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery
permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

        // used in WriteLine to keep the lines shorter
        int charsToSkip = startFolder.Length;
```

```
        // var can be used for convenience with groups.
        var queryDupNames =
            from file in fileList
            group file.FullName.Substring(charsToSkip) by file.Name into
fileGroup
            where fileGroup.Count() > 1
            select fileGroup;

        // Pass the query to a method that will
        // output one page at a time.
        PageOutput<string, string>(queryDupNames);
    }

    // A Group key that can be passed to a separate method.
    // Override Equals and GetHashCode to define equality for the key.
    // Override ToString to provide a friendly name for Key.ToString()
    class PortableKey
    {
        public string Name { get; set; }
        public DateTime LastWriteTime { get; set; }
        public long Length { get; set; }

        public override bool Equals(object obj)
        {
            PortableKey other = (PortableKey)obj;
            return other.LastWriteTime == this.LastWriteTime &&
                   other.Length == this.Length &&
                   other.Name == this.Name;
        }

        public override int GetHashCode()
        {
            string str = $"{this.LastWriteTime}{this.Length}{this.Name}";
            return str.GetHashCode();
        }
        public override string ToString()
        {
            return $"{this.Name} {this.Length} {this.LastWriteTime}";
        }
    }
    static void QueryDuplicates2()
    {
        // Change the root drive or folder if necessary.
        string startFolder = @"c:\program files\Microsoft Visual Studio
9.0\Common7";

        // Make the lines shorter for the console display
        int charsToSkip = startFolder.Length;

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new
System.IO.DirectoryInfo(startFolder);
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);
```

```
        // Note the use of a compound key. Files that match
        // all three properties belong to the same group.
        // A named type is used to enable the query to be
        // passed to another method. Anonymous types can also be used
        // for composite keys but cannot be passed across method boundaries
        //
        var queryDupFiles =
            from file in fileList
            group file.FullName.Substring(charsToSkip) by
                new PortableKey { Name = file.Name, LastWriteTime =
file.LastWriteTime, Length = file.Length } into fileGroup
            where fileGroup.Count() > 1
            select fileGroup;

        var list = queryDupFiles.ToList();

        int i = queryDupFiles.Count();

        PageOutput<PortableKey, string>(queryDupFiles);
    }

    // A generic method to page the output of the QueryDuplications methods
    // Here the type of the group must be specified explicitly. "var" cannot
    // be used in method signatures. This method does not display more than
one
    // group per page.
    private static void PageOutput<K, V>
(IEnumerable<System.Linq.IGrouping<K, V>> groupByExtList)
    {
        // Flag to break out of paging loop.
        bool goAgain = true;

        // "3" = 1 line for extension + 1 for "Press any key" + 1 for input
cursor.
        int numLines = Console.WindowHeight - 3;

        // Iterate through the outer collection of groups.
        foreach (var filegroup in groupByExtList)
        {
            // Start a new extension at the top of a page.
            int currentLine = 0;

            // Output only as many lines of the current group as will fit in
the window.
            do
            {
                Console.Clear();
                Console.WriteLine("Filename = {0}", filegroup.Key.ToString()
== String.Empty ? "[none]" : filegroup.Key.ToString());

                // Get 'numLines' number of items starting at number
'currentLine'.
                var resultPage = filegroup.Skip(currentLine).Take(numLines);
```

```
                //Execute the resultPage query
                foreach (var fileName in resultPage)
                {
                    Console.WriteLine("\t{0}", fileName);
                }

                // Increment the line counter.
                currentLine += numLines;

                // Give the user a chance to escape.
                Console.WriteLine("Press any key to continue or the 'End'
key to break...");
                ConsoleKey key = Console.ReadKey().Key;
                if (key == ConsoleKey.End)
                {
                    goAgain = false;
                    break;
                }
            } while (currentLine < filegroup.Count());

            if (goAgain == false)
                break;
        }
    }
}
```

La première requête utilise une clé simple pour déterminer une correspondance ; elle trouve les fichiers qui ont le même nom mais dont le contenu peut être différent. La deuxième requête utilise une clé composée à comparer à trois propriétés de l'objet FileInfo. Cette requête est beaucoup plus susceptible de trouver les fichiers ayant le même nom et un contenu similaire ou identique.

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Guide pratique pour interroger le contenu de fichiers texte dans un dossier (LINQ) (C#)

Article • 20/07/2023

Cet exemple montre comment interroger tous les fichiers d'une arborescence de répertoires spécifiée, comment ouvrir chaque fichier et comment inspecter son contenu. Ce type de technique peut être utilisé pour créer des index ou des index inversés à partir du contenu d'une arborescence de répertoires. Dans cet exemple, une recherche de chaîne simple est effectuée. Toutefois, il est possible d'effectuer une recherche avec des critères spéciaux plus complexes à l'aide d'une expression régulière. Pour plus d'informations, consultez Guide pratique pour combiner des requêtes LINQ avec des expressions régulières (C#).

Exemple

C#

```
class QueryContents
{
    public static void Main()
    {
        // Modify this path as necessary.
        string startFolder = @"c:\program files\Microsoft Visual Studio
9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new
System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery
permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

        string searchTerm = @"Visual Studio";

        // Search the contents of each file.
        // A regular expression created with the RegEx class
        // could be used instead of the Contains method.
        // queryMatchingFiles is an IEnumerable<string>.
        var queryMatchingFiles =
            from file in fileList
            where file.Extension == ".htm"
```

```
            let fileText = GetFileText(file.FullName)
            where fileText.Contains(searchTerm)
            select file.FullName;

        // Execute the query.
        Console.WriteLine("The term \"{0}\" was found in:", searchTerm);
        foreach (string filename in queryMatchingFiles)
        {
            Console.WriteLine(filename);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // Read the contents of the file.
    static string GetFileText(string name)
    {
        string fileContents = String.Empty;

        // If the file has been deleted since we took
        // the snapshot, ignore it and return the empty string.
        if (System.IO.File.Exists(name))
        {
            fileContents = System.IO.File.ReadAllText(name);
        }
        return fileContents;
    }
}
```

Compilation du code

Créez un projet d'application console C# avec des directives `using` pour les espaces de noms System.Linq et System.IO.

Guide pratique pour interroger un ArrayList avec LINQ (C#)

Article • 20/07/2023

Quand vous utilisez LINQ pour interroger des collections IEnumerable non génériques telles que ArrayList, vous devez déclarer explicitement le type de la variable de portée pour qu'il reflète le type spécifique des objets de la collection. Par exemple, si vous avez un ArrayList d'objets `Student`, votre clause from doit ressembler à ceci :

C#

```
var query = from Student s in arrList
//...
```

En spécifiant le type de la variable de portée, vous effectuez un cast de chaque élément du `Student` en ArrayList.

L'utilisation d'une variable de portée explicitement typée dans une expression de requête équivaut à appeler la méthode Cast. Cast lève une exception si le cast spécifié ne peut pas être effectué. Cast et OfType sont les deux méthodes d'opérateur de requête standard qui fonctionnent sur les types IEnumerable non génériques. Pour plus d'informations, consultez Relations des types dans des opérations de requête LINQ.

Exemple

L'exemple suivant montre une requête simple sur un ArrayList. Notez que cet exemple utilise des initialiseurs d'objets quand le code appelle la méthode Add, mais cela n'est pas une obligation.

C#

```
using System;
using System.Collections;
using System.Linq;

namespace NonGenericLINQ
{
    public class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int[] Scores { get; set; }
    }
```

```
    class Program
    {
        static void Main(string[] args)
        {
            ArrayList arrList = new ArrayList();
            arrList.Add(
                new Student
                    {
                        FirstName = "Svetlana", LastName = "Omelchenko",
Scores = new int[] { 98, 92, 81, 60 }
                    });
            arrList.Add(
                new Student
                    {
                        FirstName = "Claire", LastName = "O'Donnell", Scores
= new int[] { 75, 84, 91, 39 }
                    });
            arrList.Add(
                new Student
                    {
                        FirstName = "Sven", LastName = "Mortensen", Scores =
new int[] { 88, 94, 65, 91 }
                    });
            arrList.Add(
                new Student
                    {
                        FirstName = "Cesar", LastName = "Garcia", Scores =
new int[] { 97, 89, 85, 82 }
                    });

            var query = from Student student in arrList
                        where student.Scores[0] > 95
                        select student;

            foreach (Student s in query)
                Console.WriteLine(s.LastName + ": " + s.Scores[0]);

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }
}
/* Output:
    Omelchenko: 98
    Garcia: 97
*/
```

Guide pratique pour ajouter des méthodes personnalisées aux requêtes LINQ (C#)

Article • 07/04/2023

Vous étendez l'ensemble de méthodes que vous utilisez pour les requêtes LINQ en ajoutant des méthodes d'extension à l'interface IEnumerable<T>. Par exemple, en plus des opérations moyennes ou maximales standard, vous créez une méthode d'agrégation personnalisée pour calculer une valeur unique à partir d'une séquence de valeurs. Vous pouvez également créer une méthode qui fonctionne comme un filtre personnalisé ou une transformation de données pour une séquence de valeurs, et qui retourne une nouvelle séquence. Distinct, Skip et Reverse en sont quelques exemples.

Quand vous étendez l'interface IEnumerable<T>, vous pouvez appliquer vos méthodes personnalisées à n'importe quelle collection énumérable. Pour plus d'informations, consultez Méthodes d'extension.

Ajouter une méthode d'agrégation

Une méthode d'agrégation calcule une valeur à partir d'un ensemble de valeurs. LINQ fournit plusieurs méthodes d'agrégation, notamment Average, Min et Max. Vous pouvez créer votre propre méthode d'agrégation en ajoutant une méthode d'extension à l'interface IEnumerable<T>.

L'exemple de code suivant montre comment créer une méthode d'extension appelée `Median` pour calculer une valeur médiane pour une séquence de nombres de type `double`.

C#

```
public static class EnumerableExtension
{
    public static double Median(this IEnumerable<double>? source)
    {
        if (source is null || !source.Any())
        {
            throw new InvalidOperationException("Cannot compute median for a
null or empty set.");
        }

        var sortedList =
            source.OrderBy(number => number).ToList();
```

```
        int itemIndex = sortedList.Count / 2;

        if (sortedList.Count % 2 == 0)
        {
            // Even number of items.
            return (sortedList[itemIndex] + sortedList[itemIndex - 1]) / 2;
        }
        else
        {
            // Odd number of items.
            return sortedList[itemIndex];
        }
    }
}
```

Vous appelez cette méthode d'extension pour toute collection énumérable de la même façon que vous appelez d'autres méthodes d'agrégation depuis l'interface IEnumerable<T>.

L'exemple de code suivant montre comment utiliser la méthode `Median` pour un tableau de type `double`.

C#

```
double[] numbers = { 1.9, 2, 8, 4, 5.7, 6, 7.2, 0 };
var query = numbers.Median();

Console.WriteLine($"double: Median = {query}");
// This code produces the following output:
//     double: Median = 4.85
```

Surcharger une méthode d'agrégation pour accepter différents types

Vous pouvez surcharger votre méthode d'agrégation pour qu'elle accepte des séquences de différents types. L'approche standard consiste à créer une surcharge pour chaque type. Une autre approche consiste à créer une surcharge qui accepte un type générique et le convertit en un autre type à l'aide d'un délégué. Vous pouvez également combiner ces deux approches.

Créer une surcharge pour chaque type

Vous pouvez créer une surcharge pour chacun des types que vous voulez prendre en charge. L'exemple de code suivant montre une surcharge de la méthode `Median` pour le

type `int`.

```
C#
```

```
// int overload
public static double Median(this IEnumerable<int> source) =>
    (from number in source select (double)number).Median();
```

Vous pouvez maintenant appeler les surcharges `Median` pour les types `integer` et `double`, comme indiqué dans le code suivant :

```
C#
```

```
double[] numbers1 = { 1.9, 2, 8, 4, 5.7, 6, 7.2, 0 };
var query1 = numbers1.Median();

Console.WriteLine($"double: Median = {query1}");

int[] numbers2 = { 1, 2, 3, 4, 5 };
var query2 = numbers2.Median();

Console.WriteLine($"int: Median = {query2}");
// This code produces the following output:
//     double: Median = 4.85
//     int: Median = 3
```

Créer une surcharge générique

Vous pouvez également créer une surcharge qui accepte une séquence d'objets génériques. Cette surcharge prend un délégué comme paramètre et l'utilise pour convertir une séquence d'objets de type générique en un autre type d'objets.

Le code suivant montre une surcharge de la méthode `Median` qui prend le délégué Func<T,TResult> comme paramètre. Ce délégué prend un objet de type générique T et retourne un objet de type `double`.

```
C#
```

```
// generic overload
public static double Median<T>(
    this IEnumerable<T> numbers, Func<T, double> selector) =>
    (from num in numbers select selector(num)).Median();
```

Vous pouvez maintenant appeler la méthode `Median` pour une séquence d'objets de tout type. Si le type n'a pas sa propre surcharge de méthode, vous devez passer un

paramètre délégué. En C#, vous pouvez utiliser une expression lambda à cet effet. En outre, en Visual Basic uniquement, si vous utilisez la clause `Aggregate` ou `Group By` au lieu de l'appel de méthode, vous pouvez passer n'importe quelle valeur ou expression qui se trouve dans la portée de cette clause.

L'exemple de code suivant montre comment appeler la méthode `Median` pour un tableau d'entiers et un tableau de chaînes. Pour les chaînes, c'est la valeur médiane des longueurs de chaînes du tableau qui est calculée. L'exemple montre comment passer le paramètre de délégué Func<T,TResult> à la méthode `Median` pour chaque cas.

C#

```
int[] numbers3 = { 1, 2, 3, 4, 5 };

/*
    You can use the num => num lambda expression as a parameter for the
Median method
    so that the compiler will implicitly convert its value to double.
    If there is no implicit conversion, the compiler will display an error
message.
*/
var query3 = numbers3.Median(num => num);

Console.WriteLine($"int: Median = {query3}");

string[] numbers4 = { "one", "two", "three", "four", "five" };

// With the generic overload, you can also use numeric properties of
objects.
var query4 = numbers4.Median(str => str.Length);

Console.WriteLine($"string: Median = {query4}");
// This code produces the following output:
//     int: Median = 3
//     string: Median = 4
```

Ajouter une méthode qui retourne une séquence

Vous pouvez étendre l'interface IEnumerable<T> avec une méthode de requête personnalisée qui retourne une séquence de valeurs. Dans ce cas, la méthode doit retourner une collection de type IEnumerable<T>. Ces méthodes peuvent être utilisées pour appliquer des filtres ou des transformations de données à une séquence de valeurs.

L'exemple suivant montre comment créer une méthode d'extension nommée `AlternateElements` qui retourne un élément sur deux d'une collection, en commençant par le premier élément.

C#

```
// Extension method for the IEnumerable<T> interface.
// The method returns every other element of a sequence.
public static IEnumerable<T> AlternateElements<T>(this IEnumerable<T>
source)
{
    int index = 0;
    foreach (T element in source)
    {
        if (index % 2 == 0)
        {
            yield return element;
        }

        index++;
    }
}
```

Vous pouvez appeler cette méthode d'extension pour n'importe quelle collection énumérable, de la même façon que vous appelez d'autres méthodes depuis l'interface IEnumerable<T>, comme le montre le code suivant :

C#

```
string[] strings = { "a", "b", "c", "d", "e" };

var query5 = strings.AlternateElements();

foreach (var element in query5)
{
    Console.WriteLine(element);
}
// This code produces the following output:
//     a
//     c
//     e
```

Voir aussi

- IEnumerable<T>
- Méthodes d'extension

LINQ (Language-Integrated Query)

Article • 24/02/2023

LINQ (Language-Integrated Query) est le nom d'un ensemble de technologies basé sur l'intégration de fonctions de requête directement dans le langage C#. En règle générale, les requêtes de données sont exprimées comme de simples chaînes, sans vérification de type au moment de l'exécution, ni prise en charge IntelliSense. En outre, vous devez apprendre un langage de requête différent pour chaque type de source de données : bases de données SQL, documents XML, services web, etc. Avec LINQ, une requête est une construction de langage de premier ordre, comme les classes, les méthodes et les événements.

Pour un développeur qui écrit des requêtes, la partie « intégrée au langage » la plus visible de LINQ est l'expression de requête. Les expressions de requête sont écrites selon une *syntaxe de requête* déclarative. En utilisant la syntaxe de requête, vous pouvez effectuer des opérations de filtrage, de classement et de regroupement sur des sources de données avec un minimum de code. Vous utilisez les mêmes modèles d'expression de requête de base pour interroger et transformer des données dans les bases de données SQL, les jeux de données ADO .NET, les flux et documents XML et les collections .NET.

L'exemple suivant montre l'opération de requête complète. L'opération complète comprend la création d'une source de données, la définition de l'expression de requête et l'exécution de la requête dans une instruction `foreach`.

C#

```
// Specify the data source.
int[] scores = { 97, 92, 81, 60 };

// Define the query expression.
IEnumerable<int> scoreQuery =
    from score in scores
    where score > 80
    select score;

// Execute the query.
foreach (int i in scoreQuery)
{
    Console.Write(i + " ");
}

// Output: 97 92 81
```

Vue d'ensemble des expressions de requête

- Les expressions de requête peuvent être utilisées pour interroger et transformer des données à partir de n'importe quelle source de données compatible LINQ. Par exemple, une même requête peut récupérer des données d'une base de données SQL et générer un flux XML en sortie.

- Les expressions de requête sont faciles à saisir car elles utilisent de nombreuses constructions familières du langage C#.

- Les variables d'une expression de requête sont toutes fortement typées, même si, dans de nombreux cas, il n'est pas nécessaire de fournir le type explicitement, car le compilateur peut le déduire. Pour plus d'informations, consultez la page Relations entre les types dans les opérations de requête LINQ.

- Une requête ne s'exécute pas tant que vous n'avez pas itéré la variable de requête, par exemple dans une instruction `foreach`. Pour plus d'informations, consultez Introduction aux requêtes LINQ.

- Lors de la compilation, les expressions de requête sont converties en appels de méthode d'opérateur de requête standard en fonction des règles définies dans la spécification du langage C#. Toute requête exprimable avec la syntaxe de requête peut également être exprimée avec la syntaxe de méthode. Toutefois, dans la plupart des cas, la syntaxe de requête est plus lisible et plus concise. Pour plus d'informations, consultez les pages Spécification du langage C# et Vue d'ensemble des opérateurs de requête standard.

- En règle générale, lorsque vous écrivez des requêtes LINQ, nous vous recommandons d'utiliser la syntaxe de requête dans la mesure du possible et la syntaxe de méthode si nécessaire. Il n'y a aucune différence de sémantique ou de performances entre les deux formats. Les expressions de requête sont souvent plus lisibles que les expressions équivalentes écrites avec la syntaxe de méthode.

- Certaines opérations de requête, comme Count ou Max, n'ont pas d'expression de requête équivalente et doivent par conséquent être exprimées sous la forme d'un appel de méthode. La syntaxe de méthode peut être combinée avec la syntaxe de requête de différentes manières. Pour plus d'informations, consultez la page Syntaxe de requête et syntaxe de méthode dans LINQ.

- Les expressions de requête peuvent être compilées dans des arborescences d'expressions ou des délégués, selon le type auquel la requête est appliquée. Les requêtes IEnumerable<T> sont compilées en délégués. Les requêtes IQueryable et

IQueryable<T> sont compilées en arborescences d'expression. Pour plus d'informations, consultez la page Arborescences d'expressions.

Étapes suivantes

Pour approfondir votre connaissance de LINQ, commencez par vous familiariser avec certains concepts de base expliqués sur la page Principes de base des expressions de requête, puis lisez la documentation relative à la technologie LINQ qui vous intéresse :

- Documents XML : LINQ to XML
- ADO.NET Entity Framework : LINQ to Entities
- Collections, fichiers, chaînes, etc. .NET : LINQ to Objects

Pour approfondir votre compréhension de LINQ en général, consultez la page LINQ en C#.

Pour commencer à travailler avec LINQ en C#, consultez le didacticiel Travailler avec LINQ.

LINQ (Language-Integrated Query)

Article • 24/02/2023

LINQ (Language-Integrated Query) est le nom d'un ensemble de technologies basé sur l'intégration de fonctions de requête directement dans le langage C#. En règle générale, les requêtes de données sont exprimées comme de simples chaînes, sans vérification de type au moment de l'exécution, ni prise en charge IntelliSense. En outre, vous devez apprendre un langage de requête différent pour chaque type de source de données : bases de données SQL, documents XML, services web, etc. Avec LINQ, une requête est une construction de langage de premier ordre, comme les classes, les méthodes et les événements.

Pour un développeur qui écrit des requêtes, la partie « intégrée au langage » la plus visible de LINQ est l'expression de requête. Les expressions de requête sont écrites selon une *syntaxe de requête* déclarative. En utilisant la syntaxe de requête, vous pouvez effectuer des opérations de filtrage, de classement et de regroupement sur des sources de données avec un minimum de code. Vous utilisez les mêmes modèles d'expression de requête de base pour interroger et transformer des données dans les bases de données SQL, les jeux de données ADO .NET, les flux et documents XML et les collections .NET.

L'exemple suivant montre l'opération de requête complète. L'opération complète comprend la création d'une source de données, la définition de l'expression de requête et l'exécution de la requête dans une instruction `foreach`.

C#

```
// Specify the data source.
int[] scores = { 97, 92, 81, 60 };

// Define the query expression.
IEnumerable<int> scoreQuery =
    from score in scores
    where score > 80
    select score;

// Execute the query.
foreach (int i in scoreQuery)
{
    Console.Write(i + " ");
}

// Output: 97 92 81
```

Vue d'ensemble des expressions de requête

- Les expressions de requête peuvent être utilisées pour interroger et transformer des données à partir de n'importe quelle source de données compatible LINQ. Par exemple, une même requête peut récupérer des données d'une base de données SQL et générer un flux XML en sortie.

- Les expressions de requête sont faciles à saisir car elles utilisent de nombreuses constructions familières du langage C#.

- Les variables d'une expression de requête sont toutes fortement typées, même si, dans de nombreux cas, il n'est pas nécessaire de fournir le type explicitement, car le compilateur peut le déduire. Pour plus d'informations, consultez la page Relations entre les types dans les opérations de requête LINQ.

- Une requête ne s'exécute pas tant que vous n'avez pas itéré la variable de requête, par exemple dans une instruction `foreach`. Pour plus d'informations, consultez Introduction aux requêtes LINQ.

- Lors de la compilation, les expressions de requête sont converties en appels de méthode d'opérateur de requête standard en fonction des règles définies dans la spécification du langage C#. Toute requête exprimable avec la syntaxe de requête peut également être exprimée avec la syntaxe de méthode. Toutefois, dans la plupart des cas, la syntaxe de requête est plus lisible et plus concise. Pour plus d'informations, consultez les pages Spécification du langage C# et Vue d'ensemble des opérateurs de requête standard.

- En règle générale, lorsque vous écrivez des requêtes LINQ, nous vous recommandons d'utiliser la syntaxe de requête dans la mesure du possible et la syntaxe de méthode si nécessaire. Il n'y a aucune différence de sémantique ou de performances entre les deux formats. Les expressions de requête sont souvent plus lisibles que les expressions équivalentes écrites avec la syntaxe de méthode.

- Certaines opérations de requête, comme Count ou Max, n'ont pas d'expression de requête équivalente et doivent par conséquent être exprimées sous la forme d'un appel de méthode. La syntaxe de méthode peut être combinée avec la syntaxe de requête de différentes manières. Pour plus d'informations, consultez la page Syntaxe de requête et syntaxe de méthode dans LINQ.

- Les expressions de requête peuvent être compilées dans des arborescences d'expressions ou des délégués, selon le type auquel la requête est appliquée. Les requêtes IEnumerable<T> sont compilées en délégués. Les requêtes IQueryable et

IQueryable<T> sont compilées en arborescences d'expression. Pour plus d'informations, consultez la page Arborescences d'expressions.

Étapes suivantes

Pour approfondir votre connaissance de LINQ, commencez par vous familiariser avec certains concepts de base expliqués sur la page Principes de base des expressions de requête, puis lisez la documentation relative à la technologie LINQ qui vous intéresse :

- Documents XML : LINQ to XML
- ADO.NET Entity Framework : LINQ to Entities
- Collections, fichiers, chaînes, etc. .NET : LINQ to Objects

Pour approfondir votre compréhension de LINQ en général, consultez la page LINQ en C#.

Pour commencer à travailler avec LINQ en C#, consultez le didacticiel Travailler avec LINQ.

LINQ (Language-Integrated Query)

Article • 24/02/2023

LINQ (Language-Integrated Query) est le nom d'un ensemble de technologies basé sur l'intégration de fonctions de requête directement dans le langage C#. En règle générale, les requêtes de données sont exprimées comme de simples chaînes, sans vérification de type au moment de l'exécution, ni prise en charge IntelliSense. En outre, vous devez apprendre un langage de requête différent pour chaque type de source de données : bases de données SQL, documents XML, services web, etc. Avec LINQ, une requête est une construction de langage de premier ordre, comme les classes, les méthodes et les événements.

Pour un développeur qui écrit des requêtes, la partie « intégrée au langage » la plus visible de LINQ est l'expression de requête. Les expressions de requête sont écrites selon une *syntaxe de requête* déclarative. En utilisant la syntaxe de requête, vous pouvez effectuer des opérations de filtrage, de classement et de regroupement sur des sources de données avec un minimum de code. Vous utilisez les mêmes modèles d'expression de requête de base pour interroger et transformer des données dans les bases de données SQL, les jeux de données ADO .NET, les flux et documents XML et les collections .NET.

L'exemple suivant montre l'opération de requête complète. L'opération complète comprend la création d'une source de données, la définition de l'expression de requête et l'exécution de la requête dans une instruction `foreach`.

C#

```
// Specify the data source.
int[] scores = { 97, 92, 81, 60 };

// Define the query expression.
IEnumerable<int> scoreQuery =
    from score in scores
    where score > 80
    select score;

// Execute the query.
foreach (int i in scoreQuery)
{
    Console.Write(i + " ");
}

// Output: 97 92 81
```

Vue d'ensemble des expressions de requête

- Les expressions de requête peuvent être utilisées pour interroger et transformer des données à partir de n'importe quelle source de données compatible LINQ. Par exemple, une même requête peut récupérer des données d'une base de données SQL et générer un flux XML en sortie.

- Les expressions de requête sont faciles à saisir car elles utilisent de nombreuses constructions familières du langage C#.

- Les variables d'une expression de requête sont toutes fortement typées, même si, dans de nombreux cas, il n'est pas nécessaire de fournir le type explicitement, car le compilateur peut le déduire. Pour plus d'informations, consultez la page Relations entre les types dans les opérations de requête LINQ.

- Une requête ne s'exécute pas tant que vous n'avez pas itéré la variable de requête, par exemple dans une instruction `foreach`. Pour plus d'informations, consultez Introduction aux requêtes LINQ.

- Lors de la compilation, les expressions de requête sont converties en appels de méthode d'opérateur de requête standard en fonction des règles définies dans la spécification du langage C#. Toute requête exprimable avec la syntaxe de requête peut également être exprimée avec la syntaxe de méthode. Toutefois, dans la plupart des cas, la syntaxe de requête est plus lisible et plus concise. Pour plus d'informations, consultez les pages Spécification du langage C# et Vue d'ensemble des opérateurs de requête standard.

- En règle générale, lorsque vous écrivez des requêtes LINQ, nous vous recommandons d'utiliser la syntaxe de requête dans la mesure du possible et la syntaxe de méthode si nécessaire. Il n'y a aucune différence de sémantique ou de performances entre les deux formats. Les expressions de requête sont souvent plus lisibles que les expressions équivalentes écrites avec la syntaxe de méthode.

- Certaines opérations de requête, comme Count ou Max, n'ont pas d'expression de requête équivalente et doivent par conséquent être exprimées sous la forme d'un appel de méthode. La syntaxe de méthode peut être combinée avec la syntaxe de requête de différentes manières. Pour plus d'informations, consultez la page Syntaxe de requête et syntaxe de méthode dans LINQ.

- Les expressions de requête peuvent être compilées dans des arborescences d'expressions ou des délégués, selon le type auquel la requête est appliquée. Les requêtes IEnumerable<T> sont compilées en délégués. Les requêtes IQueryable et

IQueryable<T> sont compilées en arborescences d'expression. Pour plus d'informations, consultez la page Arborescences d'expressions.

Étapes suivantes

Pour approfondir votre connaissance de LINQ, commencez par vous familiariser avec certains concepts de base expliqués sur la page Principes de base des expressions de requête, puis lisez la documentation relative à la technologie LINQ qui vous intéresse :

- Documents XML : LINQ to XML
- ADO.NET Entity Framework : LINQ to Entities
- Collections, fichiers, chaînes, etc. .NET : LINQ to Objects

Pour approfondir votre compréhension de LINQ en général, consultez la page LINQ en C#.

Pour commencer à travailler avec LINQ en C#, consultez le didacticiel Travailler avec LINQ.

Attributes

Article • 03/15/2023

Attributes provide a powerful method of associating metadata, or declarative information, with code (assemblies, types, methods, properties, and so forth). After an attribute is associated with a program entity, the attribute can be queried at run time by using a technique called *reflection*.

Attributes have the following properties:

- Attributes add metadata to your program. *Metadata* is information about the types defined in a program. All .NET assemblies contain a specified set of metadata that describes the types and type members defined in the assembly. You can add custom attributes to specify any additional information that is required.
- You can apply one or more attributes to entire assemblies, modules, or smaller program elements such as classes and properties.
- Attributes can accept arguments in the same way as methods and properties.
- Your program can examine its own metadata or the metadata in other programs by using reflection.

Reflection provides objects (of type Type) that describe assemblies, modules, and types. You can use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties. If you're using attributes in your code, reflection enables you to access them. For more information, see Attributes.

Here's a simple example of reflection using the GetType() method - inherited by all types from the `Object` base class - to obtain the type of a variable:

> **Note**
>
> Make sure you add `using System;` and `using System.Reflection;` at the top of your *.cs* file.

C#

```
// Using GetType to obtain type information:
int i = 42;
Type type = i.GetType();
Console.WriteLine(type);
```

The output is: `System.Int32`.

The following example uses reflection to obtain the full name of the loaded assembly.

C#

```
// Using Reflection to get information of an Assembly:
Assembly info = typeof(int).Assembly;
Console.WriteLine(info);
```

The output is something like: `System.Private.CoreLib, Version=7.0.0.0, Culture=neutral, PublicKeyToken=7cec85d7bea7798e`.

> **Note**
>
> The C# keywords `protected` and `internal` have no meaning in Intermediate Language (IL) and are not used in the reflection APIs. The corresponding terms in IL are *Family* and *Assembly*. To identify an `internal` method using reflection, use the **IsAssembly** property. To identify a `protected internal` method, use the **IsFamilyOrAssembly**.

Using attributes

Attributes can be placed on almost any declaration, though a specific attribute might restrict the types of declarations on which it's valid. In C#, you specify an attribute by placing the name of the attribute enclosed in square brackets (`[]`) above the declaration of the entity to which it applies.

In this example, the SerializableAttribute attribute is used to apply a specific characteristic to a class:

C#

```
[Serializable]
public class SampleClass
{
    // Objects of this type can be serialized.
}
```

A method with the attribute DllImportAttribute is declared like the following example:

C#

```
[System.Runtime.InteropServices.DllImport("user32.dll")]
extern static void SampleMethod();
```

More than one attribute can be placed on a declaration as the following example shows:

C#

```
void MethodA([In][Out] ref double x) { }
void MethodB([Out][In] ref double x) { }
void MethodC([In, Out] ref double x) { }
```

Some attributes can be specified more than once for a given entity. An example of such a multiuse attribute is ConditionalAttribute:

C#

```
[Conditional("DEBUG"), Conditional("TEST1")]
void TraceMethod()
{
    // ...
}
```

> ⓘ Note
>
> By convention, all attribute names end with the word "Attribute" to distinguish them from other items in the .NET libraries. However, you do not need to specify the attribute suffix when using attributes in code. For example, `[DllImport]` is equivalent to `[DllImportAttribute]`, but `DllImportAttribute` is the attribute's actual name in the .NET Class Library.

Attribute parameters

Many attributes have parameters, which can be positional, unnamed, or named. Any positional parameters must be specified in a certain order and can't be omitted. Named parameters are optional and can be specified in any order. Positional parameters are specified first. For example, these three attributes are equivalent:

C#

```
[DllImport("user32.dll")]
[DllImport("user32.dll", SetLastError=false, ExactSpelling=false)]
[DllImport("user32.dll", ExactSpelling=false, SetLastError=false)]
```

The first parameter, the DLL name, is positional and always comes first; the others are named. In this case, both named parameters default to false, so they can be omitted. Positional parameters correspond to the parameters of the attribute constructor. Named or optional parameters correspond to either properties or fields of the attribute. Refer to the individual attribute's documentation for information on default parameter values.

For more information on allowed parameter types, see the Attributes section of the C# language specification

Attribute targets

The *target* of an attribute is the entity that the attribute applies to. For example, an attribute may apply to a class, a particular method, or an entire assembly. By default, an attribute applies to the element that follows it. But you can also explicitly identify, for example, whether an attribute is applied to a method, or to its parameter, or to its return value.

To explicitly identify an attribute target, use the following syntax:

```
C#

[target : attribute-list]
```

The list of possible `target` values is shown in the following table.

Target value	Applies to
`assembly`	Entire assembly
`module`	Current assembly module
`field`	Field in a class or a struct
`event`	Event
`method`	Method or `get` and `set` property accessors
`param`	Method parameters or `set` property accessor parameters
`property`	Property
`return`	Return value of a method, property indexer, or `get` property accessor
`type`	Struct, class, interface, enum, or delegate

You would specify the `field` target value to apply an attribute to the backing field created for an auto-implemented property.

The following example shows how to apply attributes to assemblies and modules. For more information, see Common Attributes (C#).

C#

```
using System;
using System.Reflection;
[assembly: AssemblyTitleAttribute("Production assembly 4")]
[module: CLSCompliant(true)]
```

The following example shows how to apply attributes to methods, method parameters, and method return values in C#.

C#

```
// default: applies to method
[ValidatedContract]
int Method1() { return 0; }

// applies to method
[method: ValidatedContract]
int Method2() { return 0; }

// applies to parameter
int Method3([ValidatedContract] string contract) { return 0; }

// applies to return value
[return: ValidatedContract]
int Method4() { return 0; }
```

> ⓘ **Note**
>
> Regardless of the targets on which `ValidatedContract` is defined to be valid, the `return` target has to be specified, even if `ValidatedContract` were defined to apply only to return values. In other words, the compiler will not use `AttributeUsage` information to resolve ambiguous attribute targets. For more information, see **AttributeUsage**.

Common uses for attributes

The following list includes a few of the common uses of attributes in code:

- Marking methods using the `WebMethod` attribute in Web services to indicate that the method should be callable over the SOAP protocol. For more information, see WebMethodAttribute.
- Describing how to marshal method parameters when interoperating with native code. For more information, see MarshalAsAttribute.
- Describing the COM properties for classes, methods, and interfaces.
- Calling unmanaged code using the DllImportAttribute class.
- Describing your assembly in terms of title, version, description, or trademark.
- Describing which members of a class to serialize for persistence.
- Describing how to map between class members and XML nodes for XML serialization.
- Describing the security requirements for methods.
- Specifying characteristics used to enforce security.
- Controlling optimizations by the just-in-time (JIT) compiler so the code remains easy to debug.
- Obtaining information about the caller to a method.

Reflection overview

Reflection is useful in the following situations:

- When you have to access attributes in your program's metadata. For more information, see Retrieving Information Stored in Attributes.
- For examining and instantiating types in an assembly.
- For building new types at run time. Use classes in System.Reflection.Emit.
- For performing late binding, accessing methods on types created at run time. See the article Dynamically Loading and Using Types.

Related sections

For more information:

- Common Attributes (C#)
- Caller Information (C#)
- Attributes
- Reflection
- Viewing Type Information
- Reflection and Generic Types
- System.Reflection.Emit
- Retrieving Information Stored in Attributes

Sérialisation dans .NET

Article • 02/08/2023

La sérialisation correspond au processus de conversion de l'état d'un objet en un formulaire persistant ou transportable. Le complément de la sérialisation est la désérialisation, qui convertit un flux de données en un objet. Ces deux processus permettent de stocker et de transférer facilement des données.

.NET propose les technologies de sérialisation suivantes :

- La sérialisation binaire préserve le respect des types, qui permet de conserver l'état d'un objet entre plusieurs appels d'une application. Par exemple, vous pouvez partager un objet entre plusieurs applications en le sérialisant dans le Presse-papiers. Vous pouvez sérialiser un objet vers un flux, un disque, la mémoire, le réseau, et ainsi de suite. La communication à distance utilise la sérialisation pour passer des objets « par valeur » d'un ordinateur ou d'un domaine d'application à un autre.

- La sérialisation XML et SOAP sérialise uniquement des propriétés et des champs publics, mais ne conserve pas le respect des types. Ceci est utile lorsque vous souhaitez fournir ou consommer des données sans restreindre l'application qui les utilise. XML étant une norme ouverte, elle constitue une option intéressante pour partager des données via le Web. Le protocole SOAP est également une norme ouverte et représente par conséquent une option avantageuse.

- La sérialisation JSON sérialise uniquement des propriétés publiques, mais ne conserve pas le respect des types. JSON est une norme ouverte qui constitue une option intéressante pour partager des données via le web.

Référence

System.Runtime.Serialization
Contient des classes qui peuvent être utilisées pour sérialiser et désérialiser des objets.

System.Xml.Serialization
Contient des classes qui peuvent être utilisées pour sérialiser des objets en documents ou en flux de données au format XML.

System.Text.Json
Contient des classes qui peuvent être utilisées pour sérialiser des objets en documents ou en flux de données au format JSON.

Comment : sérialiser un objet

Article • 07/04/2023

Pour sérialiser un objet, créez tout d'abord l'objet à sérialiser et définissez ses propriétés et champs publics. Pour ce faire, vous devez déterminer le format de transport dans lequel le flux de données XML sera stocké, sous forme de flux de données ou de fichier. Par exemple, si le flux de données XML doit être enregistré dans un formulaire permanent, créez un objet FileStream.

> **Notes**
>
> Pour obtenir plus d'exemples de sérialisation XML, consultez **Exemples de sérialisation XML**.

Pour sérialiser un objet

1. Créez l'objet et définissez ses champs et propriétés publics.
2. Construisez un XmlSerializer à l'aide du type de l'objet. Pour plus d'informations, consultez les constructeurs de classe XmlSerializer.
3. Appelez la méthode Serialize pour générer un flux de données XML ou une représentation de fichier des propriétés et des champs publics de l'objet. L'exemple suivant illustre la création d'un fichier.

```
C#
```

```
MySerializableClass myObject = new MySerializableClass();
// Insert code to set properties and fields of the object.
XmlSerializer mySerializer = new
XmlSerializer(typeof(MySerializableClass));
// To write to a file, create a StreamWriter object.
StreamWriter myWriter = new StreamWriter("myFileName.xml");
mySerializer.Serialize(myWriter, myObject);
myWriter.Close();
```

Voir aussi

- Introduction à la sérialisation XML
- Comment : désérialiser un objet

Comment sérialiser et désérialiser JSON dans .NET

Article • 01/06/2023

Cet article explique comment utiliser l'espace de noms System.Text.Json pour sérialiser et désérialiser à partir de JavaScript Object Notation (JSON). Si vous transférez du code existant à partir de `Newtonsoft.Json`, consultez Comment migrer vers System.Text.Json.

Exemples de code

Exemples de code de cet article :

- Utilisez directement la bibliothèque, et non une infrastructure comme ASP.NET Core.
- Utilisez la classe JsonSerializer avec des types personnalisés pour la sérialisation/désérialisation.

  Pour plus d'informations sur la lecture et l'écriture de données JSON sans utiliser `JsonSerializer`, consultez Comment utiliser le DOM JSON, Comment utiliser Utf8JsonReader et Comment utiliser Utf8JsonWriter.
- Utilisez l'option WriteIndented pour mettre en forme le JSON pour améliorer sa lisibilité lorsque cela est utile.

  Pour une utilisation en production, vous acceptez généralement la valeur par défaut de `false` pour ce paramètre, car l'ajout d'espaces blancs inutiles peut avoir un impact négatif sur les performances et l'utilisation de la bande passante.
- Reportez-vous à la classe suivante et ses variantes :

  C#

```
public class WeatherForecast
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string? Summary { get; set; }
}
```

Espaces de noms

L'espace de noms System.Text.Json contient tous les points d'entrée et les types principaux. L'espace de noms System.Text.Json.Serialization contient des attributs et des API pour les scénarios avancés, et la personnalisation spécifique à la sérialisation et à la désérialisation. Les exemples de code présentés dans cet article nécessitent des directives `using` pour l'un de ces espaces de noms ou les deux :

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;
```

> ⓘ **Important**
>
> - Les attributs de l'espace de noms **System.Runtime.Serialization** ne sont pas pris en charge par `System.Text.Json`.
> - **System.SerializableAttribute** et l'interface **ISerializable** ne sont pas pris en charge par `System.Text.Json`. Ces types sont utilisés uniquement pour la **Sérialisation binaire et XML**.

Comment écrire des objets .NET au format JSON (sérialisation)

Pour écrire du JSON dans une chaîne ou dans un fichier, appelez la méthode JsonSerializer.Serialize.

L'exemple suivant crée du JSON en tant que chaîne :

C#

```
using System.Text.Json;

namespace SerializeBasic
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    public class Program
    {
        public static void Main()
```

```
        {
            var weatherForecast = new WeatherForecast
            {
                Date = DateTime.Parse("2019-08-01"),
                TemperatureCelsius = 25,
                Summary = "Hot"
            };

            string jsonString = JsonSerializer.Serialize(weatherForecast);

            Console.WriteLine(jsonString);
        }
    }
}
// output:
//{"Date":"2019-08-01T00:00:00-
07:00","TemperatureCelsius":25,"Summary":"Hot"}
```

La sortie JSON est réduite au minimum (les espaces blancs, la mise en retrait et les caractères de nouvelle ligne sont supprimés) par défaut.

L'exemple suivant utilise du code synchrone pour créer un fichier JSON :

C#

```
using System.Text.Json;

namespace SerializeToFile
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    public class Program
    {
        public static void Main()
        {
            var weatherForecast = new WeatherForecast
            {
                Date = DateTime.Parse("2019-08-01"),
                TemperatureCelsius = 25,
                Summary = "Hot"
            };

            string fileName = "WeatherForecast.json";
            string jsonString = JsonSerializer.Serialize(weatherForecast);
            File.WriteAllText(fileName, jsonString);

            Console.WriteLine(File.ReadAllText(fileName));
        }
```

```
    }
}
// output:
//{"Date":"2019-08-01T00:00:00-
07:00","TemperatureCelsius":25,"Summary":"Hot"}
```

L'exemple suivant utilise du code asynchrone pour créer un fichier JSON :

C#

```
using System.Text.Json;

namespace SerializeToFileAsync
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    public class Program
    {
        public static async Task Main()
        {
            var weatherForecast = new WeatherForecast
            {
                Date = DateTime.Parse("2019-08-01"),
                TemperatureCelsius = 25,
                Summary = "Hot"
            };

            string fileName = "WeatherForecast.json";
            using FileStream createStream = File.Create(fileName);
            await JsonSerializer.SerializeAsync(createStream,
weatherForecast);
            await createStream.DisposeAsync();

            Console.WriteLine(File.ReadAllText(fileName));
        }
    }
}
// output:
//{"Date":"2019-08-01T00:00:00-
07:00","TemperatureCelsius":25,"Summary":"Hot"}
```

Les exemples précédents utilisent l'inférence de type pour le type sérialisé. Une surcharge de `Serialize()` prend un paramètre de type générique :

C#

```
using System.Text.Json;

namespace SerializeWithGenericParameter
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    public class Program
    {
        public static void Main()
        {
            var weatherForecast = new WeatherForecast
            {
                Date = DateTime.Parse("2019-08-01"),
                TemperatureCelsius = 25,
                Summary = "Hot"
            };

            string jsonString = JsonSerializer.Serialize<WeatherForecast>
(weatherForecast);

            Console.WriteLine(jsonString);
        }
    }
}
// output:
//{"Date":"2019-08-01T00:00:00-
07:00","TemperatureCelsius":25,"Summary":"Hot"}
```

Exemple de sérialisation

Voici un exemple montrant comment une classe qui contient des propriétés de collection et un type défini par l'utilisateur est sérialisée :

C#

```
using System.Text.Json;

namespace SerializeExtra
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
        public string? SummaryField;
        public IList<DateTimeOffset>? DatesAvailable { get; set; }
```

```
        public Dictionary<string, HighLowTemps>? TemperatureRanges { get;
set; }
        public string[]? SummaryWords { get; set; }
    }

    public class HighLowTemps
    {
        public int High { get; set; }
        public int Low { get; set; }
    }

    public class Program
    {
        public static void Main()
        {
            var weatherForecast = new WeatherForecast
            {
                Date = DateTime.Parse("2019-08-01"),
                TemperatureCelsius = 25,
                Summary = "Hot",
                SummaryField = "Hot",
                DatesAvailable = new List<DateTimeOffset>()
                    { DateTime.Parse("2019-08-01"), DateTime.Parse("2019-08-
02") },
                TemperatureRanges = new Dictionary<string, HighLowTemps>
                    {
                        ["Cold"] = new HighLowTemps { High = 20, Low = -10
},
                        ["Hot"] = new HighLowTemps { High = 60 , Low = 20 }
                    },
                SummaryWords = new[] { "Cool", "Windy", "Humid" }
            };

            var options = new JsonSerializerOptions { WriteIndented = true
};
            string jsonString = JsonSerializer.Serialize(weatherForecast,
options);

            Console.WriteLine(jsonString);
        }
    }
}
// output:
//{
//  "Date": "2019-08-01T00:00:00-07:00",
//  "TemperatureCelsius": 25,
//  "Summary": "Hot",
//  "DatesAvailable": [
//    "2019-08-01T00:00:00-07:00",
//    "2019-08-02T00:00:00-07:00"
//  ],
//  "TemperatureRanges": {
//    "Cold": {
//      "High": 20,
//      "Low": -10
```

```
//    },
//    "Hot": {
//    "High": 60,
//      "Low": 20
//    }
//  },
//  "SummaryWords": [
//    "Cool",
//    "Windy",
//    "Humid"
//  ]
//}
```

Sérialiser vers UTF-8

La sérialisation vers un tableau d'octets UTF-8 est environ 5 à 10 % plus rapide que l'utilisation des méthodes basées sur des chaînes. La différence est que les octets (UTF-8) n'ont pas besoin d'être convertis en chaînes (UTF-16).

Pour sérialiser dans un tableau d'octets UTF-8, appelez la méthode JsonSerializer.SerializeToUtf8Bytes :

C#

```
byte[] jsonUtf8Bytes =JsonSerializer.SerializeToUtf8Bytes(weatherForecast);
```

Une surcharge Serialize qui prend un Utf8JsonWriter est également disponible.

Comportements de sérialisation

- Par défaut, l'ensemble des propriétés publiques sont sérialisées. Vous pouvez spécifier des propriétés à ignorer.
- L'encodeur par défaut échappe les caractères non ASCII, les caractères HTML dans la plage ASCII et les caractères qui doivent être échappés selon la spécification JSON RFC 8259 .
- Par défaut, le JSON est minifié. Vous pouvez imprimer le JSON de façon élégante.
- Par défaut, la casse des noms JSON correspond aux noms .NET. Vous pouvez personnaliser la casse de nom JSON.
- Par défaut, les références circulaires sont détectées et des exceptions levées en conséquence. Vous pouvez conserver les références et gérer les références circulaires.
- Par défaut, les champs sont ignorés. Vous pouvez inclure des champs.

Lorsque vous utilisez System.Text.Json indirectement dans une application ASP.NET Core, certains comportements par défaut sont différents. Pour plus d'informations, consultez Valeurs web par défaut pour JsonSerializerOptions.

Les types pris en charge comprennent les suivants :

- Primitives .NET qui mappées sur des primitives JavaScript, comme les types numériques, les chaînes et les booléens.
- Objets CLR traditionnels (OCT) définis par l'utilisateur.
- Tableaux unidimensionnels et en escalier (`T[][]`).
- Collections et dictionnaires des espaces de noms suivants.
  - System.Collections
  - System.Collections.Generic
  - System.Collections.Immutable
  - System.Collections.Concurrent
  - System.Collections.Specialized
  - System.Collections.ObjectModel

Pour plus d'informations, consultez Types de collection pris en charge dans System.Text.Json.

Vous pouvez implémenter des convertisseurs personnalisés pour gérer des types supplémentaires ou pour fournir des fonctionnalités qui ne sont pas prises en charge par les convertisseurs intégrés.

Comment lire du JSON en tant qu'objets .NET (désérialiser)

Une façon courante de désérialiser du JSON consiste à créer d'abord une classe avec des propriétés et des champs qui représentent une ou plusieurs des propriétés JSON. Ensuite, pour désérialiser à partir d'une chaîne ou d'un fichier, appelez la méthode JsonSerializer.Deserialize. Pour les surcharges génériques, vous transmettez le type de la classe que vous avez créée comme paramètre de type générique. Pour les surcharges non génériques, vous transmettez le type de la classe que vous avez créée en tant que paramètre de méthode. Vous pouvez désérialiser de façon synchrone ou asynchrone.

Toutes les propriétés JSON qui ne sont pas représentées dans votre classe sont ignorées par défaut. En outre, si des propriétés sur le type sont requises, mais qu'elles ne sont pas présentes dans la charge utile JSON, la désérialisation échoue.

L'exemple suivant montre comment désérialiser une chaîne JSON :

C#

```
using System.Text.Json;

namespace DeserializeExtra
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
        public string? SummaryField;
        public IList<DateTimeOffset>? DatesAvailable { get; set; }
        public Dictionary<string, HighLowTemps>? TemperatureRanges { get;
set; }
        public string[]? SummaryWords { get; set; }
    }

    public class HighLowTemps
    {
        public int High { get; set; }
        public int Low { get; set; }
    }

    public class Program
    {
        public static void Main()
        {
            string jsonString =
@"{
  ""Date"": ""2019-08-01T00:00:00-07:00"",
  ""TemperatureCelsius"": 25,
  ""Summary"": ""Hot"",
  ""DatesAvailable"": [
    ""2019-08-01T00:00:00-07:00"",
    ""2019-08-02T00:00:00-07:00""
  ],
  ""TemperatureRanges"": {
                ""Cold"": {
                    ""High"": 20,
      ""Low"": -10
                },
    ""Hot"": {
                    ""High"": 60,
      ""Low"": 20
    }
            },
  ""SummaryWords"": [
    ""Cool"",
    ""Windy"",
    ""Humid""
  ]
}
";
```

```
            WeatherForecast? weatherForecast =
                JsonSerializer.Deserialize<WeatherForecast>(jsonString);

            Console.WriteLine($"Date: {weatherForecast?.Date}");
            Console.WriteLine($"TemperatureCelsius:
{weatherForecast?.TemperatureCelsius}");
            Console.WriteLine($"Summary: {weatherForecast?.Summary}");
        }
    }
}
// output:
//Date: 8/1/2019 12:00:00 AM -07:00
//TemperatureCelsius: 25
//Summary: Hot
```

Pour désérialiser à partir d'un fichier à l'aide de code synchrone, lisez le fichier dans une chaîne, comme illustré dans l'exemple suivant :

C#

```
using System.Text.Json;

namespace DeserializeFromFile
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    public class Program
    {
        public static void Main()
        {
            string fileName = "WeatherForecast.json";
            string jsonString = File.ReadAllText(fileName);
            WeatherForecast weatherForecast =
JsonSerializer.Deserialize<WeatherForecast>(jsonString)!;

            Console.WriteLine($"Date: {weatherForecast.Date}");
            Console.WriteLine($"TemperatureCelsius:
{weatherForecast.TemperatureCelsius}");
            Console.WriteLine($"Summary: {weatherForecast.Summary}");
        }
    }
}
// output:
//Date: 8/1/2019 12:00:00 AM -07:00
//TemperatureCelsius: 25
//Summary: Hot
```

Pour désérialiser à partir d'un fichier à l'aide de code asynchrone, appelez la méthode DeserializeAsync :

C#

```
using System.Text.Json;

namespace DeserializeFromFileAsync
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    public class Program
    {
        public static async Task Main()
        {
            string fileName = "WeatherForecast.json";
            using FileStream openStream = File.OpenRead(fileName);
            WeatherForecast? weatherForecast =
                await JsonSerializer.DeserializeAsync<WeatherForecast>
(openStream);
            Console.WriteLine($"Date: {weatherForecast?.Date}");
            Console.WriteLine($"TemperatureCelsius:
{weatherForecast?.TemperatureCelsius}");
            Console.WriteLine($"Summary: {weatherForecast?.Summary}");
        }
    }
}
// output:
//Date: 8/1/2019 12:00:00 AM -07:00
//TemperatureCelsius: 25
//Summary: Hot
```

Conseil

Si vous avez du JSON que vous souhaitez désérialiser et que vous n'avez pas la classe pour le désérialiser, vous avez d'autres options que la création manuelle de la classe dont vous avez besoin :

- Désérialisez dans un **DOM (modèle objet de document) JSON** et extrayez ce dont vous avez besoin du DOM.

  Le DOM vous permet d'accéder à une sous-section d'une charge utile JSON et de désérialiser une valeur unique, un type personnalisé ou un tableau. Pour

plus d'informations sur le DOM **JsonNode**, consultez **Désérialiser les sous-sections d'une charge utile JSON**. Pour plus d'informations sur le DOM **JsonDocument**, consultez **Comment rechercher des sous-éléments dans jsonDocument et JsonElement**.

- Utilisez **Utf8JsonReader** directement.
- Utilisez Visual Studio 2022 pour générer automatiquement la classe dont vous avez besoin :
  - Copiez le JSON que vous devez désérialiser.
  - Créez un fichier de classe et supprimez le code du modèle.
  - Choisissez **Modifier**>**Collage spécial**>**Coller le code JSON en tant que classes**. Le résultat est une classe que vous pouvez utiliser pour votre cible de désérialisation.

Désérialiser à partir d'UTF-8

Pour désérialiser à partir de UTF-8, appelez une surcharge de JsonSerializer.Deserialize qui prend un `ReadOnlySpan<byte>` ou un `Utf8JsonReader`, comme illustré dans les exemples suivants. Les exemples supposent que le JSON se trouve dans un tableau d'octets nommé jsonUtf8Bytes.

C#

```
var readOnlySpan = new ReadOnlySpan<byte>(jsonUtf8Bytes);
WeatherForecast deserializedWeatherForecast =
    JsonSerializer.Deserialize<WeatherForecast>(readOnlySpan)!;
```

C#

```
var utf8Reader = new Utf8JsonReader(jsonUtf8Bytes);
WeatherForecast deserializedWeatherForecast =
    JsonSerializer.Deserialize<WeatherForecast>(ref utf8Reader)!;
```

Comportement de désérialisation

Les comportements suivants s'appliquent lors de la désérialisation de JSON :

- Par défaut, la correspondance de nom de propriété respecte la casse. Vous pouvez spécifier l'insensibilité à la casse.

- Si le fichier JSON contient une valeur pour une propriété en lecture seule, la valeur est ignorée par défaut. Vous pouvez définir l'option PreferredObjectCreationHandling sur JsonObjectCreationHandling.Populate pour activer la désérialisation des propriétés en lecture seule.
- Les constructeurs non publics sont ignorés par le sérialiseur.
- La désérialisation vers des objets ou des propriétés immuables qui n'ont pas d'accesseurs publics `set` est prise en charge. Consultez Types et enregistrements immuables.
- Par défaut, les énumérations sont prises en charge en tant que nombres. Vous pouvez sérialiser des noms d'énumération sous forme de chaînes.
- Par défaut, les champs sont ignorés. Vous pouvez inclure des champs.
- Par défaut, les commentaires ou les virgules de fin dans le JSON lèvent des exceptions. Vous pouvez autoriser les commentaires et les virgules de fin.
- La profondeur maximale par défaut est de 64.

Lorsque vous utilisez System.Text.Json indirectement dans une application ASP.NET Core, certains comportements par défaut sont différents. Pour plus d'informations, consultez Valeurs web par défaut pour JsonSerializerOptions.

Vous pouvez implémenter des convertisseurs personnalisés pour fournir des fonctionnalités qui ne sont pas prises en charge par les convertisseurs intégrés.

Sérialiser en JSON mis en forme

Pour imprimer élégamment la sortie JSON, définissez JsonSerializerOptions.WriteIndented sur `true` :

C#

```
using System.Text.Json;

namespace SerializeWriteIndented
{
    public class WeatherForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string? Summary { get; set; }
    }

    public class Program
    {
        public static void Main()
        {
            var weatherForecast = new WeatherForecast
            {
```

```
                Date = DateTime.Parse("2019-08-01"),
                TemperatureCelsius = 25,
                Summary = "Hot"
            };

            var options = new JsonSerializerOptions { WriteIndented = true
};
            string jsonString = JsonSerializer.Serialize(weatherForecast,
options);

            Console.WriteLine(jsonString);
        }
    }
}
// output:
//{
//  "Date": "2019-08-01T00:00:00-07:00",
//  "TemperatureCelsius": 25,
//  "Summary": "Hot"
//}
```

Si vous utilisez `JsonSerializerOptions` plusieurs fois avec les mêmes options, ne créez pas d'instance de `JsonSerializerOptions` chaque fois que vous l'utilisez. Réutilisez la même instance pour chaque appel. Pour plus d'informations, consultez Réutiliser des instances de JsonSerializerOptions.

Inclure les champs

Utilisez le paramètre global JsonSerializerOptions.IncludeFields ou l'attribut [JsonInclude] pour inclure des champs lors de la sérialisation ou de la désérialisation, comme illustré dans l'exemple suivant :

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;

namespace Fields
{
    public class Forecast
    {
        public DateTime Date;
        public int TemperatureC;
        public string? Summary;
    }

    public class Forecast2
    {
        [JsonInclude]
```

```
        public DateTime Date;
        [JsonInclude]
        public int TemperatureC;
        [JsonInclude]
        public string? Summary;
    }

    public class Program
    {
        public static void Main()
        {
            var json =
                @"{""Date"":""2020-09-
06T11:31:01.923395"",""TemperatureC"":-1,""Summary"":""Cold""} ";
            Console.WriteLine($"Input JSON: {json}");

            var options = new JsonSerializerOptions
            {
                IncludeFields = true,
            };
            var forecast = JsonSerializer.Deserialize<Forecast>(json,
options)!;

            Console.WriteLine($"forecast.Date: {forecast.Date}");
            Console.WriteLine($"forecast.TemperatureC:
{forecast.TemperatureC}");
            Console.WriteLine($"forecast.Summary: {forecast.Summary}");

            var roundTrippedJson =
                JsonSerializer.Serialize<Forecast>(forecast, options);

            Console.WriteLine($"Output JSON: {roundTrippedJson}");

            var forecast2 = JsonSerializer.Deserialize<Forecast2>(json)!;

            Console.WriteLine($"forecast2.Date: {forecast2.Date}");
            Console.WriteLine($"forecast2.TemperatureC:
{forecast2.TemperatureC}");
            Console.WriteLine($"forecast2.Summary: {forecast2.Summary}");

            roundTrippedJson = JsonSerializer.Serialize<Forecast2>
(forecast2);

            Console.WriteLine($"Output JSON: {roundTrippedJson}");
        }
    }
}

// Produces output like the following example:
//
//Input JSON: { "Date":"2020-09-
06T11:31:01.923395","TemperatureC":-1,"Summary":"Cold"}
//forecast.Date: 9/6/2020 11:31:01 AM
//forecast.TemperatureC: -1
//forecast.Summary: Cold
```

```
//Output JSON: { "Date":"2020-09-
06T11:31:01.923395","TemperatureC":-1,"Summary":"Cold"}
//forecast2.Date: 9/6/2020 11:31:01 AM
//forecast2.TemperatureC: -1
//forecast2.Summary: Cold
//Output JSON: { "Date":"2020-09-
06T11:31:01.923395","TemperatureC":-1,"Summary":"Cold"}
```

Pour ignorer les champs en lecture seule, utilisez le paramètre global JsonSerializerOptions.IgnoreReadOnlyFields.

Méthodes d'extension HttpClient et HttpContent

La sérialisation et la désérialisation des charges utiles JSON à partir du réseau sont des opérations courantes. Les méthodes d'extension sur HttpClient et HttpContent vous permettent d'effectuer ces opérations en une seule ligne de code. Ces méthodes d'extension utilisent des valeurs web par défaut pour JsonSerializerOptions.

L'exemple suivant illustre l'utilisation de HttpClientJsonExtensions.GetFromJsonAsync et HttpClientJsonExtensions.PostAsJsonAsync :

C#

```
using System.Net.Http.Json;

namespace HttpClientExtensionMethods
{
    public class User
    {
        public int Id { get; set; }
        public string? Name { get; set; }
        public string? Username { get; set; }
        public string? Email { get; set; }
    }

    public class Program
    {
        public static async Task Main()
        {
            using HttpClient client = new()
            {
                BaseAddress = new
Uri("https://jsonplaceholder.typicode.com")
            };

            // Get the user information.
            User? user = await client.GetFromJsonAsync<User>("users/1");
```

```
            Console.WriteLine($"Id: {user?.Id}");
            Console.WriteLine($"Name: {user?.Name}");
            Console.WriteLine($"Username: {user?.Username}");
            Console.WriteLine($"Email: {user?.Email}");

            // Post a new user.
            HttpResponseMessage response = await
client.PostAsJsonAsync("users", user);
            Console.WriteLine(
                $"{(response.IsSuccessStatusCode ? "Success" : "Error")} -
{response.StatusCode}");
        }
    }
}

// Produces output like the following example but with different names:
//
//Id: 1
//Name: Tyler King
//Username: Tyler
//Email: Tyler @contoso.com
//Success - Created
```

Il existe également des méthodes d'extension pour System.Text.Json sur HttpContent.

Voir aussi

- Vue d'ensemble de System.Text.Json

Instructions (Guide de programmation C#)

Article • 07/04/2023

Les actions qu'un programme effectue sont exprimées dans les instructions. Les actions courantes incluent la déclaration de variables, l'assignation de valeurs, l'appel de méthodes, l'exécution de boucles dans les collections et la création de branches sur des blocs de code, selon une condition donnée. L'ordre dans lequel les instructions sont exécutées dans un programme est appelé le flux de contrôle ou le flux d'exécution. Le flux de contrôle peut varier chaque fois qu'un programme est exécuté, selon la manière dont le programme réagit à l'entrée qu'il reçoit au moment de l'exécution.

Une instruction peut comporter une seule ligne de code terminée par un point-virgule ou une série d'instructions d'une ligne dans un bloc. Un bloc d'instructions est placé entre des accolades ({}) et peut contenir des blocs imbriqués. Le code suivant illustre deux exemples d'instructions d'une ligne, et un bloc d'instructions multiligne :

C#

```
    static void Main()
    {
        // Declaration statement.
        int counter;

        // Assignment statement.
        counter = 1;

        // Error! This is an expression, not an expression statement.
        // counter + 1;

        // Declaration statements with initializers are functionally
        // equivalent to  declaration statement followed by assignment
statement:
        int[] radii = { 15, 32, 108, 74, 9 }; // Declare and initialize an
array.
        const double pi = 3.14159; // Declare and initialize  constant.

        // foreach statement block that contains multiple statements.
        foreach (int radius in radii)
        {
            // Declaration statement with initializer.
            double circumference = pi * (2 * radius);

            // Expression statement (method invocation). A single-line
            // statement can span multiple text lines because line breaks
            // are treated as white space, which is ignored by the compiler.
            System.Console.WriteLine("Radius of circle #{0} is {1}.
```

```
Circumference = {2:N2}",
                                    counter, radius, circumference);

            // Expression statement (postfix increment).
            counter++;
        } // End of foreach statement block
    } // End of Main method body.
} // End of SimpleStatements class.
/*
   Output:
    Radius of circle #1 = 15. Circumference = 94.25
    Radius of circle #2 = 32. Circumference = 201.06
    Radius of circle #3 = 108. Circumference = 678.58
    Radius of circle #4 = 74. Circumference = 464.96
    Radius of circle #5 = 9. Circumference = 56.55
*/
```

Types d'instructions

Le tableau suivant répertorie les différents types d'instructions en C# et leurs mots clés associés, avec des liens vers des rubriques contenant plus d'informations :

Category	Mots clés C# / Remarques
Instructions de déclaration	Une instruction de déclaration introduit une nouvelle variable ou constante. Une déclaration de variable peut éventuellement assigner une valeur à la variable. Dans une déclaration de constante, l'assignation est obligatoire.
Instructions d'expression	Les instructions d'expression qui calculent une valeur doivent stocker la valeur dans une variable.
Instructions de sélection	Les instructions de sélection permettent de créer des branches vers différentes sections de code, selon une ou plusieurs conditions spécifiées. Pour plus d'informations, voir les rubriques suivantes : • if • switch
Instructions d'itération	Les instructions d'itération permettent d'exécuter une boucle dans des collections telles que des tableaux, ou d'effectuer à plusieurs reprises le même jeu d'instructions jusqu'à ce qu'une condition spécifiée soit remplie. Pour plus d'informations, voir les rubriques suivantes : • do • for • foreach • while

Category	Mots clés C# / Remarques
Instructions de saut	Les instructions de saut transfèrent le contrôle vers une autre section de code. Pour plus d'informations, voir les rubriques suivantes : • break • continue • goto • return • yield
Instructions de gestion des exceptions	Les instructions de gestion des exceptions permettent de récupérer normalement en cas de conditions exceptionnelles au moment de l'exécution. Pour plus d'informations, voir les rubriques suivantes : • throw • try-catch • try-finally • try-catch-finally
checked et unchecked	Les instructions `checked` et `unchecked` vous permettent de spécifier si les opérations numériques de type entier sont autorisées à provoquer un dépassement de capacité lorsque le résultat est stocké dans une variable trop petite pour contenir la valeur résultante.
Instruction `await`	Si vous marquez une méthode avec le modificateur async, vous pouvez utiliser l'opérateur await dans la méthode. Quand le contrôle atteint une expression `await` dans la méthode async, il retourne à l'appelant, et la progression dans la méthode est interrompue jusqu'à ce que la tâche attendue soit terminée. Quand la tâche est terminée, l'exécution peut reprendre dans la méthode. Pour obtenir un exemple simple, consultez la section « Méthodes async » de Méthodes. Pour plus d'informations, consultez Programmation asynchrone avec async et await.
Instruction `yield return`	Un itérateur exécute une itération personnalisée sur une collection, comme une liste ou un tableau. Un itérateur utilise l'instruction yield return pour retourner chaque élément un par un. Quand une instruction `yield return` est atteinte, l'emplacement actuel dans le code est mémorisé. L'exécution redémarre à partir de cet emplacement au prochain appel de l'itérateur. Pour plus d'informations, consultez Itérateurs.
Instruction `fixed`	L'instruction fixed empêche le récupérateur de mémoire de déplacer une variable mobile. Pour plus d'informations, consultez fixed.
Instruction `lock`	L'instruction lock permet de limiter l'accès aux blocs de code à un seul thread à la fois. Pour plus d'informations, consultez lock.

Category	Mots clés C# / Remarques
Instructions étiquetées	Vous pouvez donner une étiquette à une instruction, puis utiliser le mot clé goto pour sauter jusqu'à l'instruction étiquetée. (Examinez l'exemple de la ligne suivante.)
Instruction vide	L'instruction vide se compose seulement d'un point-virgule. Elle ne fait rien et peut être utilisée à un emplacement où une instruction est requise alors qu'aucune action ne doit être effectuée.

Instructions de déclaration

Le code suivant montre des exemples de déclarations de variables avec et sans assignation initiale, ainsi qu'une déclaration de constante avec l'initialisation nécessaire.

C#

```
// Variable declaration statements.
double area;
double radius = 2;

// Constant declaration statement.
const double pi = 3.14159;
```

Instructions d'expression

Le code suivant montre des exemples d'instructions d'expression, y compris l'assignation, la création d'objets avec assignation et l'appel de méthode.

C#

```
// Expression statement (assignment).
area = 3.14 * (radius * radius);

// Error. Not  statement because no assignment:
//circ * 2;

// Expression statement (method invocation).
System.Console.WriteLine();

// Expression statement (new object creation).
System.Collections.Generic.List<string> strings =
    new System.Collections.Generic.List<string>();
```

Instruction vide

Les exemples suivants illustrent deux utilisations d'une instruction vide :

C#

```
void ProcessMessages()
{
    while (ProcessMessage())
        ; // Statement needed here.
}

void F()
{
    //...
    if (done) goto exit;
//...
exit:
    ; // Statement needed here.
}
```

Instructions incorporées

Certaines instructions, par exemple les instructions d'itération, ont toujours une instruction incorporée qui les suit. Cette instruction incorporée peut être une seule instruction ou plusieurs instructions placées entre des accolades ({}) dans un bloc d'instructions. Même les instructions incorporées d'une seule ligne peuvent être placées entre des accolades ({}), comme illustré dans l'exemple suivant :

C#

```
// Recommended style. Embedded statement in  block.
foreach (string s in System.IO.Directory.GetDirectories(
                        System.Environment.CurrentDirectory))
{
    System.Console.WriteLine(s);
}

// Not recommended.
foreach (string s in System.IO.Directory.GetDirectories(
                        System.Environment.CurrentDirectory))
    System.Console.WriteLine(s);
```

Une instruction incorporée qui n'est pas placée entre des accolades ({}) ne peut pas être une instruction de déclaration ni une instruction étiquetée. Ceci est illustré dans l'exemple suivant :

C#

```
if(pointB == true)
    //Error CS1023:
    int radius = 5;
```

Placez l'instruction incorporée dans un bloc pour corriger l'erreur :

C#

```
if (b == true)
{
    // OK:
    System.DateTime d = System.DateTime.Now;
    System.Console.WriteLine(d.ToLongDateString());
}
```

Blocs d'instructions imbriqués

Les blocs d'instructions peuvent être imbriqués, comme illustré dans le code suivant :

C#

```
foreach (string s in System.IO.Directory.GetDirectories(
    System.Environment.CurrentDirectory))
{
    if (s.StartsWith("CSharp"))
    {
        if (s.EndsWith("TempFolder"))
        {
            return s;
        }
    }
}
return "Not found.";
```

Instructions inaccessibles

Si le compilateur détermine que le flux de contrôle ne peut jamais atteindre une instruction particulière, il génère l'avertissement CS0162, comme illustré dans l'exemple suivant :

C#

```
// An over-simplified example of unreachable code.
const int val = 5;
```

```
if (val < 4)
{
    System.Console.WriteLine("I'll never write anything."); //CS0162
}
```

spécification du langage C#

Pour plus d'informations, consultez la section Instructions de la Spécification du langage C#.

Voir aussi

- Guide de programmation C#
- Mots clés des instructions
- Opérateurs et expressions C#

Membres expression-bodied (Guide de programmation C#)

Article • 10/05/2023

Les définitions de corps d'expression vous permettent de fournir l'implémentation d'un membre sous une forme lisible et très concise. Vous pouvez utiliser une définition de corps d'expression chaque fois que la logique d'un membre pris en charge, comme une méthode ou une propriété, se compose d'une seule expression. La syntaxe générale d'une définition de corps d'expression est la suivante :

```
C#

member => expression;
```

où *expression* est une expression valide.

Les définitions de corps d'expression peuvent être utilisées avec les membres de type suivants :

- Méthode
- Propriété en lecture seule
- Propriété
- Constructeur
- Finaliseur
- Indexeur

Méthodes

Une méthode expression-bodied se compose d'une seule expression qui retourne une valeur dont le type correspond au type de retour de la méthode ou, pour les méthodes qui retournent `void`, qui effectue une opération. Par exemple, les types qui substituent la méthode ToString incluent généralement une expression unique qui retourne la représentation sous forme de chaîne de l'objet actuel.

L'exemple suivant définit une classe `Person` qui substitue la méthode ToString avec une définition de corps d'expression. Il définit également une méthode `DisplayName` qui affiche un nom sur la console. Notez que le mot clé `return` n'est pas utilisé dans la définition de corps d'expression `ToString`.

```
C#
```

```
using System;

namespace ExpressionBodiedMembers;

public class Person
{
   public Person(string firstName, string lastName)
   {
      fname = firstName;
      lname = lastName;
   }

   private string fname;
   private string lname;

   public override string ToString() => $"{fname} {lname}".Trim();
   public void DisplayName() => Console.WriteLine(ToString());
}

class Example
{
   static void Main()
   {
      Person p = new Person("Mandy", "Dejesus");
      Console.WriteLine(p);
      p.DisplayName();
   }
}
```

Pour plus d'informations, consultez Méthodes (Guide de programmation C#).

Propriétés en lecture seule

Vous pouvez utiliser une définition de corps d'expression pour implémenter une propriété en lecture seule. Pour ce faire, utilisez la syntaxe suivante :

C#

```
PropertyType PropertyName => expression;
```

L'exemple suivant définit une classe `Location` dont la propriété en lecture seule `Name` est implémentée comme une définition de corps d'expression qui retourne la valeur du champ privé `locationName` :

C#

```
public class Location
{
```

```
    private string locationName;

    public Location(string name)
    {
        locationName = name;
    }

    public string Name => locationName;
}
```

Pour plus d'informations sur les propriétés, consultez Propriétés (Guide de programmation C#).

Propriétés

Vous pouvez utiliser des définitions de corps d'expression pour implémenter la propriété `get` et les accesseurs `set`. L'exemple suivant montre comment faire :

C#

```
public class Location
{
    private string locationName;

    public Location(string name) => Name = name;

    public string Name
    {
        get => locationName;
        set => locationName = value;
    }
}
```

Pour plus d'informations sur les propriétés, consultez Propriétés (Guide de programmation C#).

Constructeurs

Une définition de corps d'expression pour un constructeur se compose généralement d'une seule expression d'assignation ou d'un appel de méthode qui gère les arguments du constructeur ou qui initialise l'état de l'instance.

L'exemple suivant définit une classe `Location` dont le constructeur a un seul paramètre de chaîne nommé *name*. La définition de corps d'expression assigne l'argument à la propriété `Name`.

C#

```
public class Location
{
   private string locationName;

   public Location(string name) => Name = name;

   public string Name
   {
      get => locationName;
      set => locationName = value;
   }
}
```

Pour plus d'informations, consultez Constructeurs (Guide de programmation C#).

Finaliseurs

Une définition de corps d'expression pour un finaliseur contient généralement des instructions de nettoyage, telles que des instructions qui libèrent les ressources non managées.

L'exemple suivant définit un finaliseur qui utilise une définition de corps d'expression pour indiquer que le finaliseur a été appelé.

C#

```
public class Destroyer
{
   public override string ToString() => GetType().Name;

   ~Destroyer() => Console.WriteLine($"The {ToString()} finalizer is
executing.");
}
```

Pour plus d'informations, consultez Finaliseurs (Guide de programmation C#).

Indexeurs

Comme avec les propriétés, l'indexeur `get` et les accesseurs `set` se composent de définitions de corps d'expression si l'accesseur `get` se compose d'une seule expression qui retourne une valeur ou que l'accesseur `set` effectue une affectation simple.

L'exemple suivant définit une classe nommée `Sports` qui inclut un tableau String interne contenant les noms de plusieurs sports. L'indexeur `get` et les accesseurs `set` sont implémentés en tant que définitions de corps d'expression.

C#

```
using System;
using System.Collections.Generic;

namespace SportsExample;

public class Sports
{
   private string[] types = { "Baseball", "Basketball", "Football",
                              "Hockey", "Soccer", "Tennis",
                              "Volleyball" };

   public string this[int i]
   {
      get => types[i];
      set => types[i] = value;
   }
}
```

Pour plus d'informations, consultez Indexeurs (Guide de programmation C#).

Voir aussi

- Règles de style de code .NET pour les membres expression-bodied

Comparaisons d'égalité (Guide de programmation C#)

Article • 10/05/2023

Il est parfois nécessaire de comparer l'égalité de deux valeurs. Dans certains cas, vous testez *l'égalité de valeur*, également appelée *équivalence*, qui signifie que les valeurs contenues dans les deux variables sont égales. Dans d'autres cas, vous devez déterminer si deux variables référencent le même objet sous-jacent en mémoire. Ce type d'égalité est appelée *l'égalité de référence* ou *identité*. Cette rubrique décrit ces deux genres d'égalité et fournit des liens vers d'autres rubriques pour obtenir plus d'informations.

Égalité des références

L'égalité de références signifie que deux références d'objets référencent le même objet sous-jacent. Cela peut se produire par simple assignation, comme illustré dans l'exemple suivant.

C#

```
using System;
class Test
{
    public int Num { get; set; }
    public string Str { get; set; }

    static void Main()
    {
        Test a = new Test() { Num = 1, Str = "Hi" };
        Test b = new Test() { Num = 1, Str = "Hi" };

        bool areEqual = System.Object.ReferenceEquals(a, b);
        // False:
        System.Console.WriteLine("ReferenceEquals(a, b) = {0}", areEqual);

        // Assign b to a.
        b = a;

        // Repeat calls with different results.
        areEqual = System.Object.ReferenceEquals(a, b);
        // True:
        System.Console.WriteLine("ReferenceEquals(a, b) = {0}", areEqual);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
```

```
    }
}
```

Dans ce code, deux objets sont créés, mais après l'instruction d'assignation, les deux références référencent le même objet. Elles présentent donc une égalité de référence. Utilisez la méthode ReferenceEquals pour déterminer si deux références référencent le même objet.

Le concept d'égalité de référence s'applique uniquement aux types référence. Les objets de type valeur ne peuvent pas avoir une égalité de référence, car quand une instance d'un type valeur est assignée à une variable, une copie de la valeur est effectuée. Ainsi, vous ne pouvez jamais avoir deux structs unboxed qui référencent le même emplacement en mémoire. De plus, si vous utilisez ReferenceEquals pour comparer deux types valeur, le résultat est toujours `false`, même si les valeurs contenues dans les objets sont toutes identiques. Cela est dû au fait que chaque variable est convertie en une instance d'objet distincte. Pour plus d'informations, consultez Guide pratique pour tester l'égalité des références (Identité).

Égalité des valeurs

L'égalité de valeur signifie que deux objets contiennent les mêmes valeurs. Pour les types valeur primitifs tels que int ou bool, les tests d'égalité de valeur sont simples. Vous pouvez utiliser l'opérateur ==, comme indiqué dans l'exemple suivant.

C#

```
int a = GetOriginalValue();
int b = GetCurrentValue();

// Test for value equality.
if (b == a)
{
    // The two integers are equal.
}
```

Pour la plupart des autres types, les tests d'égalité de valeur sont plus complexes parce qu'ils vous demandent de comprendre comment le type la définit. Pour les classes et structs qui ont plusieurs champs ou propriétés, l'égalité de valeur signifie souvent que tous les champs ou propriétés ont la même valeur. Par exemple, deux objets `Point` peuvent être définis comme équivalents si pointA.X est égal à pointB.X et que pointA.Y est égal à pointB.Y. Pour les enregistrements, l'égalité des valeurs signifie que deux variables d'un type d'enregistrement sont égales si les types correspondent et que toutes les valeurs de propriété et de champ correspondent.

Toutefois, il n'est pas obligatoire que l'équivalence soit basée sur tous les champs dans un type. Elle peut être basée sur une partie. Quand vous comparez des types dont vous n'êtes pas propriétaire, vous devez bien comprendre comment l'équivalence est définie pour ce type. Pour plus d'informations sur la façon de définir l'égalité des valeurs dans vos propres classes et structs, consultez Guide pratique pour définir une égalité de valeurs pour un type.

Égalité des valeurs à virgule flottante

Les comparaisons d'égalité des valeurs à virgule flottante (double et float) sont problématiques en raison de l'imprécision de l'arithmétique à virgule flottante sur les ordinateurs binaires. Pour plus d'informations, consultez les notes dans la rubrique System.Double.

Rubriques connexes

Intitulé	Description
Guide pratique pour tester l'égalité des références (Identité)	Décrit comment déterminer si deux variables présentent une égalité de référence.
Comment définir une égalité de valeurs pour un type	Décrit comment fournir une définition personnalisée de l'égalité de valeur pour un type.
Guide de programmation C#	Fournit des liens vers des informations détaillées sur les fonctionnalités importantes du langage C# et sur les fonctionnalités disponibles pour C# via .NET.
Types	Fournit des informations sur le système de typeC# et des liens vers des informations supplémentaires.
Enregistrements	Fournit des informations sur les types d'enregistrements, qui testent par défaut l'égalité des valeurs.

Voir aussi

- Guide de programmation C#

Guide pratique pour définir l'égalité des valeurs pour une classe ou un struct (Guide de programmation C#)

Article • 07/04/2023

Les enregistrements implémentent automatiquement l'égalité des valeurs. Envisagez de définir un `record` au lieu d'un `class` lorsque votre type modélise des données doit implémenter l'égalité des valeurs.

Quand vous définissez une classe ou un struct, vous décidez s'il est judicieux de créer une définition personnalisée de l'égalité des valeurs (ou équivalence) pour le type. En règle générale, vous implémentez l'égalité des valeurs lorsque vous prévoyez d'ajouter des objets de ce type à une collection, ou lorsque leur objectif principal est de stocker un ensemble de champs ou de propriétés. Vous pouvez baser votre définition de l'égalité des valeurs sur une comparaison de tous les champs et propriétés du type, ou vous pouvez la baser sur un sous-ensemble.

Dans les deux cas, et tant dans les classes que dans les structs, votre implémentation doit suivre les cinq garanties d'équivalence (pour les règles suivantes, supposons que `x`, `y` et `z` ne sont pas null) :

1. La propriété réflexive : `x.Equals(x)` renvoie `true`.
2. La propriété symétrique : `x.Equals(y)` renvoie la même valeur que `y.Equals(x)`.
3. La propriété transitive : si `(x.Equals(y) && y.Equals(z))` renvoie `true`, alors `x.Equals(z)` renvoie `true`.
4. Les invocations successives de `x.Equals(y)` renvoient la même valeur tant que les objets référencés par x et y ne sont pas modifiés.
5. Toute valeur non null n'est pas égale à null. Toutefois, `x.Equals(y)` lève une exception lorsque `x` a la valeur null. Cela enfreint les règles 1 ou 2, selon l'argument de `Equals`.

Tout struct que vous définissez a déjà une implémentation par défaut de l'égalité des valeurs dont il hérite de la substitution System.ValueType de la méthode Object.Equals(Object). Cette implémentation utilise la réflexion pour examiner tous les champs et propriétés du type. Bien que cette implémentation produise des résultats

corrects, elle est relativement lente par rapport à une implémentation personnalisée que vous écrivez spécifiquement pour le type.

Les détails d'implémentation pour l'égalité des valeurs sont différents pour les classes et les structs. Toutefois, les classes et les structs nécessitent tous deux les mêmes étapes de base pour l'implémentation de l'égalité :

1. Substituez la méthode virtualObject.Equals(Object). Dans la plupart des cas, votre implémentation de `bool Equals( object obj )` doit simplement appeler la méthode `Equals` propre au type qui est l'implémentation de l'interface System.IEquatable<T>. (Voir l'étape 2.)

2. Implémentez l'interface System.IEquatable<T> en fournissant une méthode `Equals` propre au type. C'est ici qu'est effectuée la comparaison d'équivalence. Par exemple, vous pouvez décider de définir l'égalité en comparant seulement un ou deux champs de votre type. Ne levez pas d'exceptions à partir de `Equals`. Pour les classes associées par héritage :

   - cette méthode doit examiner uniquement les champs qui sont déclarés dans la classe. Elle doit appeler `base.Equals` pour examiner les champs qui sont dans la classe de base. (N'appelez pas `base.Equals` si le type hérite directement de Object, car Object l'implémentation de Object.Equals(Object) effectue une vérification de l'égalité des références)

   - Deux variables doivent être considérées comme égales uniquement si les types d'exécution des variables comparées sont identiques. Vérifiez également que `IEquatable`l'implémentation de la `Equals` méthode pour le type d'exécution est utilisée si les types d'exécution et de compilation d'une variable sont différents. Une stratégie pour s'assurer que les types d'exécution sont toujours comparés correctement consiste à implémenter `IEquatable` uniquement dans `sealed`les classes. Pour plus d'informations, consultez l'exemple de classe plus loin dans cet article.

3. Facultatif, mais recommandé : Surchargez les opérateurs == et ! =.

4. Substituez Object.GetHashCode pour que deux objets ayant une égalité des valeurs produisent le même code de hachage.

5. Facultatif : Pour prendre en charge les définitions de « supérieur à » ou « inférieur à », implémentez l'interface IComparable<T> de votre type, et surchargez également les opérateurs <= et >=.

> ⓘ **Notes**

À compter de C# 9.0, vous pouvez utiliser des enregistrements pour obtenir la sémantique d'égalité des valeurs sans code standard inutile.

Exemple de classe

L'exemple suivant montre comment implémenter l'égalité des valeurs dans une classe (type référence).

C#

```
namespace ValueEqualityClass;

class TwoDPoint : IEquatable<TwoDPoint>
{
    public int X { get; private set; }
    public int Y { get; private set; }

    public TwoDPoint(int x, int y)
    {
        if (x is (< 1 or > 2000) || y is (< 1 or > 2000))
        {
            throw new ArgumentException("Point must be in range 1 - 2000");
        }
        this.X = x;
        this.Y = y;
    }

    public override bool Equals(object obj) => this.Equals(obj as
TwoDPoint);

    public bool Equals(TwoDPoint p)
    {
        if (p is null)
        {
            return false;
        }

        // Optimization for a common success case.
        if (Object.ReferenceEquals(this, p))
        {
            return true;
        }

        // If run-time types are not exactly the same, return false.
        if (this.GetType() != p.GetType())
        {
            return false;
        }

        // Return true if the fields match.
        // Note that the base class is not invoked because it is
```

```
        // System.Object, which defines Equals as reference equality.
        return (X == p.X) && (Y == p.Y);
    }

    public override int GetHashCode() => (X, Y).GetHashCode();

    public static bool operator ==(TwoDPoint lhs, TwoDPoint rhs)
    {
        if (lhs is null)
        {
            if (rhs is null)
            {
                return true;
            }

            // Only the left side is null.
            return false;
        }
        // Equals handles case of null on right side.
        return lhs.Equals(rhs);
    }

    public static bool operator !=(TwoDPoint lhs, TwoDPoint rhs) => !(lhs ==
rhs);
}

// For the sake of simplicity, assume a ThreeDPoint IS a TwoDPoint.
class ThreeDPoint : TwoDPoint, IEquatable<ThreeDPoint>
{
    public int Z { get; private set; }

    public ThreeDPoint(int x, int y, int z)
        : base(x, y)
    {
        if ((z < 1) || (z > 2000))
        {
            throw new ArgumentException("Point must be in range 1 - 2000");
        }
        this.Z = z;
    }

    public override bool Equals(object obj) => this.Equals(obj as
ThreeDPoint);

    public bool Equals(ThreeDPoint p)
    {
        if (p is null)
        {
            return false;
        }

        // Optimization for a common success case.
        if (Object.ReferenceEquals(this, p))
        {
            return true;
```

```
        }

        // Check properties that this class declares.
        if (Z == p.Z)
        {
            // Let base class check its own fields
            // and do the run-time type comparison.
            return base.Equals((TwoDPoint)p);
        }
        else
        {
            return false;
        }
    }

    public override int GetHashCode() => (X, Y, Z).GetHashCode();

    public static bool operator ==(ThreeDPoint lhs, ThreeDPoint rhs)
    {
        if (lhs is null)
        {
            if (rhs is null)
            {
                // null == null = true.
                return true;
            }

            // Only the left side is null.
            return false;
        }
        // Equals handles the case of null on right side.
        return lhs.Equals(rhs);
    }

    public static bool operator !=(ThreeDPoint lhs, ThreeDPoint rhs) => !
(lhs == rhs);
}

class Program
{
    static void Main(string[] args)
    {
        ThreeDPoint pointA = new ThreeDPoint(3, 4, 5);
        ThreeDPoint pointB = new ThreeDPoint(3, 4, 5);
        ThreeDPoint pointC = null;
        int i = 5;

        Console.WriteLine("pointA.Equals(pointB) = {0}",
pointA.Equals(pointB));
        Console.WriteLine("pointA == pointB = {0}", pointA == pointB);
        Console.WriteLine("null comparison = {0}", pointA.Equals(pointC));
        Console.WriteLine("Compare to some other type = {0}",
pointA.Equals(i));

        TwoDPoint pointD = null;
```

```
        TwoDPoint pointE = null;

        Console.WriteLine("Two null TwoDPoints are equal: {0}", pointD ==
pointE);

        pointE = new TwoDPoint(3, 4);
        Console.WriteLine("(pointE == pointA) = {0}", pointE == pointA);
        Console.WriteLine("(pointA == pointE) = {0}", pointA == pointE);
        Console.WriteLine("(pointA != pointE) = {0}", pointA != pointE);

        System.Collections.ArrayList list = new
System.Collections.ArrayList();
        list.Add(new ThreeDPoint(3, 4, 5));
        Console.WriteLine("pointE.Equals(list[0]): {0}",
pointE.Equals(list[0]));

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
    pointA.Equals(pointB) = True
    pointA == pointB = True
    null comparison = False
    Compare to some other type = False
    Two null TwoDPoints are equal: True
    (pointE == pointA) = False
    (pointA == pointE) = False
    (pointA != pointE) = True
    pointE.Equals(list[0]): False
*/
```

Sur les classes (types référence), l'implémentation par défaut des deux méthodes Object.Equals(Object) effectue une comparaison d'égalité de référence, et non une vérification de l'égalité des valeurs. Quand un implémenteur substitue la méthode virtuelle, l'objectif est de lui donner une sémantique d'égalité des valeurs.

Les opérateurs `==` et `!=` peuvent être utilisés avec des classes, même si la classe ne les surcharge pas. Toutefois, le comportement par défaut consiste à effectuer une vérification de l'égalité de référence. Dans une classe, si vous surchargez la méthode `Equals`, vous devez surcharger les opérateurs `==` et `!=`, mais cela n'est pas obligatoire.

> **ⓘ Important**
>
> L'exemple de code précédent peut ne pas traiter tous les scénarios d'héritage comme vous l'attendez. Prenez le code suivant :

C#

```
TwoDPoint p1 = new ThreeDPoint(1, 2, 3);
TwoDPoint p2 = new ThreeDPoint(1, 2, 4);
Console.WriteLine(p1.Equals(p2)); // output: True
```

Ce code signale que `p1` est égal à `p2` malgré la différence de valeurs `z`. La différence est ignorée, car le compilateur choisit l'implémentation `TwoDPoint` de `IEquatable` en fonction du type de compilation.

L'égalité de valeur intégrée des `record` types gère correctement les scénarios de ce type. Si `TwoDPoint` et `ThreeDPoint` étaient des types `record`, le résultat de `p1.Equals(p2)` serait `False`. Pour plus d'informations, consultez **Égalité dans record les hiérarchies d'héritage de type** .

Exemple de struct

L'exemple suivant montre comment implémenter l'égalité des valeurs dans un struct (type valeur) :

C#

```
namespace ValueEqualityStruct
{
    struct TwoDPoint : IEquatable<TwoDPoint>
    {
        public int X { get; private set; }
        public int Y { get; private set; }

        public TwoDPoint(int x, int y)
            : this()
        {
            if (x is (< 1 or > 2000) || y is (< 1 or > 2000))
            {
                throw new ArgumentException("Point must be in range 1 -
2000");
            }
            X = x;
            Y = y;
        }

        public override bool Equals(object? obj) => obj is TwoDPoint other
&& this.Equals(other);

        public bool Equals(TwoDPoint p) => X == p.X && Y == p.Y;

        public override int GetHashCode() => (X, Y).GetHashCode();
```

```
        public static bool operator ==(TwoDPoint lhs, TwoDPoint rhs) =>
lhs.Equals(rhs);

        public static bool operator !=(TwoDPoint lhs, TwoDPoint rhs) => !
(lhs == rhs);
    }

    class Program
    {
        static void Main(string[] args)
        {
            TwoDPoint pointA = new TwoDPoint(3, 4);
            TwoDPoint pointB = new TwoDPoint(3, 4);
            int i = 5;

            // True:
            Console.WriteLine("pointA.Equals(pointB) = {0}",
pointA.Equals(pointB));
            // True:
            Console.WriteLine("pointA == pointB = {0}", pointA == pointB);
            // True:
            Console.WriteLine("object.Equals(pointA, pointB) = {0}",
object.Equals(pointA, pointB));
            // False:
            Console.WriteLine("pointA.Equals(null) = {0}",
pointA.Equals(null));
            // False:
            Console.WriteLine("(pointA == null) = {0}", pointA == null);
            // True:
            Console.WriteLine("(pointA != null) = {0}", pointA != null);
            // False:
            Console.WriteLine("pointA.Equals(i) = {0}", pointA.Equals(i));
            // CS0019:
            // Console.WriteLine("pointA == i = {0}", pointA == i);

            // Compare unboxed to boxed.
            System.Collections.ArrayList list = new
System.Collections.ArrayList();
            list.Add(new TwoDPoint(3, 4));
            // True:
            Console.WriteLine("pointA.Equals(list[0]): {0}",
pointA.Equals(list[0]));

            // Compare nullable to nullable and to non-nullable.
            TwoDPoint? pointC = null;
            TwoDPoint? pointD = null;
            // False:
            Console.WriteLine("pointA == (pointC = null) = {0}", pointA ==
pointC);
            // True:
            Console.WriteLine("pointC == pointD = {0}", pointC == pointD);

            TwoDPoint temp = new TwoDPoint(3, 4);
            pointC = temp;
```

```
            // True:
            Console.WriteLine("pointA == (pointC = 3,4) = {0}", pointA ==
pointC);

            pointD = temp;
            // True:
            Console.WriteLine("pointD == (pointC = 3,4) = {0}", pointD ==
pointC);

            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }

    /* Output:
        pointA.Equals(pointB) = True
        pointA == pointB = True
        Object.Equals(pointA, pointB) = True
        pointA.Equals(null) = False
        (pointA == null) = False
        (pointA != null) = True
        pointA.Equals(i) = False
        pointE.Equals(list[0]): True
        pointA == (pointC = null) = False
        pointC == pointD = True
        pointA == (pointC = 3,4) = True
        pointD == (pointC = 3,4) = True
    */
}
```

Pour les structs, l'implémentation par défaut de Object.Equals(Object) (qui est la version substituée dans System.ValueType) effectue une vérification de l'égalité des valeurs à l'aide de la réflexion pour comparer les valeurs de chaque champ dans le type. Quand un implémenteur substitue la méthode virtuelle `Equals` dans un struct, l'objectif est de fournir un moyen plus efficace d'effectuer la vérification de l'égalité des valeurs, et éventuellement, de baser la comparaison sur un sous-ensemble de champs ou de propriétés du struct.

Les opérateurs == et != ne peuvent pas fonctionner sur un struct, sauf si le struct les surcharge explicitement.

Voir aussi

- Comparaisons d'égalité
- Guide de programmation C#

Comment : tester l'égalité des références (Identité) (Guide de programmation C#)

Article • 07/04/2023

Il n'est pas utile d'implémenter une logique personnalisée pour prendre en charge les comparaisons d'égalité de références dans vos types. Cette fonctionnalité est fournie pour tous les types par la méthode Object.ReferenceEquals statique.

L'exemple suivant montre comment déterminer si deux variables affichent une *égalité de références*, à savoir qu'elles font référence à un même objet en mémoire.

L'exemple montre aussi pourquoi Object.ReferenceEquals retourne toujours `false` pour les types valeur et pourquoi vous ne devez pas utiliser ReferenceEquals pour déterminer l'égalité de chaînes.

Exemple

C#

```
using System.Text;

namespace TestReferenceEquality
{
    struct TestStruct
    {
        public int Num { get; private set; }
        public string Name { get; private set; }

        public TestStruct(int i, string s) : this()
        {
            Num = i;
            Name = s;
        }
    }

    class TestClass
    {
        public int Num { get; set; }
        public string? Name { get; set; }
    }

    class Program
    {
        static void Main()
```

```
        {
            // Demonstrate reference equality with reference types.
            #region ReferenceTypes

            // Create two reference type instances that have identical
values.
            TestClass tcA = new TestClass() { Num = 1, Name = "New
TestClass" };
            TestClass tcB = new TestClass() { Num = 1, Name = "New
TestClass" };

            Console.WriteLine("ReferenceEquals(tcA, tcB) = {0}",
                              Object.ReferenceEquals(tcA, tcB)); // false

            // After assignment, tcB and tcA refer to the same object.
            // They now have reference equality.
            tcB = tcA;
            Console.WriteLine("After assignment: ReferenceEquals(tcA, tcB) =
{0}",
                              Object.ReferenceEquals(tcA, tcB)); // true

            // Changes made to tcA are reflected in tcB. Therefore, objects
            // that have reference equality also have value equality.
            tcA.Num = 42;
            tcA.Name = "TestClass 42";
            Console.WriteLine("tcB.Name = {0} tcB.Num: {1}", tcB.Name,
tcB.Num);
            #endregion

            // Demonstrate that two value type instances never have
reference equality.
            #region ValueTypes

            TestStruct tsC = new TestStruct( 1, "TestStruct 1");

            // Value types are copied on assignment. tsD and tsC have
            // the same values but are not the same object.
            TestStruct tsD = tsC;
            Console.WriteLine("After assignment: ReferenceEquals(tsC, tsD) =
{0}",
                              Object.ReferenceEquals(tsC, tsD)); // false
            #endregion

            #region stringRefEquality
            // Constant strings within the same assembly are always interned
by the runtime.
            // This means they are stored in the same location in memory.
Therefore,
            // the two strings have reference equality although no
assignment takes place.
            string strA = "Hello world!";
            string strB = "Hello world!";
            Console.WriteLine("ReferenceEquals(strA, strB) = {0}",
                            Object.ReferenceEquals(strA, strB)); // true
```

```
            // After a new string is assigned to strA, strA and strB
            // are no longer interned and no longer have reference equality.
            strA = "Goodbye world!";
            Console.WriteLine("strA = \"{0}\" strB = \"{1}\"", strA, strB);

            Console.WriteLine("After strA changes, ReferenceEquals(strA,
strB) = {0}",
                              Object.ReferenceEquals(strA, strB)); // false

            // A string that is created at runtime cannot be interned.
            StringBuilder sb = new StringBuilder("Hello world!");
            string stringC = sb.ToString();
            // False:
            Console.WriteLine("ReferenceEquals(stringC, strB) = {0}",
                              Object.ReferenceEquals(stringC, strB));

            // The string class overloads the == operator to perform an
equality comparison.
            Console.WriteLine("stringC == strB = {0}", stringC == strB); //
true

            #endregion

            // Keep the console open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }
}

/* Output:
    ReferenceEquals(tcA, tcB) = False
    After assignment: ReferenceEquals(tcA, tcB) = True
    tcB.Name = TestClass 42 tcB.Num: 42
    After assignment: ReferenceEquals(tsC, tsD) = False
    ReferenceEquals(strA, strB) = True
    strA = "Goodbye world!" strB = "Hello world!"
    After strA changes, ReferenceEquals(strA, strB) = False
    ReferenceEquals(stringC, strB) = False
    stringC == strB = True
*/
```

L'implémentation de `Equals` dans la classe de base universelle System.Object assure aussi une vérification de l'égalité de références, mais il est préférable de ne pas l'utiliser, car si une classe se substitue à la méthode, vous risquez d'obtenir des résultats inattendus. Il en va de même pour les opérateurs `==` et `!=`. Quand ils s'appliquent à des types référence, le comportement par défaut de `==` et de `!=` est d'effectuer une vérification de l'égalité des références. Cependant, les classes dérivées peuvent surcharger l'opérateur pour effectuer une vérification d'égalité de valeurs. Pour réduire le risque d'erreurs, il est recommandé d'utiliser systématiquement ReferenceEquals quand il s'agit de déterminer si deux objets présentent une égalité de références.

Les chaînes constantes au sein d'un même assembly sont toujours intégrées par le runtime. Autrement dit, une seule instance de chaque chaîne littérale unique est conservée. En revanche, le runtime ne garantit pas que les chaînes créées au moment de l'exécution sont intégrées, ni que deux chaînes constantes égales d'assemblys différents sont intégrées.

Voir aussi

- Comparaisons d'égalité

Cast et conversions de types (guide de programmation C#)

Article • 07/04/2023

C# étant typé statiquement au moment de la compilation, une fois qu'une variable est déclarée elle ne peut plus être redéclarée et aucune valeur d'un autre type ne peut lui être assignée, sauf si ce type peut être converti de manière implicite au type de la variable. Par exemple, `string` ne peut pas être converti implicitement en `int`. Après avoir déclaré `i` comme `int`, vous ne pouvez donc pas lui assigner la chaîne « Hello », comme l'illustre le code suivant :

C#

```
int i;

// error CS0029: Cannot implicitly convert type 'string' to 'int'
i = "Hello";
```

Cependant, vous pouvez être amené à copier une valeur dans un paramètre de variable ou de méthode d'un autre type. C'est notamment le cas si vous avez une variable de type entier que vous devez passer à une méthode dont le paramètre est de type `double`. Il peut également arriver que vous deviez assigner une variable de classe à une variable de type interface. Les opérations de ce genre sont appelées « *conversions de types* ».
En C#, vous pouvez effectuer les conversions suivantes :

- **Conversions implicites** : aucune syntaxe spéciale n'est requise, car la conversion aboutit toujours et aucune donnée n'est perdue. Citons par exemple les conversions de types intégraux en d'autres plus importants, et les conversions de classes dérivées en classes de base.

- **Conversions explicites (casts)** : les conversions explicites nécessitent une expression cast. Un cast est exigé quand les informations peuvent être perdues durant la conversion, ou quand la conversion peut échouer pour d'autres raisons. Exemples classiques : conversion numérique en type qui a moins de précision ou une plus petite plage, et conversion d'une instance de classe de base en classe dérivée.

- **Conversions définies par l'utilisateur** : les conversions définies par l'utilisateur sont effectuées par des méthodes spéciales que vous pouvez définir pour permettre des conversions explicites ou implicites entre des types personnalisés qui n'ont pas de

relation classe de base/classe dérivée. Pour plus d'informations, consultez Opérateurs de conversion définie par l'utilisateur.

- **Conversions avec les classes d'assistance** : pour effectuer une conversion entre des types non compatibles, tels que des entiers et des objets System.DateTime ou des chaînes hexadécimales et des tableaux d'octets, vous pouvez utiliser la classe System.BitConverter, la classe System.Convert et les méthodes `Parse` des types numériques intégrés, comme Int32.Parse. Pour plus d'informations, consultez Guide pratique pour convertir un tableau d'octets en int, Guide pratique pour convertir une chaîne en nombre et Guide pratique pour effectuer une conversion entre des chaînes hexadécimales et des types numériques.

Conversions implicites

Pour les types numériques intégrés, une conversion implicite peut être effectuée quand la valeur à stocker peut tenir dans la variable sans être tronquée ni arrondie. Pour les types intégraux, cela signifie que la plage du type de source est un sous-ensemble approprié de la plage du type cible. Par exemple, une variable de type long (entier sur 64 bits) peut stocker n'importe quelle valeur pouvant être stockée par un int (entier sur 32 bits). Dans l'exemple suivant, le compilateur convertit implicitement la valeur de `num` à droite en type `long` avant de l'assigner à `bigNum`.

C#

```
// Implicit conversion. A long can
// hold any value an int can hold, and more!
int num = 2147483647;
long bigNum = num;
```

Pour obtenir la liste complète de toutes les conversions numériques implicites, consultez la section Conversions numériques implicites de l'article Conversions numériques intégrées.

Pour les types référence, il existe toujours une conversion implicite entre une classe et l'une de ses interfaces ou classes de base directes ou indirectes. Aucune syntaxe spéciale n'est nécessaire, car une classe dérivée contient toujours tous les membres d'une classe de base.

C#

```
Derived d = new Derived();
```

```
// Always OK.
Base b = d;
```

Conversions explicites

Toutefois, si une conversion ne peut pas être réalisée sans risque de perte d'informations, le compilateur exige une conversion explicite, aussi appelée *cast*. Un cast est un moyen d'informer explicitement le compilateur que vous prévoyez d'effectuer la conversion et que vous êtes conscient que des pertes de données peuvent se produire, ou que le cast peut échouer au moment de l'exécution. Pour effectuer un cast, spécifiez le type voulu entre parenthèses devant la valeur ou la variable à convertir. Le programme suivant convertit un double en int. Le programme ne sera pas compilé sans le cast.

C#

```
class Test
{
    static void Main()
    {
        double x = 1234.7;
        int a;
        // Cast double to int.
        a = (int)x;
        System.Console.WriteLine(a);
    }
}
// Output: 1234
```

Pour obtenir la liste des conversions numériques explicites prises en charge, consultez la section Conversions numériques explicites de l'article Conversions numériques intégrées.

Pour les types référence, un cast explicite est exigé si vous devez effectuer une conversion d'un type de base en type dérivé :

C#

```
// Create a new derived type.
Giraffe g = new Giraffe();

// Implicit conversion to base type is safe.
Animal a = g;

// Explicit conversion is required to cast back
// to derived type. Note: This will compile but will
```

```
// throw an exception at run time if the right-side
// object is not in fact a Giraffe.
Giraffe g2 = (Giraffe)a;
```

Une opération cast entre types référence ne change pas le type au moment de l'exécution de l'objet sous-jacent. Elle change seulement le type de la valeur utilisée comme référence à cet objet. Pour plus d'informations, consultez Polymorphisme.

Exceptions de conversion de type au moment de l'exécution

Dans certaines conversions de types référence, le compilateur ne peut pas déterminer si un cast sera valide. Il est possible qu'une opération cast dont la compilation fonctionne échoue au moment de l'exécution. Comme l'illustre l'exemple suivant, un cast de type qui échoue au moment de l'exécution provoque la levée d'une exception InvalidCastException.

C#

```
class Animal
{
    public void Eat() => System.Console.WriteLine("Eating.");

    public override string ToString() => "I am an animal.";
}

class Reptile : Animal { }
class Mammal : Animal { }

class UnSafeCast
{
    static void Main()
    {
        Test(new Mammal());

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }

    static void Test(Animal a)
    {
        // System.InvalidCastException at run time
        // Unable to cast object of type 'Mammal' to type 'Reptile'
        Reptile r = (Reptile)a;
    }
}
```

La méthode `Test` possède un paramètre `Animal`, et le fait de transformer explicitement l'argument `a` en `Reptile` constitue une hypothèse dangereuse. Il est préférable de vérifier le type plutôt que de faire des hypothèses. C# fournit l'opérateur is pour vous permettre de tester la compatibilité avant d'effectuer réellement un cast. Pour plus d'informations, consultez Guide pratique pour caster de manière sécurisée avec les critères spéciaux, ainsi que les opérateurs is et as.

spécification du langage C#

Pour plus d'informations, consultez la section Conversions de la spécification du langage C#.

Voir aussi

- Guide de programmation C#
- Types
- Expression cast
- Opérateurs de conversion définie par l'utilisateur
- Conversion de type généralisée
- Comment convertir une chaîne en nombre

Boxing et unboxing (Guide de programmation C#)

Article • 07/04/2023

Le boxing est la conversion d'un type valeur en type `object` ou en un type interface implémenté par ce type valeur. Quand le common language runtime (CLR) exécute un boxing d'un type valeur, il enveloppe la valeur dans une instance System.Object et la stocke sur le tas managé. L'unboxing extrait le type valeur de l'objet. La conversion boxing est implicite ; la conversion unboxing est explicite. Le concept de boxing et de unboxing repose sur la vue unifiée par C# du système de type, dans lequel une valeur de n'importe quel type peut être traitée en tant qu'objet.

Dans l'exemple suivant, la variable de type entier `i` est convertie (*boxed*) et assignée à l'objet `o`.

C#

```
int i = 123;
// The following line boxes i.
object o = i;
```

L'objet `o` peut ensuite être unboxed et assigné à la variable de type entier `i` :

C#

```
o = 123;
i = (int)o;  // unboxing
```

Les exemples suivants montrent comment le boxing est utilisé dans C#.

C#

```
// String.Concat example.
// String.Concat has many versions. Rest the mouse pointer on
// Concat in the following statement to verify that the version
// that is used here takes three object arguments. Both 42 and
// true must be boxed.
Console.WriteLine(String.Concat("Answer", 42, true));

// List example.
// Create a list of objects to hold a heterogeneous collection
// of elements.
List<object> mixedList = new List<object>();
```

```
// Add a string element to the list.
mixedList.Add("First Group:");

// Add some integers to the list.
for (int j = 1; j < 5; j++)
{
    // Rest the mouse pointer over j to verify that you are adding
    // an int to a list of objects. Each element j is boxed when
    // you add j to mixedList.
    mixedList.Add(j);
}

// Add another string and more integers.
mixedList.Add("Second Group:");
for (int j = 5; j < 10; j++)
{
    mixedList.Add(j);
}

// Display the elements in the list. Declare the loop variable by
// using var, so that the compiler assigns its type.
foreach (var item in mixedList)
{
    // Rest the mouse pointer over item to verify that the elements
    // of mixedList are objects.
    Console.WriteLine(item);
}

// The following loop sums the squares of the first group of boxed
// integers in mixedList. The list elements are objects, and cannot
// be multiplied or added to the sum until they are unboxed. The
// unboxing must be done explicitly.
var sum = 0;
for (var j = 1; j < 5; j++)
{
    // The following statement causes a compiler error: Operator
    // '*' cannot be applied to operands of type 'object' and
    // 'object'.
    //sum += mixedList[j] * mixedList[j]);

    // After the list elements are unboxed, the computation does
    // not cause a compiler error.
    sum += (int)mixedList[j] * (int)mixedList[j];
}

// The sum displayed is 30, the sum of 1 + 4 + 9 + 16.
Console.WriteLine("Sum: " + sum);

// Output:
// Answer42True
// First Group:
// 1
// 2
// 3
// 4
```

```
// Second Group:
// 5
// 6
// 7
// 8
// 9
// Sum: 30
```

Performances

Par rapport aux assignations simples, le boxing et l'unboxing sont des processus qui coûtent cher en calcul. Lorsqu'un type valeur est boxed, un nouvel objet doit être alloué et construit. À un degré moindre, le cast requis pour l'unboxing coûte également cher en calcul. Pour plus d'informations, consultez Performances.

Boxing

Le boxing est utilisé pour stocker des types valeur dans le tas rassemblé par garbage collection. Le boxing est une conversion implicite d'un type valeur en type `object` ou en un type interface implémenté par ce type valeur. Le boxing d'un type valeur alloue une instance d'objet sur le tas et copie la valeur dans le nouvel objet.

Dans l'exemple suivant, une variable de type valeur est déclarée :

C#

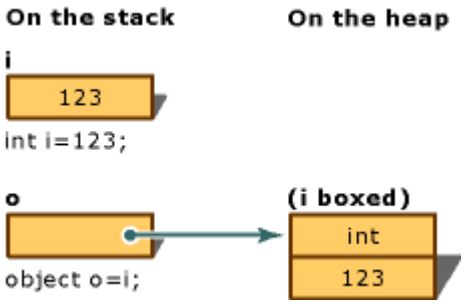
```
int i = 123;
```

L'instruction ci-dessous réalise implicitement une opération de boxing sur la variable `i` :

C#

```
// Boxing copies the value of i into object o.
object o = i;
```

Le résultat de cette instruction crée, sur la pile, un objet `o` qui fait référence à une valeur de type `int` sur le tas. Cette valeur est une copie de la valeur de type valeur qui a été assignée à la variable `i`. La différence entre les deux variables, `i` et `o`, est illustrée dans l'image de conversion boxing ci-dessous :



Il est également possible, mais jamais obligatoire, d'effectuer un boxing explicite comme dans l'exemple suivant :

C#

```
int i = 123;
object o = (object)i;  // explicit boxing
```

Exemple

Cet exemple utilise le boxing pour convertir une variable `i` (entier) en un objet `o`. Ensuite, la valeur `i` stockée dans la variable `123` est remplacée par la valeur `456`. L'exemple montre que le type valeur d'origine et que l'objet boxed utilisent des emplacements de mémoire distincts et peuvent, par conséquent, stocker des valeurs différentes.

C#

```
class TestBoxing
{
    static void Main()
    {
        int i = 123;

        // Boxing copies the value of i into object o.
        object o = i;

        // Change the value of i.
        i = 456;

        // The change in i doesn't affect the value stored in o.
        System.Console.WriteLine("The value-type value = {0}", i);
        System.Console.WriteLine("The object-type value = {0}", o);
    }
}
/* Output:
    The value-type value = 456
    The object-type value = 123
*/
```

Unboxing

L'unboxing est une conversion explicite du type `object` en un type valeur, ou d'un type interface en un type valeur qui implémente l'interface. Une opération d'unboxing comprend les étapes suivantes :

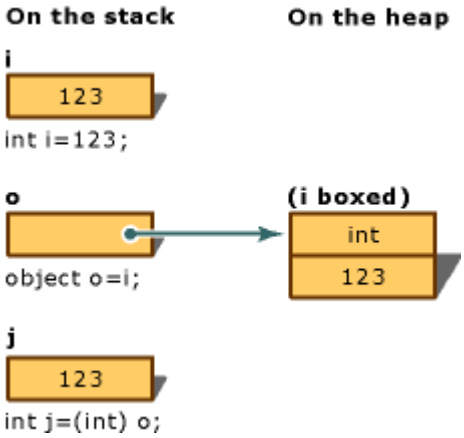
- Vérification de l'instance de l'objet pour s'assurer qu'il s'agit bien d'une valeur boxed du type valeur spécifié.
- Copie de la valeur de l'instance dans la variable de type valeur.

Les instructions suivantes expliquent les opérations de boxing et d'unboxing :

C#

```
int i = 123;      // a value type
object o = i;     // boxing
int j = (int)o;   // unboxing
```

L'illustration suivante montre le résultat de ces instructions :



Pour que l'unboxing de types valeur réussisse au moment de l'exécution, l'élément qui est unboxed doit être une référence à un objet précédemment créé par boxing d'une instance de ce type valeur. La tentative d'extraction de `null` provoque un NullReferenceException. La tentative d'extraction d'une référence vers un type de valeur incompatible provoque un InvalidCastException.

Exemple

L'exemple suivant montre un cas d'unboxing non valide et l'`InvalidCastException` qui en résulte. Avec `try` et `catch`, un message d'erreur est affiché lorsque l'erreur se produit.

```
```

C#

```
class TestUnboxing
{
    static void Main()
    {
        int i = 123;
        object o = i;  // implicit boxing

        try
        {
            int j = (short)o;  // attempt to unbox

            System.Console.WriteLine("Unboxing OK.");
        }
        catch (System.InvalidCastException e)
        {
            System.Console.WriteLine("{0} Error: Incorrect unboxing.",
e.Message);
        }
    }
}
```

Sortie de ce programme :

```
Specified cast is not valid. Error: Incorrect unboxing.
```

Si vous modifiez l'instruction :

C#

```
int j = (short)o;
```

to:

C#

```
int j = (int)o;
```

la conversion sera réalisée, avec le résultat suivant :

```
Unboxing OK.
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Guide de programmation C#
- Types référence
- Types de valeur

Guide pratique pour convertir un tableau d'octets en valeur int (Guide de programmation C#)

Article • 07/04/2023

Cet exemple montre comment utiliser la classe BitConverter pour convertir un tableau d'octets en valeur int, puis la reconvertir en tableau d'octets. Vous devrez peut-être convertir des octets en un type de données intégré après avoir lu les octets sur le réseau. Outre la méthode ToInt32(Byte[], Int32) de l'exemple, le tableau suivant répertorie les méthodes dans la classe BitConverter qui convertissent des octets (d'un tableau d'octets) en d'autres types intégrés.

Type retourné	Méthode
`bool`	ToBoolean(Byte[], Int32)
`char`	ToChar(Byte[], Int32)
`double`	ToDouble(Byte[], Int32)
`short`	ToInt16(Byte[], Int32)
`int`	ToInt32(Byte[], Int32)
`long`	ToInt64(Byte[], Int32)
`float`	ToSingle(Byte[], Int32)
`ushort`	ToUInt16(Byte[], Int32)
`uint`	ToUInt32(Byte[], Int32)
`ulong`	ToUInt64(Byte[], Int32)

Exemples

Cet exemple initialise un tableau d'octets, inverse le tableau si l'architecture de l'ordinateur est little-endian (autrement dit, l'octet le moins significatif est stocké en premier), puis appelle la méthode ToInt32(Byte[], Int32) pour convertir quatre octets du tableau en valeur `int`. Le deuxième argument de ToInt32(Byte[], Int32) spécifie l'index de départ du tableau d'octets.

> ⓘ Notes
>
> La sortie peut varier en fonction du caractère endian de l'architecture de votre ordinateur.

C#

```
byte[] bytes = { 0, 0, 0, 25 };

// If the system architecture is little-endian (that is, little end first),
// reverse the byte array.
if (BitConverter.IsLittleEndian)
    Array.Reverse(bytes);

int i = BitConverter.ToInt32(bytes, 0);
Console.WriteLine("int: {0}", i);
// Output: int: 25
```

Dans cet exemple, la méthode GetBytes(Int32) de la classe BitConverter est appelée pour convertir une valeur `int` en tableau d'octets.

> ⓘ Notes
>
> La sortie peut varier en fonction du caractère endian de l'architecture de votre ordinateur.

C#

```
byte[] bytes = BitConverter.GetBytes(201805978);
Console.WriteLine("byte array: " + BitConverter.ToString(bytes));
// Output: byte array: 9A-50-07-0C
```

Voir aussi

- BitConverter
- IsLittleEndian
- Types

Guide pratique pour convertir une chaîne en nombre (Guide de programmation C#)

Article • 07/04/2023

Vous convertissez une `string` en nombre en appelant la méthode `Parse` ou `TryParse` trouvée sur les types numériques (`int`, `long`, `double` etc.) ou en utilisant des méthodes de la classe System.Convert.

Il est légèrement plus efficace et direct d'appeler une méthode `TryParse` (par exemple int.TryParse("11", out number)) ou une méthode `Parse` (par exemple, var number = int.Parse("11")). L'utilisation d'une méthode Convert s'avère plus utile pour les objets généraux qui implémentent IConvertible.

Vous pouvez utiliser les méthodes `Parse` ou `TryParse` sur le type numérique que doit contenir la chaîne, notamment le type System.Int32. La méthode Convert.ToInt32 utilise Parse en interne. La méthode `Parse` retourne le nombre converti ; la méthode `TryParse` retourne une valeur qui indique si la conversion a réussi et retourne le nombre converti dans un paramètre `out`. Si la chaîne n'est pas dans un format valide, `Parse` lève une exception, mais `TryParse` retourne `false`. Lorsque vous appelez une méthode `Parse`, vous devez toujours utiliser la gestion des exceptions pour intercepter une FormatException lorsque l'opération d'analyse échoue.

Appeler des méthodes Parse ou TryParse

Les méthodes `Parse` et `TryParse` ignorent l'espace blanc au début et à la fin de la chaîne, mais tous les autres caractères doivent être des caractères qui forment le type numérique approprié (`int`, `long`, `ulong`, `float`, `decimal` etc.). La présence d'un espace blanc dans la chaîne qui forme le nombre génère une erreur. Par exemple, vous pouvez utiliser `decimal.TryParse` pour analyser « 10 », « 10.3 » ou « 10 », mais vous ne pouvez pas utiliser cette méthode pour analyser 10 à partir de « 10X », « 1 0 » (notez l'espace incorporé), « 10 .3 » (notez l'espace incorporé), « 10e1 » (`float.TryParse` fonctionne ici) et ainsi de suite. Par ailleurs, une chaîne dont la valeur est `null` ou String.Empty échoue à l'analyse. Vous pouvez rechercher une chaîne Null ou vide avant de tenter de l'analyser en appelant la méthode String.IsNullOrEmpty.

L'exemple suivant illustre des appels à `Parse` et à `TryParse` qui ont réussi ou échoué.

C#

```
using System;

public static class StringConversion
{
    public static void Main()
    {
        string input = String.Empty;
        try
        {
            int result = Int32.Parse(input);
            Console.WriteLine(result);
        }
        catch (FormatException)
        {
            Console.WriteLine($"Unable to parse '{input}'");
        }
        // Output: Unable to parse ''

        try
        {
            int numVal = Int32.Parse("-105");
            Console.WriteLine(numVal);
        }
        catch (FormatException e)
        {
            Console.WriteLine(e.Message);
        }
        // Output: -105

        if (Int32.TryParse("-105", out int j))
        {
            Console.WriteLine(j);
        }
        else
        {
            Console.WriteLine("String could not be parsed.");
        }
        // Output: -105

        try
        {
            int m = Int32.Parse("abc");
        }
        catch (FormatException e)
        {
            Console.WriteLine(e.Message);
        }
        // Output: Input string was not in a correct format.

        const string inputString = "abc";
        if (Int32.TryParse(inputString, out int numValue))
        {
```

```
            Console.WriteLine(numValue);
        }
        else
        {
            Console.WriteLine($"Int32.TryParse could not parse
'{inputString}' to an int.");
        }
        // Output: Int32.TryParse could not parse 'abc' to an int.
    }
}
```

L'exemple suivant illustre une approche de l'analyse d'une chaîne prévoyant d'inclure des caractères numériques de début (y compris des caractères hexadécimaux) et des caractères non numériques de fin. Il attribue dès le début des caractères valides à une nouvelle chaîne avant d'appeler la méthode TryParse. Étant donné que les chaînes à analyser contiennent peu de caractères, l'exemple appelle la méthode String.Concat pour attribuer des caractères valides à une nouvelle chaîne. Pour une chaîne plus grande, la classe StringBuilder peut être utilisée à la place.

C#

```
using System;

public static class StringConversion
{
    public static void Main()
    {
        var str = "  10FFxxx";
        string numericString = string.Empty;
        foreach (var c in str)
        {
            // Check for numeric characters (hex in this case) or leading or
trailing spaces.
            if ((c >= '0' && c <= '9') || (char.ToUpperInvariant(c) >= 'A'
&& char.ToUpperInvariant(c) <= 'F') || c == ' ')
            {
                numericString = string.Concat(numericString, c.ToString());
            }
            else
            {
                break;
            }
        }

        if (int.TryParse(numericString,
System.Globalization.NumberStyles.HexNumber, null, out int i))
        {
            Console.WriteLine($"'{str}' --> '{numericString}' --> {i}");
        }
        // Output: '  10FFxxx' --> '  10FF' --> 4351
```

```
        str = "   -10FFXXX";
        numericString = "";
        foreach (char c in str)
        {
            // Check for numeric characters (0-9), a negative sign, or
leading or trailing spaces.
            if ((c >= '0' && c <= '9') || c == ' ' || c == '-')
            {
                numericString = string.Concat(numericString, c);
            }
            else
            {
                break;
            }
        }

        if (int.TryParse(numericString, out int j))
        {
            Console.WriteLine($"'{str}' --> '{numericString}' --> {j}");
        }
        // Output: '   -10FFXXX' --> '   -10' --> -10
    }
}
```

Appeler des méthodes Convert

Le tableau suivant répertorie quelques unes des méthodes de la classe Convert que vous pouvez utiliser pour convertir une chaîne en nombre.

Type numérique	Méthode
`decimal`	ToDecimal(String)
`float`	ToSingle(String)
`double`	ToDouble(String)
`short`	ToInt16(String)
`int`	ToInt32(String)
`long`	ToInt64(String)
`ushort`	ToUInt16(String)
`uint`	ToUInt32(String)
`ulong`	ToUInt64(String)

L'exemple suivant appelle la méthode Convert.ToInt32(String) pour convertir une chaîne d'entrée en un int. L'exemple intercepte les deux exceptions les plus courantes qui peuvent être levées par cette méthode, FormatException et OverflowException. Si le nombre obtenu peut être incrémenté sans dépasser Int32.MaxValue, l'exemple ajoute 1 au résultat et affiche la sortie.

C#

```
using System;

public class ConvertStringExample1
{
    static void Main(string[] args)
    {
        int numVal = -1;
        bool repeat = true;

        while (repeat)
        {
            Console.Write("Enter a number between -2,147,483,648 and
+2,147,483,647 (inclusive): ");

            string? input = Console.ReadLine();

            // ToInt32 can throw FormatException or OverflowException.
            try
            {
                numVal = Convert.ToInt32(input);
                if (numVal < Int32.MaxValue)
                {
                    Console.WriteLine("The new value is {0}", ++numVal);
                }
                else
                {
                    Console.WriteLine("numVal cannot be incremented beyond
its current value");
                }
            }
            catch (FormatException)
            {
                Console.WriteLine("Input string is not a sequence of
digits.");
            }
            catch (OverflowException)
            {
                Console.WriteLine("The number cannot fit in an Int32.");
            }

            Console.Write("Go again? Y/N: ");
            string? go = Console.ReadLine();
            if (go?.ToUpper() != "Y")
            {
                repeat = false;
```

```
            }
        }
    }
}
// Sample Output:
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive):
473
//   The new value is 474
//   Go again? Y/N: y
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive):
2147483647
//   numVal cannot be incremented beyond its current value
//   Go again? Y/N: y
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive):
-1000
//   The new value is -999
//   Go again? Y/N: n
```

Comment effectuer une conversion entre des chaînes hexadécimales et des types numériques (Guide de programmation C#)

Article • 10/05/2023

Ces exemples montrent comment effectuer les tâches suivantes :

- Obtenir la valeur hexadécimale de chaque caractère dans une chaîne (string).
- Obtenir la valeur char qui correspond à chaque valeur d'une chaîne hexadécimale.
- Convertir une `string` hexadécimale en int.
- Convertir une `string` hexadécimale en float.
- Convertir un tableau d'octets en `string` hexadécimale.

Exemples

Cet exemple génère la valeur hexadécimale de chaque caractère dans une `string`. Il convertit d'abord la `string` en un tableau de caractères. Ensuite, il appelle ToInt32(Char) sur chaque caractère afin d'obtenir sa valeur numérique. Pour finir, il représente le nombre sous forme hexadécimale dans une `string`.

C#

```
string input = "Hello World!";
char[] values = input.ToCharArray();
foreach (char letter in values)
{
    // Get the integral value of the character.
    int value = Convert.ToInt32(letter);
    // Convert the integer value to a hexadecimal value in string form.
    Console.WriteLine($"Hexadecimal value of {letter} is {value:X}");
}
/* Output:
    Hexadecimal value of H is 48
    Hexadecimal value of e is 65
    Hexadecimal value of l is 6C
    Hexadecimal value of l is 6C
    Hexadecimal value of o is 6F
    Hexadecimal value of   is 20
```

```
    Hexadecimal value of W is 57
    Hexadecimal value of o is 6F
    Hexadecimal value of r is 72
    Hexadecimal value of l is 6C
    Hexadecimal value of d is 64
    Hexadecimal value of ! is 21
 */
```

Cet exemple analyse une `string` de valeurs hexadécimales et génère le caractère correspondant à chacune d'elles. Il appelle d'abord la méthode Split(Char[]) pour obtenir chaque valeur hexadécimale sous la forme d'une `string` individuelle dans un tableau. Ensuite, il appelle ToInt32(String, Int32) pour convertir la valeur hexadécimale en valeur décimale représentée sous forme d'objet int. Il illustre deux manières d'obtenir le caractère correspondant à ce code de caractère. La première technique utilise `string`, qui retourne le caractère correspondant à l'argument entier sous forme de ConvertFromUtf32(Int32). La deuxième technique effectue un cast explicite de l'`int` en char.

C#

```
string hexValues = "48 65 6C 6C 6F 20 57 6F 72 6C 64 21";
string[] hexValuesSplit = hexValues.Split(' ');
foreach (string hex in hexValuesSplit)
{
    // Convert the number expressed in base-16 to an integer.
    int value = Convert.ToInt32(hex, 16);
    // Get the character corresponding to the integral value.
    string stringValue = Char.ConvertFromUtf32(value);
    char charValue = (char)value;
    Console.WriteLine("hexadecimal value = {0}, int value = {1}, char value
= {2} or {3}",
                      hex, value, stringValue, charValue);
}
/* Output:
    hexadecimal value = 48, int value = 72, char value = H or H
    hexadecimal value = 65, int value = 101, char value = e or e
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 6F, int value = 111, char value = o or o
    hexadecimal value = 20, int value = 32, char value =   or
    hexadecimal value = 57, int value = 87, char value = W or W
    hexadecimal value = 6F, int value = 111, char value = o or o
    hexadecimal value = 72, int value = 114, char value = r or r
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 64, int value = 100, char value = d or d
    hexadecimal value = 21, int value = 33, char value = ! or !
*/
```

Cet exemple montre une autre façon de convertir un `string` hexadécimal en entier, en appelant la méthode Parse(String, NumberStyles).

C#

```
string hexString = "8E2";
int num = Int32.Parse(hexString,
System.Globalization.NumberStyles.HexNumber);
Console.WriteLine(num);
//Output: 2274
```

L'exemple suivant montre comment convertir un `string` hexadécimal en float à l'aide de la classe System.BitConverter et de la méthode UInt32.Parse.

C#

```


string hexString = "43480170";
uint num = uint.Parse(hexString,
System.Globalization.NumberStyles.AllowHexSpecifier);

byte[] floatVals = BitConverter.GetBytes(num);
float f = BitConverter.ToSingle(floatVals, 0);
Console.WriteLine("float convert = {0}", f);

// Output: 200.0056
```

L'exemple suivant montre comment convertir un tableau d'octets en chaîne hexadécimale à l'aide de la classe System.BitConverter.

C#

```
byte[] vals = { 0x01, 0xAA, 0xB1, 0xDC, 0x10, 0xDD };

string str = BitConverter.ToString(vals);
Console.WriteLine(str);

str = BitConverter.ToString(vals).Replace("-", "");
Console.WriteLine(str);

/*Output:
  01-AA-B1-DC-10-DD
  01AAB1DC10DD
 */
```

L'exemple suivant montre comment convertir un tableau d'octets en chaîne hexadécimale en appelant la méthode Convert.ToHexString introduite dans .NET 5.0.

C#

```
byte[] array = { 0x64, 0x6f, 0x74, 0x63, 0x65, 0x74 };

string hexValue = Convert.ToHexString(array);
Console.WriteLine(hexValue);

/*Output:
  646F74636574
 */
```

Voir aussi

- Chaînes de format numériques standard
- Types
- Comment déterminer si une chaîne représente une valeur numérique

Utilisation de type dynamique

Article • 23/06/2023

Le type `dynamic` est un type statique ; toutefois, un objet de type `dynamic` ignore la vérification des types statiques. Dans la plupart des cas, il fonctionne comme s'il était de type `object`. Le compilateur suppose qu'un élément `dynamic` prend en charge n'importe quelle opération. Par conséquent, vous n'avez pas besoin de déterminer si l'objet obtient sa valeur à partir d'une COM API, à partir d'un langage dynamique tel que IronPython, du modèle DOM (Document Object Model) HTML, de la réflexion ou d'un autre endroit dans le programme. Toutefois, si le code n'est pas valide, les erreurs s'affichent au moment de l'exécution.

Par exemple, si la méthode d'instance `exampleMethod1` dans le code suivant n'a qu'un seul paramètre, le compilateur reconnaît que le premier appel à la méthode, `ec.exampleMethod1(10, 4)`, n'est pas valide, car il contient deux arguments. Cet appel entraîne une erreur du compilateur. Le compilateur ne vérifie pas le deuxième appel à la méthode, `dynamic_ec.exampleMethod1(10, 4)`, car le type de `dynamic_ec` est `dynamic`. Par conséquent, aucune erreur de compilateur n'est signalée. Toutefois, l'erreur ne s'échappe pas indéfiniment. Elle apparaît au moment de l'exécution et provoque une exception d'exécution.

C#

```
static void Main(string[] args)
{
    ExampleClass ec = new ExampleClass();
    // The following call to exampleMethod1 causes a compiler error
    // if exampleMethod1 has only one parameter. Uncomment the line
    // to see the error.
    //ec.exampleMethod1(10, 4);

    dynamic dynamic_ec = new ExampleClass();
    // The following line is not identified as an error by the
    // compiler, but it causes a run-time exception.
    dynamic_ec.exampleMethod1(10, 4);

    // The following calls also do not cause compiler errors, whether
    // appropriate methods exist or not.
    dynamic_ec.someMethod("some argument", 7, null);
    dynamic_ec.nonexistentMethod();
}
```

C#

```
class ExampleClass
{
    public ExampleClass() { }
    public ExampleClass(int v) { }

    public void exampleMethod1(int i) { }

    public void exampleMethod2(string str) { }
}
```

Le rôle du compilateur dans ces exemples consiste à empaqueter des informations sur ce que chaque instruction propose de faire à l'objet ou à l'expression `dynamic`. Le runtime examine les informations stockées et toute instruction qui n'est pas valide provoque une exception d'exécution.

Le résultat de la plupart des opérations dynamiques est lui-même de type `dynamic`. Par exemple, si vous placez le pointeur de la souris sur `testSum` dans l'exemple suivant, IntelliSense affiche le type **(variable locale) dynamic testSum**.

C#

```
dynamic d = 1;
var testSum = d + 3;
// Rest the mouse pointer over testSum in the following statement.
System.Console.WriteLine(testSum);
```

Les opérations dans lesquelles le résultat n'est pas `dynamic` sont les suivantes :

- Les conversions de `dynamic` vers un autre type.
- Les appels de constructeur qui incluent des arguments de type `dynamic`.

Par exemple, le type de `testInstance` dans la déclaration suivante est `ExampleClass`, et non pas `dynamic` :

C#

```
var testInstance = new ExampleClass(d);
```

Conversions

Les conversions entre des objets dynamiques et d'autres types sont faciles à exécuter. Les conversions permettent au développeur de basculer entre le comportement dynamique et non dynamique.

Vous pouvez convertir tout en `dynamic` implicitement, comme illustré dans les exemples suivants.

C#

```
dynamic d1 = 7;
dynamic d2 = "a string";
dynamic d3 = System.DateTime.Today;
dynamic d4 = System.Diagnostics.Process.GetProcesses();
```

À l'inverse, vous pouvez appliquer dynamiquement n'importe quelle conversion implicite à n'importe quelle expression de type `dynamic`.

C#

```
int i = d1;
string str = d2;
DateTime dt = d3;
System.Diagnostics.Process[] procs = d4;
```

Résolution de surcharge avec des arguments de type dynamic

La résolution de surcharge a lieu au moment de l'exécution, et non pas au moment de la compilation, si un ou plusieurs arguments d'un appel de méthode sont de type `dynamic`, ou si le récepteur de l'appel de méthode est de type `dynamic`. Dans l'exemple suivant, si la seule méthode `exampleMethod2` accessible prend un argument de chaîne, l'envoi de `d1` en tant qu'argument n'entraîne pas d'erreur du compilateur, mais provoque une exception d'exécution. La résolution de surcharge échoue au moment de l'exécution car le type de `d1` est `int`, et `exampleMethod2` exige une chaîne.

C#

```
// Valid.
ec.exampleMethod2("a string");

// The following statement does not cause a compiler error, even though ec
is not
// dynamic. A run-time exception is raised because the run-time type of d1
is int.
ec.exampleMethod2(d1);
// The following statement does cause a compiler error.
//ec.exampleMethod2(7);
```

Dynamic Language Runtime

Le runtime de langage dynamique (DLR) fournit l'infrastructure qui prend en charge le type `dynamic` en C#, ainsi que l'implémentation de langages de programmation dynamiques tels que IronPython et IronRuby. Pour plus d'informations sur le DLR, consultez Vue d'ensemble du Dynamic Language Runtime.

COM interop

De nombreuses méthodes COM autorisent une variation des types d'arguments et du type de retour en désignant les types comme `object`. L'interopérabilité COM nécessite un cast explicite des valeurs à coordonner avec des variables fortement typées en C#. Si vous effectuez la compilation à l'aide de l'option **EmbedInteropTypes** (Options de compilateur C#), l'introduction du type `dynamic` vous permet de traiter les occurrences d'`object` dans les signatures COM comme si elles étaient de type `dynamic`, vous évitant ainsi une grande partie des opérations de cast. Pour plus d'informations sur l'utilisation du type `dynamic` avec des objets COM, consultez l'article sur Comment accéder aux objets Office Interop à l'aide des fonctionnalités C#.

Articles connexes

Intitulé	Description
dynamic	Décrit l'utilisation du mot clé `dynamic`.
Vue d'ensemble du Dynamic Language Runtime	Fournit une vue d'ensemble du DLR, qui est un environnement d'exécution ajoutant au Common Language Runtime (CLR) un ensemble de services pour les langages dynamiques.
Procédure pas à pas : création et utilisation d'objets dynamiques	Fournit des instructions pas à pas pour la création d'un objet dynamique personnalisé et pour la création d'un projet qui accède à une bibliothèque `IronPython`.

Procédure pas-à-pas : création et utilisation d'objets dynamiques en C#

Article • 09/05/2023

Les objets dynamiques exposent des membres tels que les propriétés et les méthodes au moment de l'exécution et non lors de la compilation. Les objets dynamiques vous permettent de créer des objets utilisables avec des structures qui ne correspondent pas à un type ou un format statique. Par exemple, vous pouvez utiliser un objet dynamique pour référencer le modèle DOM (Document Object Model) HTML, qui peut contenir n'importe quelle combinaison d'attributs et d'éléments de balisage HTML valides. Étant donné que chaque document HTML est unique, les membres d'un document HTML particulier sont déterminés au moment de l'exécution. L'une des méthodes courantes pour référencer un attribut d'un élément HTML consiste à passer le nom de l'attribut à la méthode `GetProperty` de l'élément. Pour référencer l'attribut `id` de l'élément HTML `<div id="Div1">`, vous obtenez d'abord une référence à l'élément `<div>`, puis utilisez `divElement.GetProperty("id")`. Si vous utilisez un objet dynamique, vous pouvez référencer l'attribut `id` comme `divElement.id`.

Les objets dynamiques permettent également d'accéder facilement aux langages dynamiques comme IronPython et IronRuby. Vous pouvez utiliser un objet dynamique pour faire référence à un script dynamique interprété au moment de l'exécution.

Vous référencez un objet dynamique à l'aide de la liaison tardive. Vous spécifiez le type d'un objet à liaison tardive en tant que `dynamic`. Pour plus d'informations, consultez dynamique.

Vous pouvez créer des objets dynamiques personnalisés en utilisant les classes contenues dans l'espace de noms System.Dynamic. Par exemple, vous pouvez créer un ExpandoObject et spécifier les membres de cet objet au moment de l'exécution. Vous pouvez aussi créer votre propre type qui hérite de la classe DynamicObject. Vous pouvez ensuite substituer les membres de la classe DynamicObject pour fournir des fonctionnalités dynamiques au moment de l'exécution.

Cet article contient deux procédures pas à pas indépendantes :

- Créer un objet personnalisé qui expose dynamiquement le contenu d'un fichier texte comme propriétés d'un objet.
- Créer un projet qui utilise une bibliothèque `IronPython`.

Prérequis

- Visual Studio 2022 version 17.3 ou ultérieure avec la charge de travail de **développement de bureau .NET** installée. Le SDK .NET 7 est inclus lorsque vous sélectionnez cette charge de travail.

> **Notes**
>
> Il est possible que pour certains des éléments de l'interface utilisateur de Visual Studio, votre ordinateur affiche des noms ou des emplacements différents de ceux indiqués dans les instructions suivantes. L'édition de Visual Studio dont vous disposez et les paramètres que vous utilisez déterminent ces éléments. Pour plus d'informations, consultez **Personnalisation de l'IDE**.

- Pour la deuxième procédure pas à pas, installez IronPython pour .NET. Accédez à la page de téléchargement pour obtenir la version la plus récente.

Créer un objet dynamique personnalisé

La première procédure pas à pas définit un objet dynamique personnalisé qui recherche le contenu d'un fichier texte. Une propriété dynamique spécifie le texte à rechercher. Par exemple, si le code appelant spécifie `dynamicFile.Sample`, la classe dynamique retourne une liste générique de chaînes qui contient toutes les lignes du fichier qui commencent par « Sample ». La recherche respecte la casse. La classe dynamique prend également en charge deux arguments facultatifs. Le premier argument est une valeur enum de l'option de recherche qui spécifie que la classe dynamique doit rechercher des correspondances en début de ligne, en fin de ligne, ou n'importe où dans la ligne. Le deuxième argument spécifie que la classe dynamique doit supprimer des espaces à gauche et à droite dans chaque ligne avant d'effectuer la recherche. Par exemple, si le code appelant spécifie `dynamicFile.Sample(StringSearchOption.Contains)`, la classe dynamique recherche le terme « Sample » n'importe où dans une ligne. Si le code appelant spécifie `dynamicFile.Sample(StringSearchOption.StartsWith, false)`, la classe dynamique recherche le terme « Sample » au début de chaque ligne et ne supprime ni les espaces à droite, ni les espaces à gauche. Le comportement par défaut de la classe dynamique consiste à rechercher une correspondance au début de chaque ligne et à supprimer les espaces à gauche et à droite.

Créer une classe dynamique personnalisée

Démarrez Visual Studio. Sélectionnez **Créer un projet**. Dans la boîte de dialogue **Créer un projet**, sélectionnez C#, **Application console**, puis **Suivant**. Dans la boîte de dialogue **Configurer votre nouveau projet**, entrez `DynamicSample` pour le **Nom de projet**, puis sélectionnez **Suivant**. Dans la boîte de dialogue **Informations supplémentaires**, sélectionnez **.NET 7.0 (Actif)** pour le **Framework cible**, puis sélectionnez **Créer**. Dans **l'Explorateur de solutions**faites un clic droit sur le projet DynamicSample, puis sélectionnez **Ajouter**>**une Classe**. Dans la zone **Nom**, tapez `ReadOnlyFile`, puis sélectionnez **Ajouter**. En haut du fichier *ReadOnlyFile.cs* ou *ReadOnlyFile.vb*, ajoutez le code suivant pour importer les espaces de noms System.IO et System.Dynamic.

C#

```
using System.IO;
using System.Dynamic;
```

L'objet dynamique personnalisé utilise un enum pour déterminer les critères de recherche. Avant l'instruction de classe, ajoutez la définition d'enum suivante.

C#

```
public enum StringSearchOption
{
    StartsWith,
    Contains,
    EndsWith
}
```

Mettez à jour l'instruction de classe de façon à hériter de la classe `DynamicObject`, comme indiqué dans l'exemple de code suivant.

C#

```
class ReadOnlyFile : DynamicObject
```

Ajoutez le code suivant à la classe `ReadOnlyFile` afin de définir un champ privé pour le chemin du fichier et un constructeur pour la classe `ReadOnlyFile`.

C#

```
// Store the path to the file and the initial line count value.
private string p_filePath;

// Public constructor. Verify that file exists and store the path in
// the private variable.
public ReadOnlyFile(string filePath)
```

```
{
    if (!File.Exists(filePath))
    {
        throw new Exception("File path does not exist.");
    }

    p_filePath = filePath;
}
```

1. Ajoutez la méthode `GetPropertyValue` suivante à la classe `ReadOnlyFile`. La méthode `GetPropertyValue` prend les critères de recherche comme paramètre d'entrée et retourne les lignes d'un fichier texte qui correspondent aux critères de recherche. Les méthodes dynamiques fournies par la classe `ReadOnlyFile` appellent la méthode `GetPropertyValue` pour récupérer leurs résultats respectifs.

C#

```
public List<string> GetPropertyValue(string propertyName,
                                      StringSearchOption StringSearchOption =
StringSearchOption.StartsWith,
                                      bool trimSpaces = true)
{
    StreamReader sr = null;
    List<string> results = new List<string>();
    string line = "";
    string testLine = "";

    try
    {
        sr = new StreamReader(p_filePath);

        while (!sr.EndOfStream)
        {
            line = sr.ReadLine();

            // Perform a case-insensitive search by using the specified
search options.
            testLine = line.ToUpper();
            if (trimSpaces) { testLine = testLine.Trim(); }

            switch (StringSearchOption)
            {
                case StringSearchOption.StartsWith:
                    if (testLine.StartsWith(propertyName.ToUpper())) {
results.Add(line); }
                    break;
                case StringSearchOption.Contains:
                    if (testLine.Contains(propertyName.ToUpper())) {
results.Add(line); }
                    break;
                case StringSearchOption.EndsWith:
```

```
                    if (testLine.EndsWith(propertyName.ToUpper())) {
results.Add(line); }
                    break;
                }
            }
        }
        catch
        {
            // Trap any exception that occurs in reading the file and return
null.
            results = null;
        }
        finally
        {
            if (sr != null) {sr.Close();}
        }

        return results;
    }
```

Après la méthode `GetPropertyValue`, ajoutez le code suivant pour substituer la méthode TryGetMember de la classe DynamicObject. La méthode TryGetMember est appelée quand un membre d'une classe dynamique est demandé et qu'aucun argument n'est spécifié. L'argument `binder` contient les informations relatives au membre référencé et l'argument `result` référence le résultat retourné pour le membre spécifié. La méthode TryGetMember retourne une valeur booléenne qui retourne `true` si le membre demandé existe ; sinon, elle retourne la valeur `false`.

C#

```
// Implement the TryGetMember method of the DynamicObject class for dynamic
member calls.
public override bool TryGetMember(GetMemberBinder binder,
                                  out object result)
{
    result = GetPropertyValue(binder.Name);
    return result == null ? false : true;
}
```

Après la méthode `TryGetMember`, ajoutez le code suivant pour substituer la méthode TryInvokeMember de la classe DynamicObject. La méthode TryInvokeMember est appelée quand un membre d'une classe dynamique est demandé avec des arguments. L'argument `binder` contient les informations relatives au membre référencé et l'argument `result` référence le résultat retourné pour le membre spécifié. L'argument `args` contient un tableau des arguments passés au membre. La méthode

TryInvokeMember retourne une valeur booléenne qui retourne `true` si le membre demandé existe ; sinon, elle retourne la valeur `false`.

La version personnalisée de la méthode `TryInvokeMember` s'attend à ce que le premier argument soit une valeur de l'enum `StringSearchOption` que vous avez défini dans une étape précédente. La méthode `TryInvokeMember` s'attend à ce que le deuxième argument soit une valeur booléenne. Si l'un des arguments ou les deux arguments sont des valeurs valides, ils sont passés à la méthode `GetPropertyValue` pour récupérer les résultats.

C#

```
// Implement the TryInvokeMember method of the DynamicObject class for
// dynamic member calls that have arguments.
public override bool TryInvokeMember(InvokeMemberBinder binder,
                                     object[] args,
                                     out object result)
{
    StringSearchOption StringSearchOption = StringSearchOption.StartsWith;
    bool trimSpaces = true;

    try
    {
        if (args.Length > 0) { StringSearchOption =
(StringSearchOption)args[0]; }
    }
    catch
    {
        throw new ArgumentException("StringSearchOption argument must be a
StringSearchOption enum value.");
    }

    try
    {
        if (args.Length > 1) { trimSpaces = (bool)args[1]; }
    }
    catch
    {
        throw new ArgumentException("trimSpaces argument must be a Boolean
value.");
    }

    result = GetPropertyValue(binder.Name, StringSearchOption, trimSpaces);

    return result == null ? false : true;
}
```

Enregistrez et fermez le fichier.

Créer un exemple de fichier texte

Dans **l'Explorateur de solutions**faites un clic droit sur le projet DynamicSample, puis sélectionnez **Ajouter**>un **Nouvel élément**. Dans le volet **Modèles installés**, sélectionnez **Général**, puis sélectionnez le modèle **Fichier texte**. Conservez le nom par défaut de *TextFile1.txt* dans la zone **Nom**, puis sélectionnez **Ajouter**. Copiez le texte suivant dans le fichier *TextFile1.txt*.

```text
List of customers and suppliers

Supplier: Lucerne Publishing (https://www.lucernepublishing.com/)
Customer: Preston, Chris
Customer: Hines, Patrick
Customer: Cameron, Maria
Supplier: Graphic Design Institute (https://www.graphicdesigninstitute.com/)
Supplier: Fabrikam, Inc. (https://www.fabrikam.com/)
Customer: Seubert, Roxanne
Supplier: Proseware, Inc. (http://www.proseware.com/)
Customer: Adolphi, Stephan
Customer: Koch, Paul
```

Enregistrez et fermez le fichier.

Créer un exemple d'application qui utilise l'objet dynamique personnalisé

Dans l'**Explorateur de solutions**, double-cliquez sur le fichier *Program.cs*. Ajoutez le code suivant à la procédure `Main` pour créer une instance de la classe `ReadOnlyFile` pour le fichier *TextFile1.txt*. Le code utilise la liaison tardive pour appeler des membres dynamiques et récupérer des lignes de texte qui contiennent la chaîne « Customer ».

```C#
dynamic rFile = new ReadOnlyFile(@"..\..\..\TextFile1.txt");
foreach (string line in rFile.Customer)
{
    Console.WriteLine(line);
}
Console.WriteLine("----------------------------");
foreach (string line in rFile.Customer(StringSearchOption.Contains, true))
{
    Console.WriteLine(line);
}
```

Enregistrez le fichier et appuyez sur `Ctrl+F5` pour générer et exécuter l'application.

Appeler une bibliothèque de langage dynamique

La procédure pas-à-pas suivante crée un projet qui accède à une bibliothèque écrite dans le langage dynamique IronPython.

Pour créer une classe dynamique personnalisée

Dans Visual Studio, sélectionnez **Fichier**>**Nouveau**>**Projet**. Dans la boîte de dialogue **Créer un projet**, sélectionnez C#, **Application console**, puis **Suivant**. Dans la boîte de dialogue **Configurer votre nouveau projet**, entrez `DynamicIronPythonSample` pour le **Nom de projet**, puis sélectionnez **Suivant**. Dans la boîte de dialogue **Informations supplémentaires**, sélectionnez **.NET 7.0 (Actif)** pour le **Framework cible**, puis sélectionnez **Créer**. Installez le package NuGet IronPython. Modifiez le fichier *Program.cs*. En haut du fichier, ajoutez le code suivant pour importer les espaces de noms `Microsoft.Scripting.Hosting` et `IronPython.Hosting` à partir des bibliothèques IronPython et de l'espace de noms `System.Linq`.

C#

```
using System.Linq;
using Microsoft.Scripting.Hosting;
using IronPython.Hosting;
```

Dans la méthode Main, ajoutez le code suivant pour créer un objet `Microsoft.Scripting.Hosting.ScriptRuntime` pour héberger les bibliothèques IronPython. L'objet `ScriptRuntime` charge le module de bibliothèque IronPython random.py.

C#

```
// Set the current directory to the IronPython libraries.
System.IO.Directory.SetCurrentDirectory(
   Environment.GetFolderPath(Environment.SpecialFolder.ProgramFiles) +
   @"\IronPython 2.7\Lib");

// Create an instance of the random.py IronPython library.
Console.WriteLine("Loading random.py");
ScriptRuntime py = Python.CreateRuntime();
```

```
dynamic random = py.UseFile("random.py");
Console.WriteLine("random.py loaded.");
```

Une fois que le code a chargé le module random.py, ajoutez le code suivant pour créer un tableau d'entiers. Le tableau est passé à la méthode `shuffle` du module random.py, qui trie aléatoirement les valeurs dans le tableau.

C#

```
// Initialize an enumerable set of integers.
int[] items = Enumerable.Range(1, 7).ToArray();

// Randomly shuffle the array of integers by using IronPython.
for (int i = 0; i < 5; i++)
{
    random.shuffle(items);
    foreach (int item in items)
    {
        Console.WriteLine(item);
    }
    Console.WriteLine("--------------------");
}
```

Enregistrez le fichier et appuyez sur `Ctrl+F5` pour générer et exécuter l'application.

Voir aussi

- System.Dynamic
- System.Dynamic.DynamicObject
- Utilisation du type dynamic
- dynamic
- Implémenter des interfaces dynamiques (PDF téléchargeable sur Microsoft TechNet)

Versioning avec les mots clés override et new (Guide de programmation C#)

Article • 07/04/2023

Le langage C# est conçu de telle sorte que la gestion de version entre les classes de base et les classes dérivées dans différentes bibliothèques puisse évoluer et préserver une compatibilité descendante. Cela signifie, par exemple, que l'introduction dans une classe de base d'un nouveau membre portant le même nom qu'un membre dans une classe dérivée est totalement prise en charge par C# et n'engendre pas de comportement imprévisible. Cela signifie également qu'une classe doit indiquer de façon explicite si une méthode est conçue pour se substituer à une méthode héritée ou s'il s'agit d'une nouvelle méthode qui masque une méthode héritée portant un nom similaire.

Dans C#, les classes dérivées peuvent contenir des méthodes portant le même nom que des méthodes de classe de base.

- Si la méthode dans la classe dérivée n'est pas précédée des mots clés new ou override, le compilateur émet un avertissement et la méthode se comporte comme si le mot clé `new` était présent.
- Si la méthode dans la classe dérivée est précédée du mot clé `new`, la méthode est définie comme étant indépendante de la méthode dans la classe de base.
- Si la méthode dans la classe dérivée est précédée du mot clé `override`, les objets de la classe dérivée appelleront cette méthode plutôt que la méthode de la classe de base.
- Pour appliquer le mot clé `override` à la méthode de la classe dérivée, la méthode de la classe de base doit être définie comme virtuelle.
- La méthode de la classe de base peut être appelée à partir de la classe dérivée à l'aide du mot clé `base`.
- Les mots clés `override`, `virtual` et `new` peuvent également être appliqués aux propriétés, indexeurs et événements.

Par défaut, les méthodes C# ne sont pas virtuelles. Si une méthode est déclarée comme virtuelle, toute classe héritant de la méthode peut implémenter sa propre version. Pour créer une méthode virtuelle, le modificateur `virtual` est utilisé dans la déclaration de méthode de la classe de base. La classe dérivée peut ensuite remplacer la méthode

virtuelle de base avec le mot clé `override` ou masquer la méthode virtuelle dans la classe de base avec le mot clé `new`. Si ni le mot clé `override` ni le mot clé `new` ne sont spécifiés, le compilateur émettra un avertissement et la méthode dans la classe dérivée masquera la méthode dans la classe de base.

Pour illustrer ceci dans un exemple pratique, supposons un instant que la Société A ait créé une classe nommée `GraphicsClass` que votre programme utilise. Les éléments suivants représentent `GraphicsClass` :

C#

```
class GraphicsClass
{
    public virtual void DrawLine() { }
    public virtual void DrawPoint() { }
}
```

Votre société utilise cette classe, et vous l'utilisez pour dériver votre propre classe, en ajoutant une nouvelle méthode :

C#

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public void DrawRectangle() { }
}
```

Votre application est utilisée sans problèmes, jusqu'à ce que la Société A publie une nouvelle version de `GraphicsClass`, qui ressemble au code suivant :

C#

```
class GraphicsClass
{
    public virtual void DrawLine() { }
    public virtual void DrawPoint() { }
    public virtual void DrawRectangle() { }
}
```

La nouvelle version de `GraphicsClass` contient maintenant une méthode nommée `DrawRectangle`. Initialement, rien ne se produit. La nouvelle version est encore compatible au format binaire avec l'ancienne version. Tous les logiciels que vous avez déployés continueront de fonctionner, même si la nouvelle classe est installée sur ces systèmes informatiques. Tout appel existant à la méthode `DrawRectangle` continuera de faire référence à votre version, dans votre classe dérivée.

Toutefois, dès que vous recompilerez votre application à l'aide de la nouvelle version de `GraphicsClass`, vous recevrez un avertissement du compilateur, CS0108. Cet avertissement vous informe que vous devez envisager la façon dont vous souhaitez que votre méthode `DrawRectangle` se comporte dans votre application.

Si vous souhaitez que votre méthode remplace la nouvelle méthode de la classe de base, utilisez le mot clé `override` :

C#

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public override void DrawRectangle() { }
}
```

Le mot clé `override` garantit que tous les objets dérivés de `YourDerivedGraphicsClass` utiliseront la version de classe dérivée de `DrawRectangle`. Les objets dérivés de `YourDerivedGraphicsClass` peuvent toujours accéder à la version de la classe de base de `DrawRectangle` à l'aide du mot clé base :

C#

```
base.DrawRectangle();
```

Si vous ne souhaitez pas que votre méthode se substitue à la nouvelle méthode de la classe de base, prenez en compte les considérations suivantes. Pour éviter toute confusion entre les deux méthodes, vous pouvez renommer votre méthode. Ceci peut prendre du temps et constituer un risque d'erreur ou bien ne pas être tout simplement pratique dans certains cas. Toutefois, si votre projet est relativement petit, vous pouvez utiliser les options de refactorisation de Visual Studio pour renommer la méthode. Pour plus d'informations, consultez Refactorisation des classes et des types (Concepteur de classes).

Sinon, vous pouvez éviter de recevoir l'avertissement à l'aide du mot clé `new` dans la définition de votre classe dérivée :

C#

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public new void DrawRectangle() { }
}
```

L'utilisation du mot clé `new` indique au compilateur que votre définition masque la définition qui est contenue dans la classe de base. Il s'agit du comportement par défaut.

Substitution et sélection de méthode

Lorsqu'une méthode est appelée sur une classe, le compilateur C# sélectionne la meilleure méthode à appeler si plusieurs méthodes sont compatibles avec l'appel, comme lorsque deux méthodes portent les mêmes noms et dont les paramètres sont compatibles avec les paramètres transmis. Les méthodes suivantes seraient compatibles :

```
C#

public class Derived : Base
{
    public override void DoWork(int param) { }
    public void DoWork(double param) { }
}
```

Quand `DoWork` est appelée sur une instance de `Derived`, le compilateur C# essaie d'abord de rendre l'appel compatible avec les versions de `DoWork` déclarées à l'origine sur `Derived`. Les méthodes override ne sont pas considérées comme déclarées sur une classe ; il s'agit de nouvelles implémentations d'une méthode déclarée sur une classe de base. Le compilateur C# n'essaiera de faire correspondre l'appel à une méthode substituée portant le même nom et dotée de paramètres compatibles que s'il ne parvient pas à faire correspondre l'appel de méthode à une méthode d'origine sur `Derived`. Par exemple :

```
C#

int val = 5;
Derived d = new Derived();
d.DoWork(val);  // Calls DoWork(double).
```

Étant donné que la variable `val` peut être convertie implicitement en un double, le compilateur C# appelle `DoWork(double)` au lieu de `DoWork(int)`. Il existe deux façons d'éviter ceci. Premièrement, évitez de déclarer de nouvelles méthodes portant le même nom que des méthodes virtuelles. Deuxièmement, vous pouvez indiquer au compilateur C# d'appeler la méthode virtuelle en le faisant chercher dans la liste de méthodes de la classe de base par casting de l'instance `Derived` en `Base`. Étant donné que la méthode est virtuelle, l'implémentation de `DoWork(int)` sur `Derived` sera appelée. Par exemple :

C#

```
((Base)d).DoWork(val);  // Calls DoWork(int) on Derived.
```

Pour obtenir d'autres exemples des mots clés `new` et `override`, consultez Savoir quand utiliser les mots clés override et new.

Voir aussi

- Guide de programmation C#
- Système de types C#
- Méthodes
- Héritage

Savoir quand utiliser les mots clés override et new (Guide de programmation C#)

Article • 07/04/2023

En C#, une méthode dans une classe dérivée peut avoir le même nom qu'une méthode dans la classe de base. Vous pouvez spécifier le mode d'interaction des méthodes avec les mots clés new et override. Le modificateur `override` *étend* la méthode `virtual` de la classe de base, tandis que le modificateur `new` *masque* une méthode de classe de base accessible. La différence est illustrée dans les exemples de cette rubrique.

Dans une application console, déclarez les deux classes suivantes, `BaseClass` et `DerivedClass`. `DerivedClass` hérite de `BaseClass`.

C#

```
class BaseClass
{
    public void Method1()
    {
        Console.WriteLine("Base - Method1");
    }
}

class DerivedClass : BaseClass
{
    public void Method2()
    {
        Console.WriteLine("Derived - Method2");
    }
}
```

Dans la méthode `Main`, déclarez les variables `bc`, `dc` et `bcdc`.

- `bc` est de type `BaseClass` et sa valeur est de type `BaseClass`.
- `dc` est de type `DerivedClass` et sa valeur est de type `DerivedClass`.
- `bcdc` est de type `BaseClass` et sa valeur est de type `DerivedClass`. Il s'agit de la variable à surveiller.

Comme `bc` et `bcdc` ont le type `BaseClass`, ils ne peuvent accéder directement qu'à `Method1`, sauf si vous effectuez un cast. La variable `dc` peut accéder à `Method1` et

`Method2`. Ces relations sont illustrées dans le code suivant.

C#

```
class Program
{
    static void Main(string[] args)
    {
        BaseClass bc = new BaseClass();
        DerivedClass dc = new DerivedClass();
        BaseClass bcdc = new DerivedClass();

        bc.Method1();
        dc.Method1();
        dc.Method2();
        bcdc.Method1();
    }
    // Output:
    // Base - Method1
    // Base - Method1
    // Derived - Method2
    // Base - Method1
}
```

Ensuite, ajoutez la méthode `Method2` suivante à `BaseClass`. La signature de cette méthode correspond à celle de la méthode `Method2` dans `DerivedClass`.

C#

```
public void Method2()
{
    Console.WriteLine("Base - Method2");
}
```

Comme `BaseClass` a maintenant une méthode `Method2`, une deuxième instruction d'appel peut être ajoutée pour les variables `BaseClass` `bc` et `bcdc`, comme illustré dans le code suivant.

C#

```
bc.Method1();
bc.Method2();
dc.Method1();
dc.Method2();
bcdc.Method1();
bcdc.Method2();
```

Quand vous générez le projet, vous constatez que l'ajout de la méthode `Method2` dans `BaseClass` génère un avertissement. L'avertissement indique que la méthode `Method2` dans `DerivedClass` masque la méthode `Method2` dans `BaseClass`. Il est recommandé d'utiliser le mot clé `new` dans la définition de `Method2` si vous avez l'intention de provoquer ce résultat. Sinon, vous pouvez renommer l'une des méthodes `Method2` pour résoudre l'avertissement, mais ce n'est pas toujours pratique.

Avant d'ajouter `new`, exécutez le programme pour afficher la sortie produite par les instructions d'appel supplémentaires. Les résultats suivants s'affichent.

C#

```
// Output:
// Base - Method1
// Base - Method2
// Base - Method1
// Derived - Method2
// Base - Method1
// Base - Method2
```

Le mot clé `new` conserve les relations qui génèrent cette sortie, mais supprime l'avertissement. Les variables de type `BaseClass` continuent d'accéder aux membres de `BaseClass` et la variable de type `DerivedClass` continue d'accéder aux membres dans `DerivedClass` avant de prendre en compte les membres hérités de `BaseClass`.

Pour supprimer l'avertissement, ajoutez le modificateur `new` à la définition de `Method2` dans `DerivedClass`, comme illustré dans le code suivant. Le modificateur peut être ajouté avant ou après `public`.

C#

```
public new void Method2()
{
    Console.WriteLine("Derived - Method2");
}
```

Réexécutez le programme pour vérifier que la sortie n'a pas changé. Vérifiez aussi que l'avertissement ne s'affiche plus. À l'aide de `new`, vous déclarez que vous êtes conscient que le membre qu'il modifie masque un membre hérité de la classe de base. Pour plus d'informations sur le masquage de nom par héritage, consultez new, modificateur.

Pour comparer ce comportement aux effets de l'utilisation de `override`, ajoutez la méthode suivante à `DerivedClass`. Le modificateur `override` peut être ajouté avant ou

après `public`.

```
C#

public override void Method1()
{
    Console.WriteLine("Derived - Method1");
}
```

Ajoutez le modificateur `virtual` à la définition de `Method1` dans `BaseClass`. Le modificateur `virtual` peut être ajouté avant ou après `public`.

```
C#

public virtual void Method1()
{
    Console.WriteLine("Base - Method1");
}
```

Réexécutez le projet. Notez en particulier les deux dernières lignes de la sortie suivante.

```
C#

// Output:
// Base - Method1
// Base - Method2
// Derived - Method1
// Derived - Method2
// Derived - Method1
// Base - Method2
```

L'utilisation du modificateur `override` permet à `bcdc` d'accéder à la méthode `Method1` définie dans `DerivedClass`. En règle générale, il s'agit du comportement souhaité dans les hiérarchies d'héritage. Vous voulez que les objets dont les valeurs sont créées à partir de la classe dérivée utilisent les méthodes définies dans la classe dérivée. Vous obtenez ce comportement en utilisant `override` pour étendre la méthode de classe de base.

Le code suivant contient l'exemple complet.

```
C#

using System;
using System.Text;

namespace OverrideAndNew
{
    class Program
```

```
{
    static void Main(string[] args)
    {
        BaseClass bc = new BaseClass();
        DerivedClass dc = new DerivedClass();
        BaseClass bcdc = new DerivedClass();

        // The following two calls do what you would expect. They call
        // the methods that are defined in BaseClass.
        bc.Method1();
        bc.Method2();
        // Output:
        // Base - Method1
        // Base - Method2

        // The following two calls do what you would expect. They call
        // the methods that are defined in DerivedClass.
        dc.Method1();
        dc.Method2();
        // Output:
        // Derived - Method1
        // Derived - Method2

        // The following two calls produce different results, depending
        // on whether override (Method1) or new (Method2) is used.
        bcdc.Method1();
        bcdc.Method2();
        // Output:
        // Derived - Method1
        // Base - Method2
    }
}

class BaseClass
{
    public virtual void Method1()
    {
        Console.WriteLine("Base - Method1");
    }

    public virtual void Method2()
    {
        Console.WriteLine("Base - Method2");
    }
}

class DerivedClass : BaseClass
{
    public override void Method1()
    {
        Console.WriteLine("Derived - Method1");
    }

    public new void Method2()
    {
```

```
            Console.WriteLine("Derived - Method2");
        }
    }
}
```

L'exemple suivant illustre un comportement similaire dans un autre contexte. L'exemple définit trois classes : une classe de base nommée `Car` et deux de ses classes dérivées, `ConvertibleCar` et `Minivan`. La classe de base contient une méthode `DescribeCar`. La méthode affiche une description de base d'une voiture et appelle ensuite `ShowDetails` pour fournir des informations supplémentaires. Chacune des trois classes définit une méthode `ShowDetails`. Le modificateur `new` est utilisé pour définir `ShowDetails` dans la classe `ConvertibleCar`. Le modificateur `override` est utilisé pour définir `ShowDetails` dans la classe `Minivan`.

C#

```
// Define the base class, Car. The class defines two methods,
// DescribeCar and ShowDetails. DescribeCar calls ShowDetails, and each
derived
// class also defines a ShowDetails method. The example tests which version
of
// ShowDetails is selected, the base class method or the derived class
method.
class Car
{
    public void DescribeCar()
    {
        System.Console.WriteLine("Four wheels and an engine.");
        ShowDetails();
    }

    public virtual void ShowDetails()
    {
        System.Console.WriteLine("Standard transportation.");
    }
}

// Define the derived classes.

// Class ConvertibleCar uses the new modifier to acknowledge that
ShowDetails
// hides the base class method.
class ConvertibleCar : Car
{
    public new void ShowDetails()
    {
        System.Console.WriteLine("A roof that opens up.");
    }
}
```

```
// Class Minivan uses the override modifier to specify that ShowDetails
// extends the base class method.
class Minivan : Car
{
    public override void ShowDetails()
    {
        System.Console.WriteLine("Carries seven people.");
    }
}
```

L'exemple vérifie la version de `ShowDetails` qui est appelée. La méthode suivante, `TestCars1`, déclare une instance de chaque classe, puis appelle `DescribeCar` sur chaque instance.

C#

```
public static void TestCars1()
{
    System.Console.WriteLine("\nTestCars1");
    System.Console.WriteLine("----------");

    Car car1 = new Car();
    car1.DescribeCar();
    System.Console.WriteLine("----------");

    // Notice the output from this test case. The new modifier is
    // used in the definition of ShowDetails in the ConvertibleCar
    // class.

    ConvertibleCar car2 = new ConvertibleCar();
    car2.DescribeCar();
    System.Console.WriteLine("----------");

    Minivan car3 = new Minivan();
    car3.DescribeCar();
    System.Console.WriteLine("----------");
}
```

`TestCars1` génère la sortie suivante. Notez en particulier les résultats de `car2`, qui ne correspondent probablement pas à ce que vous attendiez. Le type de l'objet est `ConvertibleCar`, mais `DescribeCar` n'accède pas à la version de `ShowDetails` qui est défini dans la classe `ConvertibleCar`, car cette méthode est déclarée avec le modificateur `new`, et non `override`. Par conséquent, un objet `ConvertibleCar` affiche la même description qu'un objet `Car`. Comparez les résultats pour `car3`, qui est un objet `Minivan`. Dans ce cas, la méthode `ShowDetails` déclarée dans la classe `Minivan` substitue la méthode `ShowDetails` déclarée dans la classe `Car` et la description affichée décrit un minivan.

C#

```
// TestCars1
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Carries seven people.
// ----------
```

`TestCars2` crée une liste d'objets de type `Car`. Les valeurs des objets sont instanciées à partir des classes `Car`, `ConvertibleCar` et `Minivan`. `DescribeCar` est appelé sur chaque élément de la liste. Le code suivant illustre la définition de `TestCars2`.

C#

```
public static void TestCars2()
{
    System.Console.WriteLine("\nTestCars2");
    System.Console.WriteLine("----------");

    var cars = new List<Car> { new Car(), new ConvertibleCar(),
        new Minivan() };

    foreach (var car in cars)
    {
        car.DescribeCar();
        System.Console.WriteLine("----------");
    }
}
```

La sortie suivante s'affiche. Notez qu'elle est identique à la sortie affichée par `TestCars1`. La méthode `ShowDetails` de la classe `ConvertibleCar` n'est pas appelée, que le type de l'objet soit `ConvertibleCar`, comme dans `TestCars1`, ou `Car`, comme dans `TestCars2`. À l'inverse, `car3` appelle la méthode `ShowDetails` à partir de la classe `Minivan` dans les deux cas, que son type soit `Minivan` ou `Car`.

C#

```
// TestCars2
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
```

```
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Carries seven people.
// ----------
```

Les méthodes `TestCars3` et `TestCars4` terminent l'exemple. Ces méthodes appellent `ShowDetails` directement, d'abord à partir des objets déclarés avec le type `ConvertibleCar` et `Minivan` (`TestCars3`), puis à partir des objets déclarés avec le type `Car` (`TestCars4`). Le code suivant définit ces deux méthodes.

C#

```
public static void TestCars3()
{
    System.Console.WriteLine("\nTestCars3");
    System.Console.WriteLine("----------");
    ConvertibleCar car2 = new ConvertibleCar();
    Minivan car3 = new Minivan();
    car2.ShowDetails();
    car3.ShowDetails();
}

public static void TestCars4()
{
    System.Console.WriteLine("\nTestCars4");
    System.Console.WriteLine("----------");
    Car car2 = new ConvertibleCar();
    Car car3 = new Minivan();
    car2.ShowDetails();
    car3.ShowDetails();
}
```

Les méthodes produisent la sortie suivante, qui correspond aux résultats du premier exemple de cette rubrique.

C#

```
// TestCars3
// ----------
// A roof that opens up.
// Carries seven people.

// TestCars4
// ----------
// Standard transportation.
// Carries seven people.
```

Le code suivant illustre le projet complet et sa sortie.

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace OverrideAndNew2
{
    class Program
    {
        static void Main(string[] args)
        {
            // Declare objects of the derived classes and test which version
            // of ShowDetails is run, base or derived.
            TestCars1();

            // Declare objects of the base class, instantiated with the
            // derived classes, and repeat the tests.
            TestCars2();

            // Declare objects of the derived classes and call ShowDetails
            // directly.
            TestCars3();

            // Declare objects of the base class, instantiated with the
            // derived classes, and repeat the tests.
            TestCars4();
        }

        public static void TestCars1()
        {
            System.Console.WriteLine("\nTestCars1");
            System.Console.WriteLine("----------");

            Car car1 = new Car();
            car1.DescribeCar();
            System.Console.WriteLine("----------");

            // Notice the output from this test case. The new modifier is
            // used in the definition of ShowDetails in the ConvertibleCar
            // class.
            ConvertibleCar car2 = new ConvertibleCar();
            car2.DescribeCar();
            System.Console.WriteLine("----------");

            Minivan car3 = new Minivan();
            car3.DescribeCar();
            System.Console.WriteLine("----------");
        }
        // Output:
        // TestCars1
```

```
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Carries seven people.
        // ----------

        public static void TestCars2()
        {
            System.Console.WriteLine("\nTestCars2");
            System.Console.WriteLine("----------");

            var cars = new List<Car> { new Car(), new ConvertibleCar(),
                new Minivan() };

            foreach (var car in cars)
            {
                car.DescribeCar();
                System.Console.WriteLine("----------");
            }
        }
        // Output:
        // TestCars2
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Carries seven people.
        // ----------

        public static void TestCars3()
        {
            System.Console.WriteLine("\nTestCars3");
            System.Console.WriteLine("----------");
            ConvertibleCar car2 = new ConvertibleCar();
            Minivan car3 = new Minivan();
            car2.ShowDetails();
            car3.ShowDetails();
        }
        // Output:
        // TestCars3
        // ----------
        // A roof that opens up.
        // Carries seven people.

        public static void TestCars4()
        {
```

```
            System.Console.WriteLine("\nTestCars4");
            System.Console.WriteLine("----------");
            Car car2 = new ConvertibleCar();
            Car car3 = new Minivan();
            car2.ShowDetails();
            car3.ShowDetails();
        }
        // Output:
        // TestCars4
        // ----------
        // Standard transportation.
        // Carries seven people.
    }

    // Define the base class, Car. The class defines two virtual methods,
    // DescribeCar and ShowDetails. DescribeCar calls ShowDetails, and each
derived
    // class also defines a ShowDetails method. The example tests which
version of
    // ShowDetails is used, the base class method or the derived class
method.
    class Car
    {
        public virtual void DescribeCar()
        {
            System.Console.WriteLine("Four wheels and an engine.");
            ShowDetails();
        }

        public virtual void ShowDetails()
        {
            System.Console.WriteLine("Standard transportation.");
        }
    }

    // Define the derived classes.

    // Class ConvertibleCar uses the new modifier to acknowledge that
ShowDetails
    // hides the base class method.
    class ConvertibleCar : Car
    {
        public new void ShowDetails()
        {
            System.Console.WriteLine("A roof that opens up.");
        }
    }

    // Class Minivan uses the override modifier to specify that ShowDetails
    // extends the base class method.
    class Minivan : Car
    {
        public override void ShowDetails()
        {
            System.Console.WriteLine("Carries seven people.");
```

```
        }
    }

}
```

Voir aussi

- Guide de programmation C#
- Système de types C#
- Versioning avec les mots clés override et new
- base
- abstract

Guide pratique pour substituer la méthode ToString (Guide de programmation C#)

Article • 07/04/2023

En C#, chaque classe ou struct hérite implicitement de la classe Object. Ainsi, chaque objet en C# obtient la méthode ToString, qui retourne une représentation sous forme de chaîne de cet objet. Par exemple, toutes les variables de type `int` ont une méthode `ToString`, ce qui leur permet de retourner leur contenu sous forme de chaîne :

C#

```
int x = 42;
string strx = x.ToString();
Console.WriteLine(strx);
// Output:
// 42
```

Quand vous créez une classe ou un struct personnalisé, vous devez substituer la méthode ToString pour fournir des informations sur votre type au code client.

Pour plus d'informations sur l'utilisation de chaînes de format et d'autres types de mise en forme personnalisée avec la méthode `ToString`, consultez Mise en forme des types.

> **ⓘ Important**
>
> Quand vous choisissez les informations à fournir par l'intermédiaire de cette méthode, déterminez si votre classe ou struct sera utilisé par du code non fiable. Veillez à ne pas fournir d'informations susceptibles d'être exploitées par du code malveillant.

Pour substituer la méthode `ToString` dans votre classe ou struct :

1. Déclarez une méthode `ToString` avec les modificateurs et le type de retour suivants :

   C#

   ```
   public override string ToString(){}
   ```

2. Implémentez la méthode pour qu'elle retourne une chaîne.

   L'exemple suivant retourne le nom de la classe en plus des données propres à une instance particulière de la classe.

```C#
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public override string ToString()
    {
        return "Person: " + Name + " " + Age;
    }
}
```

   Vous pouvez tester la méthode `ToString` comme illustré dans l'exemple de code suivant :

```C#
Person person = new Person { Name = "John", Age = 12 };
Console.WriteLine(person);
// Output:
// Person: John 12
```

Voir aussi

- IFormattable
- Guide de programmation C#
- Système de types C#
- Chaînes
- string
- override
- virtual
- Mise en forme des types

Membres (Guide de programmation C#)

Article • 07/04/2023

Les classes et les structs ont des membres qui représentent leurs données et leur comportement. Les membres d'une classe incluent tous les membres déclarés de la classe, ainsi que tous les membres (à l'exception des constructeurs et des finaliseurs) déclarés dans toutes les classes de sa hiérarchie d'héritage. Les membres privés dans les classes de base sont hérités, mais ne sont pas accessibles à partir des classes dérivées.

Le tableau suivant liste les types de membres qu'une classe ou un struct peut contenir :

Membre	Description
Fields	Les champs sont des variables déclarées au niveau de la classe. Un champ peut être un type numérique intégré ou une instance d'une autre classe. Par exemple, une classe de calendrier peut avoir un champ qui contient la date actuelle.
Constantes	Les constantes sont des champs dont la valeur est définie au moment de la compilation et ne peut pas être modifiée.
Propriétés	Les propriétés sont des méthodes sur une classe, accessibles comme si elles étaient des champs de cette classe. Une propriété peut fournir une protection pour un champ de classe afin d'éviter qu'il ne soit modifié sans que de l'objet en ait connaissance.
Méthodes	Les méthodes définissent les actions qu'une classe peut effectuer. Les méthodes peuvent accepter des paramètres qui fournissent des données d'entrée et peuvent retourner des données de sortie au moyen de paramètres. Les méthodes peuvent également retourner une valeur directement, sans utiliser de paramètre.
Événements	Les événements fournissent des notifications d'occurrences, comme des clics de bouton ou l'exécution réussie d'une méthode, sur d'autres objets. Les événements sont définis et déclenchés à l'aide de délégués.
Opérateurs	Les opérateurs surchargés sont considérés comme des membres de type. Quand vous surchargez un opérateur, vous le définissez sous forme de méthode statique publique dans un type. Pour plus d'informations, consultez Surcharge d'opérateur.
Indexeurs	Les indexeurs permettent à un objet d'être indexé d'une manière similaire aux tableaux.
Constructeurs	Les constructeurs sont des méthodes qui sont appelées quand l'objet est créé. Ils sont souvent utilisés pour initialiser les données d'un objet.

Membre	Description
Finaliseurs	Les finaliseurs sont très rarement utilisés en C#. Ce sont des méthodes appelées par le moteur d'exécution quand l'objet est sur le point d'être supprimé de la mémoire. Ils sont généralement utilisés pour vérifier que toutes les ressources qui doivent être libérées sont gérées correctement.
Types imbriqués	Les types imbriqués sont des types déclarés dans un autre type. Les types imbriqués sont souvent utilisés pour décrire les objets utilisés uniquement par les types qui les contiennent.

Voir aussi

- Guide de programmation C#
- Classes

Classes abstract et sealed et membres de classe (Guide de programmation C#)

Article • 07/04/2023

Le mot clé abstract vous permet de créer des classes et des membres de class qui sont incomplets et doivent être implémentés dans une classe dérivée.

Le mot clé sealed vous permet d'empêcher l'héritage d'une classe ou de certains membres de classe qui étaient précédemment marqués comme virtual.

Classes abstraites et membres de classe

Les classes peuvent être déclarées comme abstraites en plaçant le mot clé `abstract` avant la définition de classe. Par exemple :

C#

```
public abstract class A
{
    // Class members here.
}
```

Une classe abstraite ne peut pas être instanciée. Le but d'une classe abstraite est de fournir une définition commune d'une classe de base pouvant être partagée par plusieurs classes dérivées. Par exemple, une bibliothèque de classes peut définir une classe abstraite qui est utilisée comme un paramètre pour beaucoup de ses fonctions, et exiger des programmeurs qui l'utilisent de fournir leur propre implémentation de la classe en créant une classe dérivée.

Les classes abstraites peuvent également définir des méthodes abstraites. Pour ce faire, ajoutez le mot clé `abstract` avant le type de retour de la méthode. Par exemple :

C#

```
public abstract class A
{
    public abstract void DoWork(int i);
}
```

Les méthodes abstraites n'ont pas d'implémentation. Ainsi, la définition de la méthode est suivie d'un point-virgule au lieu d'un bloc de méthode normal. Les classes dérivées

de la classe abstraite doivent implémenter toutes les méthodes abstraites. Lorsqu'une classe abstraite hérite d'une méthode virtuelle d'une classe de base, la classe abstraite peut substituer la méthode virtuelle par une méthode abstraite. Par exemple :

C#

```
// compile with: -target:library
public class D
{
    public virtual void DoWork(int i)
    {
        // Original implementation.
    }
}

public abstract class E : D
{
    public abstract override void DoWork(int i);
}

public class F : E
{
    public override void DoWork(int i)
    {
        // New implementation.
    }
}
```

Si une méthode `virtual` est déclarée `abstract`, elle demeure virtuelle pour toutes les classes qui héritent de la classe abstraite. Une classe qui hérite d'une méthode abstraite ne peut pas accéder à l'implémentation d'origine de la méthode. Dans l'exemple précédent, `DoWork` sur la classe F ne peut pas appeler `DoWork` sur la classe D. De cette manière, une classe abstraite peut forcer les classes dérivées à fournir de nouvelles implémentations de méthode pour les méthodes virtuelles.

Classes sealed et membres de classe

Les classes peuvent être déclarées comme sealed en plaçant le mot clé `sealed` avant la définition de classe. Par exemple :

C#

```
public sealed class D
{
    // Class members here.
}
```

Une classe sealed ne peut pas être utilisée comme une classe de base. Pour cette raison, elle ne peut pas également être une classe abstraite. Les classes sealed empêchent la dérivation. Étant donné qu'elles ne peuvent pas être utilisées comme une classe de base, quelques optimisations au moment de l'exécution permettent d'accélérer un peu les appels aux membres des classes sealed.

Une méthode, un indexeur, une propriété ou un événement sur une classe dérivée qui substitue un membre virtuel de la classe de base peut déclarer ce membre comme sealed. Cela nie l'aspect virtuel du membre pour toute classe dérivée ultérieure. Pour ce faire, mettez le mot clé `sealed` avant le mot clé override dans la déclaration de membre de classe. Par exemple :

C#

```
public class D : C
{
    public sealed override void DoWork() { }
}
```

Voir aussi

- Guide de programmation C#
- Système de types C#
- Héritage
- Méthodes
- Fields
- Comment : définir des propriétés abstraites

Classes statiques et membres de classe statique (Guide de programmation C#)

Article • 16/03/2023

Une classe statique est fondamentalement identique à une classe non statique, à une différence près : une classe statique ne peut pas être instanciée. En d'autres termes, vous ne pouvez pas utiliser l'opérateur new pour créer une variable du type classe. Étant donné qu'il n'y a aucune variable d'instance, vous accédez aux membres d'une classe statique en utilisant le nom de classe lui-même. Par exemple, si vous avez une classe statique nommée `UtilityClass` qui a une méthode statique publique nommée `MethodA`, vous appelez la méthode comme illustré dans l'exemple suivant :

C#

```
UtilityClass.MethodA();
```

Une classe statique peut servir de conteneur pratique pour des ensembles de méthodes qui opèrent simplement sur des paramètres d'entrée et n'ont pas à obtenir ou définir de champs d'instance internes. Par exemple, dans la bibliothèque de classes .NET, la classe System.Math statique contient des méthodes qui effectuent des opérations mathématiques, sans qu'il soit nécessaire de stocker ou de récupérer des données spécifiques à une instance particulière de la classe Math. Autrement dit, vous appliquez les membres de la classe en spécifiant le nom de la classe et le nom de la méthode, comme illustré dans l'exemple suivant.

C#

```
double dub = -3.14;
Console.WriteLine(Math.Abs(dub));
Console.WriteLine(Math.Floor(dub));
Console.WriteLine(Math.Round(Math.Abs(dub)));

// Output:
// 3.14
// -4
// 3
```

Comme c'est le cas avec tous les types de classe, les informations de type pour une classe statique sont chargées par le runtime .NET lorsque le programme qui fait référence à la classe est chargé. Le programme ne peut pas spécifier exactement quand la classe est chargée. Toutefois, il est garanti qu'elle sera chargée, que ses champs seront initialisés et que son constructeur statique sera appelé avant que la classe soit

référencée pour la première fois dans votre programme. Un constructeur statique est appelé une seule fois et une classe statique reste en mémoire pendant la durée de vie du domaine d'application dans lequel votre programme réside.

> ⓘ **Notes**
>
> Pour créer une classe non statique permettant la création d'une seule instance d'elle-même, consultez **Implémentation d'un singleton en C#**.

La liste suivante fournit les fonctionnalités principales d'une classe statique :

- Elle contient uniquement des membres statiques.
- Elle ne peut pas être instanciée.
- Elle est verrouillée (sealed).
- Elle ne peut pas contenir de constructeurs d'instances.

La création d'une classe statique est par conséquent très semblable à la création d'une classe contenant uniquement des membres statiques et un constructeur privé. Un constructeur privé empêche la classe d'être instanciée. L'avantage de l'utilisation d'une classe statique est que le compilateur peut vérifier qu'aucun membre d'instance n'a été ajouté par erreur. Le compilateur garantit que les instances de cette classe ne peuvent pas être créées.

Les classes statiques sont scellées (sealed) et ne peuvent par conséquent pas être héritées. Elles ne peuvent hériter d'aucune classe ou interface, à l'exception d'Object. Les classes statiques ne peuvent pas contenir un constructeur d'instance. Toutefois, elles peuvent contenir un constructeur statique. Les classes non statiques doivent également définir un constructeur statique si la classe contient des membres statiques qui requièrent une initialisation non triviale. Pour plus d'informations, consultez Constructeurs statiques.

Exemple

Voici un exemple d'une classe statique qui contient deux méthodes qui convertissent la température des degrés Celsius en degrés Fahrenheit et des degrés Fahrenheit en degrés Celsius :

```
C#
```

```
public static class TemperatureConverter
{
    public static double CelsiusToFahrenheit(string temperatureCelsius)
    {
        // Convert argument to double for calculations.
        double celsius = Double.Parse(temperatureCelsius);

        // Convert Celsius to Fahrenheit.
        double fahrenheit = (celsius * 9 / 5) + 32;

        return fahrenheit;
    }

    public static double FahrenheitToCelsius(string temperatureFahrenheit)
    {
        // Convert argument to double for calculations.
        double fahrenheit = Double.Parse(temperatureFahrenheit);

        // Convert Fahrenheit to Celsius.
        double celsius = (fahrenheit - 32) * 5 / 9;

        return celsius;
    }
}

class TestTemperatureConverter
{
    static void Main()
    {
        Console.WriteLine("Please select the convertor direction");
        Console.WriteLine("1. From Celsius to Fahrenheit.");
        Console.WriteLine("2. From Fahrenheit to Celsius.");
        Console.Write(":");

        string? selection = Console.ReadLine();
        double F, C = 0;

        switch (selection)
        {
            case "1":
                Console.Write("Please enter the Celsius temperature: ");
                F =
TemperatureConverter.CelsiusToFahrenheit(Console.ReadLine() ?? "0");
                Console.WriteLine("Temperature in Fahrenheit: {0:F2}", F);
                break;

            case "2":
                Console.Write("Please enter the Fahrenheit temperature: ");
                C =
TemperatureConverter.FahrenheitToCelsius(Console.ReadLine() ?? "0");
                Console.WriteLine("Temperature in Celsius: {0:F2}", C);
                break;

            default:
```

```
                Console.WriteLine("Please select a convertor.");
                break;
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Example Output:
    Please select the convertor direction
    1. From Celsius to Fahrenheit.
    2. From Fahrenheit to Celsius.
    :2
    Please enter the Fahrenheit temperature: 20
    Temperature in Celsius: -6.67
    Press any key to exit.
 */
```

Membres static

Une classe non statique peut contenir des méthodes, des champs, des propriétés ou des événements statiques. Le membre statique peut être appelé sur une classe même quand aucune instance de la classe n'a été créée. Le membre statique est toujours accessible par le nom de la classe, et non par le nom de l'instance. Une seule copie d'un membre statique existe, quel que soit le nombre d'instances de la classe qui ont été créées. Les méthodes et propriétés statiques ne peuvent pas accéder à des champs ou des événements non statiques dans leur type contenant. Elles ne peuvent pas non plus accéder à une variable d'instance d'un objet quelconque à moins qu'elle soit passée explicitement dans un paramètre de méthode.

Il est plus courant de déclarer une classe non statique avec certains membres statiques que de déclarer une classe entière comme statique. Deux utilisations courantes des champs statiques consistent à conserver un décompte du nombre d'objets qui ont été instanciés ou à stocker une valeur qui doit être partagée entre toutes les instances.

Les méthodes statiques peuvent être surchargées mais pas substituées, car elles appartiennent à la classe et non pas à une instance de la classe.

Bien qu'un champ ne puisse pas être déclaré en tant que `static const`, un champ const est essentiellement statique dans son comportement. Il appartient au type, pas aux instances du type. Par conséquent, les champs `const` sont accessibles à l'aide de la même notation `ClassName.MemberName` utilisée pour les champs statiques. Aucune instance d'objet n'est requise.

C# ne prend pas en charge les variables locales statiques (c'est-à-dire, les variables déclarées dans la portée de la méthode).

Vous déclarez des membres de classe statique en utilisant le mot clé `static` avant le type de retour du membre, comme illustré dans l'exemple suivant :

C#

```
public class Automobile
{
    public static int NumberOfWheels = 4;

    public static int SizeOfGasTank
    {
        get
        {
            return 15;
        }
    }

    public static void Drive() { }

    public static event EventType? RunOutOfGas;

    // Other non-static fields and properties...
}
```

Les membres statiques sont initialisés avant le premier accès au membre statique et avant que le constructeur statique, s'il en existe un, soit appelé. Pour accéder à un membre de classe statique, utilisez le nom de la classe au lieu d'un nom de variable pour spécifier l'emplacement du membre, comme illustré dans l'exemple suivant :

C#

```
Automobile.Drive();
int i = Automobile.NumberOfWheels;
```

Si votre classe contient des champs statiques, fournissez un constructeur statique qui les initialise quand la classe est chargée.

Un appel à une méthode statique génère une instruction d'appel en langage MSIL (Microsoft Intermediate Language), alors qu'un appel à une méthode d'instance génère une instruction `callvirt`, qui vérifie également l'existence de références d'objet null. Toutefois, la plupart du temps, l'écart de performances entre les deux n'est pas significatif.

Spécification du langage C#

Pour plus d'informations, voir Classes statiques, Membres statiques et d'instance et Constructeurs statiques dans la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Guide de programmation C#
- static
- Classes
- class
- Constructeurs statiques
- Constructeurs d'instances

Modificateurs d'accès (Guide de programmation C#)

Article • 07/04/2023

Tous les types et les membres de type ont un niveau d'accessibilité. Le niveau d'accessibilité contrôle s'il peut être utilisé à partir d'un autre code dans votre assembly ou d'autres assemblies. Un assembly est un *fichier .dll* ou *.exe* créé en compilant un ou plusieurs *fichiers .cs* dans une seule compilation. Utilisez les modificateurs d'accès suivants pour spécifier l'accessibilité d'un type ou d'un membre lorsque vous le déclarez :

- public : n'importe quel autre code dans le même assembly ou un autre assembly qui le référence peut accéder à ce type ou à ce membre. Le niveau d'accessibilité des membres publics d'un type est contrôlé par le niveau d'accessibilité du type lui-même.
- private (privé) : seul du code dans la même `class` ou dans le même `struct` peut accéder au type ou au membre.
- protected (protégé) : seul du code dans la même `class` ou dans la `class` dérivée de cette `class` peut accéder au type ou au membre.
- internal (interne) : tout code du même assembly peut accéder au type ou au membre, mais l'accès est impossible dans le cas d'un code issu d'un autre assembly. En d'autres termes, les types ou les membres `internal` sont accessibles par du code faisant partie de la même compilation.
- protected internal (interne protégé) : tout code de l'assembly dans lequel le type ou le membre est déclaré ou d'une `class` dérivée dans un autre assembly peut accéder au type ou au membre.
- private protected(protégé privé) : les types dérivant de la `class` et qui sont déclarés dans l'assembly conteneur peuvent accéder au type ou au membre.

Tableau récapitulatif

Emplacement de l'appelant	`public`	`protected internal`	`protected`	`internal`	`private protected`	`private`
Dans la classe	✔	✔	✔	✔	✔	✔
Classe dérivée (même assembly)	✔	✔	✔	✔	✔	✘

Emplacement de l'appelant	`public`	`protected internal`	`protected`	`internal`	`private protected`	`private`
Classe non dérivée (même assembly)	✔	✔	✘	✔	✘	✘
Classe dérivée (assembly différent)	✔	✔	✔	✘	✘	✘
Classe non dérivée (assembly différent)	✔	✘	✘	✘	✘	✘

Les exemples suivants montrent comment spécifier des modificateurs d'accès sur un type et un membre :

C#

```
public class Bicycle
{
    public void Pedal() { }
}
```

Les modificateurs d'accès ne sont pas toujours valides pour tous les types ou membres dans tous les contextes. Dans certains cas, l'accessibilité d'un membre de type est limitée par l'accessibilité de son type conteneur.

Accessibilité des classes, des enregistrements et des structs

Les classes, les enregistrements et les structs qui sont déclarés directement dans un espace de noms (autrement dit, qui ne sont pas imbriqués dans d'autres classes ou structs) peuvent être déclarés comme `public` ou `internal`. `internal` est la valeur par défaut si aucun modificateur d'accès n'est spécifié.

Les membres de struct, y compris les classes et structs imbriqués, peuvent être déclarés `public`, `internal` ou `private`. Les membres de classe, y compris les classes et structs imbriqués, peuvent être `public`, `protected internal`, `protected`, `internal`, `private protected` ou `private`. Les membres de classe et de struct, y compris les classes et les structs imbriqués, ont un accès `private` par défaut. L'accès aux types imbriqués private n'est pas possible hors du type conteneur.

Les classes dérivées et les enregistrements dérivés ne peuvent pas avoir une meilleure accessibilité que leurs types de base. Vous ne pouvez pas déclarer publique une classe `B`

qui dérive d'une classe interne `A`. Si cela était autorisé, cela aurait pour effet de rendre `A` public, car tous les membres `protected` ou `internal` de `A` seraient accessibles à partir de la classe dérivée.

Vous pouvez autoriser d'autres assemblies spécifiques à accéder à vos types internes à l'aide d'`InternalsVisibleToAttribute`. Pour plus d'informations, consultez Assemblys friend.

Accessibilité des membres de classe, d'enregistrement et de struct

Vous pouvez déclarer les membres de classe (notamment les classes, les enregistrements et les structs imbriqués) avec l'un des six types d'accès. Les membres de struct ne peuvent pas être déclarés `protected`, `protected internal` ou `private protected` parce que les structs ne prennent pas en charge l'héritage.

Normalement, l'accessibilité d'un membre ne peut être supérieure à l'accessibilité du type qui le contient. Toutefois, un membre `public` d'une classe internal peut être accessible de l'extérieur de l'assembly si le membre implémente des méthodes d'interface ou substitue des méthodes virtuelles qui sont définies dans une classe de base public.

Le type d'un champ, d'une propriété ou d'un événement de membre doit être au moins aussi accessible que le membre lui-même. De même, le type de retour et les types de paramètre d'une méthode, d'un indexeur ou d'un délégué doivent être au moins aussi accessibles que le membre lui-même. Par exemple, vous ne pouvez pas avoir de méthode `public M` qui retourne une classe `C` à moins que `C` ne soit également `public`. Vous ne pouvez pas non plus avoir une propriété `protected` de type `A` si `A` est déclaré `private`.

Les opérateurs définis par l'utilisateur doivent toujours être déclarés `public` et `static`. Pour plus d'informations, consultez Surcharge d'opérateur.

Les finaliseurs ne peuvent pas avoir de modificateurs d'accessibilité.

Pour définir le niveau d'accès pour un membre `class`, `record` ou `struct`, ajoutez le mot clé approprié à la déclaration de membre, comme indiqué dans l'exemple suivant.

C#

```
// public class:
public class Tricycle
```

```
{
    // protected method:
    protected void Pedal() { }

    // private field:
    private int _wheels = 3;

    // protected internal property:
    protected internal int Wheels
    {
        get { return _wheels; }
    }
}
```

Autres types

Les interfaces qui sont déclarées directement dans un espace de noms peuvent être `public` ou `internal` et, tout comme les classes et les structs, les interfaces ont un niveau d'accès `internal` par défaut. Les membres d'interface sont déclarés comme `public` par défaut, car l'objectif d'une interface est de permettre à d'autres types d'accéder à une classe ou à un struct. Les déclarations de membre d'interface peuvent inclure n'importe quel modificateur d'accès. Cela est particulièrement utile pour les méthodes statiques afin de fournir les implémentations courantes nécessaires à tous les implémenteurs d'une classe.

Les membres de l'énumération sont toujours considérés comme `public`, et aucun modificateur d'accès ne peut être appliqué.

Les délégués se comportent comme des classes et des structs. Par défaut, ils ont un accès `internal` lorsqu'ils sont déclarés directement dans un espace de noms et `private` lorsqu'ils sont imbriqués.

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Spécifier l'ordre des modificateurs (règle de style IDE0036)
- Guide de programmation C#
- Système de types C#
- Interfaces

- private
- public
- internal
- protected
- protected internal
- protégé privé
- class
- struct
- interface

Champs (Guide de programmation C#)

Article • 01/06/2023

Un *champ* est une variable de tout type qui est déclarée directement dans une classe ou un struct. Les champs sont *membres* de leur type contenant.

Les classes et les structs peuvent avoir des champs d'instance, des champs statiques, ou les deux. Les champs d'instance sont spécifiques à une instance de type. Si vous avez une classe `T` avec un champ d'instance `F`, vous pouvez créer deux objets de type `T` et modifier la valeur de `F` dans chaque objet, sans affecter la valeur de l'autre objet. En revanche, un champ statique appartient au type, et il est partagé entre toutes les instances de ce type. Vous pouvez accéder au champ statique uniquement à l'aide du nom de type. Si vous accédez au champ statique par un nom d'instance, vous obtenez l'erreur de compilation CS0176 .

En règle générale, vous devez déclarer l'accessibilité à `private` ou `protected` pour les champs. Les données que votre type expose au code client doivent être fournies via des méthodes, des propriétés et des indexeurs. L'utilisation de ces constructions pour l'accès indirect aux champs internes permet d'éviter les valeurs d'entrée non valides. Un champ privé qui stocke les données exposées par une propriété publique est appelé *magasin de stockage* ou *champ de stockage*. Vous pouvez déclarer des champs `public`, mais vous ne pouvez pas empêcher le code qui utilise votre type de définir ce champ sur une valeur non valide ou modifier les données d'un objet.

En général, les champs stockent les données qui doivent être accessibles à plusieurs méthodes de type et doivent être stockées plus longtemps que la durée de vie d'une méthode. Par exemple, un type qui représente une date de calendrier peut avoir trois champs de type entier : un pour le mois, un pour le jour et un pour l'année. Les variables qui ne sont pas utilisées en dehors de la portée d'une seule méthode doivent être déclarées comme des *variables locales* dans le corps de la méthode.

Les champs sont déclarés dans le bloc Class ou Struct en spécifiant le niveau d'accès, suivi du type, puis du nom du champ. Par exemple :

C#

```
public class CalendarEntry
{

    // private field (Located near wrapping "Date" property).
    private DateTime _date;

    // Public property exposes _date field safely.
```

```
    public DateTime Date
    {
        get
        {
            return _date;
        }
        set
        {
            // Set some reasonable boundaries for likely birth dates.
            if (value.Year > 1900 && value.Year <= DateTime.Today.Year)
            {
                _date = value;
            }
            else
            {
                throw new ArgumentOutOfRangeException("Date");
            }
        }
    }

    // public field (Generally not recommended).
    public string? Day;

    // Public method also exposes _date field safely.
    // Example call: birthday.SetDate("1975, 6, 30");
    public void SetDate(string dateString)
    {
        DateTime dt = Convert.ToDateTime(dateString);

        // Set some reasonable boundaries for likely birth dates.
        if (dt.Year > 1900 && dt.Year <= DateTime.Today.Year)
        {
            _date = dt;
        }
        else
        {
            throw new ArgumentOutOfRangeException("dateString");
        }
    }

    public TimeSpan GetTimeSpan(string dateString)
    {
        DateTime dt = Convert.ToDateTime(dateString);

        if (dt.Ticks < _date.Ticks)
        {
            return _date - dt;
        }
        else
        {
            throw new ArgumentOutOfRangeException("dateString");
        }
    }
}
```

Pour accéder à un champ dans une instance, ajoutez un point après le nom de l'objet, suivi du nom de l'instance, comme dans `instancename._fieldName`. Par exemple :

```
C#

CalendarEntry birthday = new CalendarEntry();
birthday.Day = "Saturday";
```

Une valeur initiale peut être attribuée à un champ à l'aide de l'opérateur d'assignation, lorsque le champ est déclaré. Pour assigner automatiquement le champ `Day` à `"Monday"`, par exemple, vous devez déclarer `Day`, comme dans l'exemple suivant :

```
C#

public class CalendarDateWithInitialization
{
    public string Day = "Monday";
    //...
}
```

Les champs sont initialisés juste avant le constructeur pour l'instance d'objet qui est appelée. Si le constructeur assigne la valeur d'un champ, la valeur assignée pendant la déclaration du champ sera remplacée. Pour plus d'informations, consultez Utilisation de constructeurs.

> ⓘ **Notes**
>
> Un initialiseur de champ ne peut pas référencer d'autres champs d'instance.

Les champs peuvent être marqués comme public, private, protected, internal, protected internal ou private protected. Ces modificateurs d'accès définissent comment les utilisateurs du type peuvent accéder aux champs. Pour plus d'informations, consultez Modificateurs d'accès.

Facultatif : vous pouvez déclarer un champ comme static. Les champs statiques sont accessibles à tout moment aux appelants, même s'il n'existe aucune instance du type. Pour plus d'informations, consultez la page Classes statiques et membres de classes statiques.

Un champ peut être déclaré readonly. Un champ en lecture seule ne peut recevoir de valeur que pendant l'initialisation ou dans un constructeur. Un champ `static readonly` est similaire à une constante, à ceci près que le compilateur C# n'a pas accès à la valeur

d'un champ statique en lecture seule au moment de la compilation, mais seulement au moment de l'exécution. Pour plus d'informations, consultez Constantes.

Un champ peut être déclaré required. Un champ obligatoire doit être initialisé par le constructeur ou par un initialiseur d'objet lors de la création d'un objet. Vous ajoutez l'attribut System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute à n'importe quelle déclaration de constructeur qui initialise tous les membres requis.

Le modificateur `required` ne peut pas être combiné avec le modificateur `readonly` sur le même champ. Toutefois, la propriété peut être `required` et `init` uniquement.

À compter de C# 12, les paramètres du constructeur principal constituent une alternative à la déclaration de champs. Lorsque votre type a des dépendances qui doivent être fournies lors de l'initialisation, vous pouvez créer un constructeur principal qui fournit ces dépendances. Ces paramètres peuvent être capturés et utilisés à la place des champs déclarés dans vos types. Dans le cas des types record, les paramètres du constructeur principal sont exposés en tant que propriétés publiques.

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Guide de programmation C#
- Système de types C#
- Utilisation de constructeurs
- Héritage
- Modificateurs d'accès
- Classes abstract et sealed et membres de classe

Constantes (Guide de programmation C#)

Article • 07/04/2023

Les constantes sont des valeurs immuables qui sont connues au moment de la compilation et qui ne changent pas pendant la durée de vie du programme. Les constantes sont déclarées avec le modificateur const. Seuls les types intégrés en C# (à l'exception de System.Object) peuvent être déclarés comme `const`. Les types définis par l'utilisateur, notamment les classes, les structs et les tableaux, ne peuvent pas être `const`. Utilisez le modificateur readonly pour créer une classe, un struct ou un tableau initialisé une fois au moment de l'exécution (par exemple, dans un constructeur) et qui ne peut pas être modifié ultérieurement.

C# ne prend pas en charge les propriétés, les événements ou les méthodes `const`.

Le type enum vous permet de définir des constantes nommées pour les types intégrés intégraux (par exemple, `int`, `uint`, `long`, etc.). Pour plus d'informations, consultez enum.

Les constantes doivent être initialisées à mesure qu'elles sont déclarées. Par exemple :

C#

```
class Calendar1
{
    public const int Months = 12;
}
```

Dans cet exemple, la constante `Months` est toujours 12, et ne peut pas être modifiée, même par la classe elle-même. En fait, quand le compilateur rencontre un identificateur constant dans du code source C# (par exemple, `Months`), il substitue la valeur littérale directement dans le code en langage intermédiaire (IL, Intermediate Language) qu'il produit. Étant donné qu'aucune adresse de variable n'est associée à une constante au moment de l'exécution, les champs `const` ne peuvent pas être passés par référence et ne peuvent pas apparaître comme l-value dans une expression.

> ⓘ **Notes**
>
> Procédez avec précaution lorsque vous faites référence à des valeurs constantes définies dans un autre code telles que des DLL. Si une nouvelle version de la DLL

définit une nouvelle valeur pour la constante, votre programme conserve l'ancienne valeur littérale jusqu'à ce qu'elle soit recompilée sur la nouvelle version.

Plusieurs constantes du même type peuvent être déclarées en même temps, par exemple :

C#

```
class Calendar2
{
    public const int Months = 12, Weeks = 52, Days = 365;
}
```

L'expression utilisée pour initialiser une constante peut référencer une autre constante si elle ne crée pas de référence circulaire. Par exemple :

C#

```
class Calendar3
{
    public const int Months = 12;
    public const int Weeks = 52;
    public const int Days = 365;

    public const double DaysPerWeek = (double) Days / (double) Weeks;
    public const double DaysPerMonth = (double) Days / (double) Months;
}
```

Les constantes peuvent être marquées comme public, private, protected, internal, protected internal ou private protected. Ces modificateurs d'accès définissent la façon dont les utilisateurs de la classe peuvent accéder à la constante. Pour plus d'informations, consultez Modificateurs d'accès.

Les constantes sont accessibles comme si elles étaient des champs static, car la valeur de la constante est identique pour toutes les instances du type. Vous n'utilisez pas le mot clé `static` pour les déclarer. Les expressions ne se trouvant pas dans la classe qui définit la constante doivent utiliser le nom de la classe, un point et le nom de la constante pour accéder à la constante. Par exemple :

C#

```
int birthstones = Calendar.Months;
```

Spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Guide de programmation C#
- Propriétés
- Types
- readonly
- Immutability in C# Part One: Kinds of Immutability

Comment définir des propriétés abstraites (Guide de programmation C#)

Article • 07/04/2023

L'exemple suivant montre comment définir des propriétés abstract. Une déclaration de propriété abstraite ne fournit pas une implémentation des accesseurs de propriété ; elle déclare que la classe prend en charge des propriétés, mais laisse l'implémentation de l'accesseur aux classes dérivées. L'exemple suivant montre comment implémenter les propriétés abstraites héritées d'une classe de base.

Cet exemple se compose de trois fichiers, chacun étant compilé individuellement, et l'assembly obtenu est référencé par la compilation suivante :

- abstractshape.cs : classe `Shape` qui contient une propriété `Area` abstraite.
- shapes.cs : sous-classes de la classe `Shape`.
- shapetest.cs : programme de test destiné à afficher les zones de certains objets dérivés de `Shape`.

Pour compiler l'exemple, utilisez la commande suivante :

```
csc abstractshape.cs shapes.cs shapetest.cs
```

Cette commande crée le fichier exécutable shapetest.exe.

Exemples

Ce fichier déclare la classe `Shape` qui contient la propriété `Area` du type `double`.

C#

```
// compile with: csc -target:library abstractshape.cs
public abstract class Shape
{
    private string name;

    public Shape(string s)
    {
        // calling the set accessor of the Id property.
        Id = s;
    }

    public string Id
    {
```

```
        get
        {
            return name;
        }

        set
        {
            name = value;
        }
    }

    // Area is a read-only property - only a get accessor is needed:
    public abstract double Area
    {
        get;
    }

    public override string ToString()
    {
        return $"{Id} Area = {Area:F2}";
    }
}
```

- Les modificateurs sur la propriété sont placés sur la déclaration de propriété proprement dite. Par exemple :

  C#

  ```
  public abstract double Area
  ```

- Quand vous déclarez une propriété abstraite (telle que `Area` dans cet exemple), vous indiquez simplement quels les accesseurs de propriété sont disponibles, mais vous ne les implémentez pas. Dans cet exemple, seul un accesseur get est disponible, donc la propriété est en lecture seule.

Le code suivant présente trois sous-classes de `Shape` et indique comment elles substituent la propriété `Area` pour fournir leur propre implémentation.

C#

```
// compile with: csc -target:library -reference:abstractshape.dll shapes.cs
public class Square : Shape
{
    private int side;

    public Square(int side, string id)
        : base(id)
    {
        this.side = side;
```

```
    }

    public override double Area
    {
        get
        {
            // Given the side, return the area of a square:
            return side * side;
        }
    }
}

public class Circle : Shape
{
    private int radius;

    public Circle(int radius, string id)
        : base(id)
    {
        this.radius = radius;
    }

    public override double Area
    {
        get
        {
            // Given the radius, return the area of a circle:
            return radius * radius * System.Math.PI;
        }
    }
}

public class Rectangle : Shape
{
    private int width;
    private int height;

    public Rectangle(int width, int height, string id)
        : base(id)
    {
        this.width = width;
        this.height = height;
    }

    public override double Area
    {
        get
        {
            // Given the width and height, return the area of a rectangle:
            return width * height;
        }
    }
}
```

Le code suivant montre un programme de test qui crée un certain nombre d'objets dérivés de `Shape` et imprime les zones correspondantes.

C#

```
// compile with: csc -reference:abstractshape.dll;shapes.dll shapetest.cs
class TestClass
{
    static void Main()
    {
        Shape[] shapes =
        {
            new Square(5, "Square #1"),
            new Circle(3, "Circle #1"),
            new Rectangle( 4, 5, "Rectangle #1")
        };

        System.Console.WriteLine("Shapes Collection");
        foreach (Shape s in shapes)
        {
            System.Console.WriteLine(s);
        }
    }
}
/* Output:
    Shapes Collection
    Square #1 Area = 25.00
    Circle #1 Area = 28.27
    Rectangle #1 Area = 20.00
*/
```

Voir aussi

- Guide de programmation C#
- Système de types C#
- Classes abstract et sealed et membres de classe
- Propriétés

Comment : définir des constantes en C#

Article • 07/04/2023

Les constantes sont des champs dont les valeurs sont définies au moment de la compilation et ne sont pas modifiables. Utilisez des constantes pour fournir des noms explicites au lieu de littéraux numériques (« nombres magiques ») pour les valeurs spéciales.

> ⓘ **Notes**
>
> En C#, la directive de préprocesseur **#define** ne peut pas être utilisée pour définir des constantes de la manière généralement utilisée en C et C++.

Pour définir des valeurs constantes de types intégraux (`int`, `byte`, etc.), utilisez un type énuméré. Pour plus d'informations, consultez enum.

Pour définir des constantes non intégrales, une approche consiste à les regrouper dans une classe statique unique nommée `Constants`. Cette opération exige que toutes les références aux constantes soient précédées par le nom de classe, comme dans l'exemple suivant.

Exemple

C#

```
static class Constants
{
    public const double Pi = 3.14159;
    public const int SpeedOfLight = 300000; // km per sec.
}

class Program
{
    static void Main()
    {
        double radius = 5.3;
        double area = Constants.Pi * (radius * radius);
        int secsFromSun = 149476000 / Constants.SpeedOfLight; // in km
        Console.WriteLine(secsFromSun);
    }
}
```

L’utilisation du qualificateur de nom de classe permet de garantir que les personnes qui utilisent la constante, dont vous-même, comprennent qu’il s’agit d’une valeur constante et qu’elle ne peut pas être modifiée.

Voir aussi

- Système de types C#

Propriétés (Guide de programmation C#)

Article • 07/04/2023

Une propriété est un membre qui fournit un mécanisme flexible pour la lecture, l'écriture ou le calcul de la valeur d'un champ privé. Les propriétés peuvent être utilisées comme s'il s'agissait de membres de données publics, mais ce sont en fait des méthodes spéciales appelées *accesseurs*. Cette fonction permet aux données d'être facilement accessibles tout en aidant à promouvoir la sécurité et la flexibilité des méthodes.

Vue d'ensemble des propriétés

- Les propriétés permettent à une classe d'exposer un moyen public d'obtenir et de définir des valeurs, tout en masquant le code d'implémentation ou de vérification.
- Un accesseur de propriété get est utilisé pour retourner la valeur de la propriété et un accesseur de propriété set est utilisé pour affecter une nouvelle valeur. En C# 9 et versions ultérieures, un accesseur de propriété init est utilisé pour affecter une nouvelle valeur uniquement pendant la construction d'objet. Ces accesseurs peuvent avoir différents niveaux d'accès. Pour plus d'informations, consultez Restriction d'accessibilité de l'accesseur.
- Le mot clé value est utilisé pour définir la valeur affectée par l'accesseur `set` ou `init`.
- Les propriétés peuvent être *en lecture-écriture* (elles ont un accesseur `get` et un accesseur `set`), *en lecture seule* (elles ont un accesseur `get`, mais pas d'accesseur `set`) ou *en écriture seule* (elles ont un accesseur `set`, mais pas d'accesseur `get`). Les propriétés en écriture seule sont rares et sont généralement utilisées pour restreindre l'accès aux données sensibles.
- Les propriétés simples qui ne nécessitent pas de code d'accesseur personnalisé peuvent être implémentées en tant que définitions de corps d'expression ou en tant que propriétés implémentées automatiquement.

Propriétés avec des champs de stockage

Un modèle de base pour l'implémentation d'une propriété consiste à utiliser un champ de stockage privé pour définir et extraire la valeur de propriété. L'accesseur `get` retourne la valeur du champ privé et l'accesseur `set` peut effectuer une validation des données avant d'assigner une valeur au champ privé. Les deux accesseurs peuvent

également effectuer des opérations de conversion ou de calcul sur les données avant de stocker ou retourner les données.

L'exemple suivant illustre ce modèle. Dans cet exemple, la classe `TimePeriod` représente un intervalle de temps. La classe stocke l'intervalle de temps en secondes, en interne, dans un champ privé nommé `_seconds`. L'utilisateur peut éventuellement spécifier l'intervalle de temps en heures à l'aide de la propriété en lecture-écriture `Hours`. Les deux accesseurs de propriété `get` et `set` effectuent ensuite la conversion nécessaire des heures en secondes. De plus, l'accesseur `set` valide les données et lève une exception ArgumentOutOfRangeException si le nombre d'heures n'est pas valide.

C#

```
public class TimePeriod
{
    private double _seconds;

    public double Hours
    {
        get { return _seconds / 3600; }
        set
        {
            if (value < 0 || value > 24)
                throw new ArgumentOutOfRangeException(nameof(value),
                      "The valid range is between 0 and 24.");

            _seconds = value * 3600;
        }
    }
}
```

Vous pouvez accéder aux propriétés pour obtenir et définir la valeur, comme illustré dans l'exemple suivant :

C#

```
TimePeriod t = new TimePeriod();
// The property assignment causes the 'set' accessor to be called.
t.Hours = 24;

// Retrieving the property causes the 'get' accessor to be called.
Console.WriteLine($"Time in hours: {t.Hours}");
// The example displays the following output:
//    Time in hours: 24
```

Définitions de corps d'expression

Les accesseurs de propriété sont souvent des instructions sur une ligne qui ne font qu'assigner ou retourner le résultat d'une expression. Vous pouvez implémenter ces propriétés en tant que membres expression-bodied. Les définitions de corps d'expression se composent du symbole `=>` suivi de l'expression à assigner à la propriété ou à récupérer de la propriété.

Les propriétés en lecture seule peuvent implémenter l'accesseur `get` en tant que membre expression-bodied. Dans ce cas, le mot clé d'accesseur `get` et le mot clé `return` ne sont pas utilisés. L'exemple suivant implémente la propriété en lecture seule `Name` en tant que membre expression-bodied.

C#

```
public class Person
{
    private string _firstName;
    private string _lastName;

    public Person(string first, string last)
    {
        _firstName = first;
        _lastName = last;
    }

    public string Name => $"{_firstName} {_lastName}";
}
```

Les accesseurs `get` et `set` peuvent être implémentés en tant que membres expression-bodied. Dans ce cas, les mots clés `get` et `set` doivent être spécifiés. L'exemple suivant illustre l'utilisation de définitions de corps d'expression pour les deux accesseurs. Le mot clé `return` n'est pas utilisé avec l'accesseur `get`.

C#

```
public class SaleItem
{
    string _name;
    decimal _cost;

    public SaleItem(string name, decimal cost)
    {
        _name = name;
        _cost = cost;
    }

    public string Name
    {
        get => _name;
```

```
        set => _name = value;
    }

    public decimal Price
    {
        get => _cost;
        set => _cost = value;
    }
}
```

Propriétés implémentées automatiquement

Dans certains cas, les accesseurs de propriété `get` et `set` ne font qu'assigner une valeur à un champ de stockage, ou récupérer une valeur d'un champ de stockage, sans inclure de logique supplémentaire. En utilisant des propriétés implémentées automatiquement, vous simplifiez votre code, tout en laissant le compilateur C# fournir le champ de stockage de manière transparente.

Si une propriété a à la fois un accesseur `get` et un accesseur `set` (ou `get` et `init`), les deux doivent être implémentés automatiquement. Vous définissez une propriété implémentée automatiquement à l'aide des mots clés `get` et `set` sans fournir d'implémentation. L'exemple suivant est identique à l'exemple précédent, sauf qu'il utilise les propriétés implémentées automatiquement `Name` et `Price`. L'exemple supprime également le constructeur paramétrable afin que les objets `SaleItem` soient désormais initialisés avec un appel au constructeur sans paramètre et à un initialiseur d'objet.

C#

```
public class SaleItem
{
    public string Name
    { get; set; }

    public decimal Price
    { get; set; }
}
```

Les propriétés implémentées automatiquement peuvent déclarer différentes accessibilités pour les accesseurs `get` et `set`. Vous déclarez généralement un accesseur `get` public et un accesseur `set` privé. Pour en savoir plus, consultez l'article sur la restriction de l'accessibilité de l'accesseur.

Propriétés requises

À partir de C# 11, vous pouvez ajouter le membre `required` pour forcer le code client à initialiser une propriété ou un champ :

```
C#
```

```
public class SaleItem
{
    public required string Name
    { get; set; }

    public required decimal Price
    { get; set; }
}
```

Pour créer un `SaleItem`, vous devez définir les propriétés `Name` et `Price` à l'aide d'initialiseurs d'objet, comme indiqué dans le code suivant :

```
C#
```

```
var item = new SaleItem { Name = "Shoes", Price = 19.95m };
Console.WriteLine($"{item.Name}: sells for {item.Price:C2}");
```

Sections connexes

- Utilisation de propriétés
- Propriétés de l'interface
- Comparaison entre propriétés et indexeurs
- Restriction d'accessibilité de l'accesseur
- Propriétés implémentées automatiquement

Spécification du langage C#

Pour plus d'informations, consultez Propriétés dans la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Indexeurs
- get, mot clé
- set, mot clé

Utilisation de propriétés (Guide de programmation C#)

Article • 07/04/2023

Les propriétés allient des caractéristiques des champs et des méthodes. Pour l'utilisateur d'un objet, une propriété s'apparente à un champ. Pour accéder à celle-ci, il doit utiliser la même syntaxe. Pour l'implémenteur d'une classe, une propriété est constituée d'un ou deux blocs de code, représentant un accesseur get et/ou un accesseur set. Le bloc de code correspondant à l'accesseur `get` est exécuté à la lecture de la propriété ; le bloc de code correspondant à l'accesseur `set` est exécuté au moment où une valeur est assignée à la propriété. Une propriété sans accesseur `set` est considérée comme étant en lecture seule. Une propriété sans accesseur `get` est considérée comme étant en écriture seule. Une propriété qui possède les deux accesseurs est en lecture-écriture. En C# 9 et versions ultérieures, vous pouvez utiliser un accesseur `init` au lieu d'un accesseur `set` pour rendre la propriété en lecture seule.

Contrairement aux champs, les propriétés ne sont pas classifiées en tant que variables. Par conséquent, vous ne pouvez pas passer une propriété en tant que paramètre ref ou out.

Les propriétés ont diverses utilisations : elles peuvent valider les données avant d'autoriser une modification ; elles peuvent exposer de manière transparente les données d'une classe quand celles-ci sont extraites d'une autre source, telle qu'une base de données ; elles peuvent effectuer une action quand des données sont modifiées, comme déclencher un événement ou modifier la valeur d'autres champs.

Les propriétés sont déclarées dans le bloc de classe en spécifiant le niveau d'accès du champ, suivi du type de la propriété, du nom de la propriété et d'un bloc de code qui déclare un accesseur `get` et/ou un accesseur `set`. Par exemple :

C#

```
public class Date
{
    private int _month = 7;  // Backing store

    public int Month
    {
        get => _month;
        set
        {
            if ((value > 0) && (value < 13))
            {
```

```
                _month = value;
            }
        }
    }
}
```

Dans cet exemple, `Month` est déclaré en tant que propriété pour permettre à l'accesseur `set` de vérifier que la valeur définie de `Month` se trouve bien entre 1 et 12. La propriété `Month` utilise un champ privé pour assurer le suivi de la valeur réelle. L'emplacement réel des données d'une propriété est souvent appelé « magasin de stockage » de la propriété. Il est courant que les propriétés utilisent des champs privés comme magasin de stockage. Le champ est marqué comme étant privé pour garantir qu'il ne peut être modifié qu'en appelant la propriété. Pour plus d'informations sur les restrictions d'accès public et privé, consultez Modificateurs d'accès.

Les propriétés implémentées automatiquement fournissent une syntaxe simplifiée pour des déclarations de propriété simples. Pour plus d'informations, consultez Propriétés implémentées automatiquement.

Accesseur get

Le corps de l'accesseur `get` ressemble à celui d'une méthode. Il doit retourner une valeur du type de la propriété. L'exécution de l'accesseur `get` revient à lire la valeur du champ. Par exemple, quand vous retournez la variable privée de l'accesseur `get` et que les optimisations sont activées, l'appel à la méthode de l'accesseur `get` est intégré (inline) par le compilateur, si bien il n'existe aucune surcharge d'appel de méthode. Cependant, une méthode d'accesseur `get` virtuelle ne peut pas être inline, car le compilateur ne sait pas au moment de la compilation quelle méthode peut être effectivement appelée au moment de l'exécution. Voici un exemple d'accesseur `get` qui retourne la valeur d'un champ privé `_name` :

C#

```
class Employee
{
    private string _name;  // the name field
    public string Name => _name;     // the Name property
}
```

Quand vous faites référence à la propriété (pas en tant que cible d'une assignation), l'accesseur `get` est appelé pour lire la valeur de la propriété. Par exemple :

C#

```
var employee= new Employee();
//...

System.Console.Write(employee.Name);  // the get accessor is invoked here
```

L'accesseur `get` doit se terminer dans une instruction return ou throw et le contrôle ne peut pas déborder du corps de l'accesseur.

> ⚠ **Avertissement**
>
> Modifier l'état de l'objet en utilisant l'accesseur `get` n'est pas un style de programmation approprié.

L'accesseur `get` peut être utilisé pour retourner la valeur du champ ou pour la calculer et la retourner. Par exemple :

C#

```
class Manager
{
    private string _name;
    public string Name => _name != null ? _name : "NA";
}
```

Dans le segment de code précédent, si vous n'assignez pas de valeur à la propriété `Name`, elle retourne la valeur `NA`.

Accesseur set

L'accesseur `set` ressemble à une méthode dont le type de retour est void. Il utilise un paramètre implicite nommé `value`, dont le type est celui de la propriété. Dans l'exemple ci-dessous, un accesseur `set` est ajouté à la propriété `Name` :

C#

```
class Student
{
    private string _name;  // the name field
    public string Name    // the Name property
    {
        get => _name;
        set => _name = value;
```

```
    }
}
```

Quand vous assignez une valeur à la propriété, l'accesseur `set` est appelé en utilisant un argument qui fournit la nouvelle valeur. Par exemple :

C#

```
var student = new Student();
student.Name = "Joe";  // the set accessor is invoked here

System.Console.Write(student.Name);  // the get accessor is invoked here
```

Utiliser le nom de paramètre implicite, `value`, pour une déclaration de variable locale dans un accesseur `set`, est une erreur.

Accesseur init

Le code permettant de créer un accesseur `init` est identique au code utilisé pour créer un accesseur `set`, sauf que vous utilisez le mot clé `init` au lieu de `set`. La différence est que l'accesseur `init` ne peut être utilisé que dans le constructeur ou à l'aide d'un initialiseur d'objet.

Remarques

Les propriétés peuvent être marquées comme étant `public`, `private`, `protected`, `internal`, `protected internal` ou `private protected`. Ces modificateurs d'accès définissent comment les utilisateurs de la classe peuvent accéder à la propriété. Les accesseurs `get` et `set` d'une même propriété peuvent avoir des modificateurs d'accès différents. Par exemple, l'accesseur `get` peut être `public` pour autoriser l'accès en lecture seule en dehors du type, tandis que l'accesseur `set` peut être `private` ou `protected`. Pour plus d'informations, consultez Modificateurs d'accès.

Une propriété peut être déclarée en tant que propriété statique à l'aide du mot clé `static`. Les propriétés statiques sont accessibles à tout moment aux appelants, même s'il n'existe aucune instance de la classe. Pour plus d'informations, consultez la page Classes statiques et membres de classes statiques.

Une propriété peut être marquée comme étant une propriété virtuelle à l'aide du mot clé virtual. Les propriétés virtuelles permettent aux classes dérivées de substituer le

comportement de la propriété à l'aide du mot clé override. Pour plus d'informations sur ces options, consultez Héritage.

Une propriété qui se substitue à une propriété virtuelle peut aussi être sealed, ce qui signifie que pour les classes dérivées, elle n'est plus virtuelle. Enfin, une propriété peut être déclarée comme étant abstract. Les propriétés abstraites ne définissent pas d'implémentation dans la classe, et les classes dérivées doivent écrire leur propre implémentation. Pour plus d'informations sur ces options, consultez Classes abstract et sealed et membres de classe.

> ⓘ **Notes**
>
> Il s'agit d'une erreur d'utiliser un modificateur **virtual**, **abstract** ou **override** sur un accesseur de propriété **static**.

Exemples

Cet exemple illustre les propriétés d'instance, statiques et en lecture seule. Il accepte le nom de l'employé à partir du clavier, incrémente `NumberOfEmployees` de 1 et affiche le nom et le numéro de l'employé.

C#

```
public class Employee
{
    public static int NumberOfEmployees;
    private static int _counter;
    private string _name;

    // A read-write instance property:
    public string Name
    {
        get => _name;
        set => _name = value;
    }

    // A read-only static property:
    public static int Counter => _counter;

    // A Constructor:
    public Employee() => _counter = ++NumberOfEmployees; // Calculate the
employee's number:
}
```

Exemple de propriété masquée

Cet exemple montre comment accéder à une propriété de classe de base qui est masquée par une autre propriété qui porte le même nom dans une classe dérivée :

C#

```
public class Employee
{
    private string _name;
    public string Name
    {
        get => _name;
        set => _name = value;
    }
}

public class Manager : Employee
{
    private string _name;

    // Notice the use of the new modifier:
    public new string Name
    {
        get => _name;
        set => _name = value + ", Manager";
    }
}

class TestHiding
{
    public static void Test()
    {
        Manager m1 = new Manager();

        // Derived class property.
        m1.Name = "John";

        // Base class property.
        ((Employee)m1).Name = "Mary";

        System.Console.WriteLine("Name in the derived class is: {0}",
m1.Name);
        System.Console.WriteLine("Name in the base class is: {0}",
((Employee)m1).Name);
    }
}
/* Output:
    Name in the derived class is: John, Manager
    Name in the base class is: Mary
*/
```

Voici les points importants de l'exemple précédent :

- La propriété `Name` de la classe dérivée masque la propriété `Name` de la classe de base. En pareil cas, le modificateur `new` est utilisé dans la déclaration de la propriété de la classe dérivée :

  ```
  C#

  public new string Name
  ```

- La conversion de type `(Employee)` est utilisée pour accéder à la propriété masquée de la classe de base :

  ```
  C#

  ((Employee)m1).Name = "Mary";
  ```

Pour plus d'informations sur le masquage des membres, consultez new, modificateur.

Exemple de propriété de remplacement

Dans cet exemple, deux classes, `Cube` et `Square`, implémentent une classe abstract, `Shape`, et remplacent sa propriété `Area` abstract. Notez l'utilisation du modificateur override sur les propriétés. Le programme accepte le côté (« side ») comme entrée et calcule les surfaces (« areas ») du carré (« square ») et du cube. De même, il accepte la surface (« area ») comme entrée et calcule le côté (« side ») correspondant du carré (« square ») et du cube.

```
C#

abstract class Shape
{
    public abstract double Area
    {
        get;
        set;
    }
}

class Square : Shape
{
    public double side;

    //constructor
    public Square(double s) => side = s;
```

```
    public override double Area
    {
        get => side * side;
        set => side = System.Math.Sqrt(value);
    }
}

class Cube : Shape
{
    public double side;

    //constructor
    public Cube(double s) => side = s;

    public override double Area
    {
        get => 6 * side * side;
        set => side = System.Math.Sqrt(value / 6);
    }
}

class TestShapes
{
    static void Main()
    {
        // Input the side:
        System.Console.Write("Enter the side: ");
        double side = double.Parse(System.Console.ReadLine());

        // Compute the areas:
        Square s = new Square(side);
        Cube c = new Cube(side);

        // Display the results:
        System.Console.WriteLine("Area of the square = {0:F2}", s.Area);
        System.Console.WriteLine("Area of the cube = {0:F2}", c.Area);
        System.Console.WriteLine();

        // Input the area:
        System.Console.Write("Enter the area: ");
        double area = double.Parse(System.Console.ReadLine());

        // Compute the sides:
        s.Area = area;
        c.Area = area;

        // Display the results:
        System.Console.WriteLine("Side of the square = {0:F2}", s.side);
        System.Console.WriteLine("Side of the cube = {0:F2}", c.side);
    }
}
/* Example Output:
    Enter the side: 4
    Area of the square = 16.00
    Area of the cube = 96.00
```

```
    Enter the area: 24
    Side of the square = 4.90
    Side of the cube = 2.00
*/
```

Voir aussi

- Propriétés
- Propriétés de l'interface
- Propriétés implémentées automatiquement

Propriétés d'interface (Guide de programmation C#)

Article • 07/04/2023

Des propriétés peuvent être déclarées dans une interface. L'exemple suivant déclare un accesseur de propriété d'interface :

C#

```
public interface ISampleInterface
{
    // Property declaration:
    string Name
    {
        get;
        set;
    }
}
```

Les propriétés d'interface n'ont pas de corps en général. Les accesseurs indiquent si la propriété est en lecture-écriture, en lecture seule ou en écriture seule. Contrairement aux classes et aux structs, la déclaration des accesseurs sans corps ne déclare pas de propriété implémentée automatiquement. Une interface peut définir une implémentation par défaut pour les membres, notamment les propriétés. La définition d'une implémentation par défaut pour une propriété dans une interface est rare, car les interfaces peuvent ne pas définir de champs de données d'instance.

Exemple

Dans cet exemple, l'interface `IEmployee` a une propriété en lecture-écriture, `Name`, et une propriété en lecture seule, `Counter`. La classe `Employee` implémente l'interface `IEmployee` et utilise ces deux propriétés. Le programme lit le nom d'un nouvel employé et le nombre actuel d'employés et affiche le nom de l'employé et le nombre d'employés calculé.

Vous pouvez utiliser le nom qualifié complet de la propriété, qui fait référence à l'interface dans laquelle le membre est déclaré. Par exemple :

C#

```
string IEmployee.Name
{
```

```
    get { return "Employee Name"; }
    set { }
}
```

L'exemple précédent illustre l'implémentation d'interface explicite. Par exemple, si la classe `Employee` implémente deux interfaces, `ICitizen` et `IEmployee`, et que les deux interfaces ont la même propriété `Name`, l'implémentation de membre d'interface explicite est nécessaire. Autrement dit, la déclaration de propriété suivante :

C#

```
string IEmployee.Name
{
    get { return "Employee Name"; }
    set { }
}
```

implémente la propriété `Name` dans l'interface `IEmployee`, alors que la déclaration suivante :

C#

```
string ICitizen.Name
{
    get { return "Citizen Name"; }
    set { }
}
```

implémente la propriété `Name` dans l'interface `ICitizen`.

C#

```
interface IEmployee
{
    string Name
    {
        get;
        set;
    }

    int Counter
    {
        get;
    }
}

public class Employee : IEmployee
{
```

```
    public static int numberOfEmployees;

    private string _name;
    public string Name  // read-write instance property
    {
        get => _name;
        set => _name = value;
    }

    private int _counter;
    public int Counter  // read-only instance property
    {
        get => _counter;
    }

    // constructor
    public Employee() => _counter = ++numberOfEmployees;
}
```

C#

```
System.Console.Write("Enter number of employees: ");
Employee.numberOfEmployees = int.Parse(System.Console.ReadLine());

Employee e1 = new Employee();
System.Console.Write("Enter the name of the new employee: ");
e1.Name = System.Console.ReadLine();

System.Console.WriteLine("The employee information:");
System.Console.WriteLine("Employee number: {0}", e1.Counter);
System.Console.WriteLine("Employee name: {0}", e1.Name);
```

Exemple de sortie

Console

```
Enter number of employees: 210
Enter the name of the new employee: Hazem Abolrous
The employee information:
Employee number: 211
Employee name: Hazem Abolrous
```

Voir aussi

- Guide de programmation C#
- Propriétés
- Utilisation de propriétés

- Comparaison entre propriétés et indexeurs
- Indexeurs
- Interfaces

Restriction d'accessibilité de l'accesseur (Guide de programmation C#)

Article • 07/04/2023

Les parties get et set d'une propriété ou d'un indexeur sont appelées *accesseurs*. Par défaut, ces accesseurs ont la visibilité ou le niveau d'accès de la propriété ou de l'indexeur auquel ils appartiennent. Pour plus d'informations, consultez Niveaux d'accessibilité. Toutefois, il peut parfois s'avérer utile de restreindre l'accès à l'un de ces accesseurs. En général, vous restreignez l'accessibilité de l'accesseur `set`, tout en gardant l'accesseur `get` publiquement accessible. Par exemple :

C#

```
private string _name = "Hello";

public string Name
{
    get
    {
        return _name;
    }
    protected set
    {
        _name = value;
    }
}
```

Dans cet exemple, une propriété appelée `Name` définit un accesseur `get` et `set`. L'accesseur `get` reçoit le niveau d'accessibilité de la propriété elle-même, `public` dans le cas présent, alors que l'accesseur `set` est restreint explicitement par l'application du modificateur d'accès protected à l'accesseur lui-même.

ⓘ Notes

Les exemples de cet article n'utilisent pas de **propriétés implémentées automatiquement**. Les *propriétés implémentées automatiquement* fournissent une syntaxe concise pour déclarer des propriétés lorsqu'un champ de stockage personnalisé n'est pas requis.

Restrictions sur les modificateurs d'accès sur les accesseurs

L'utilisation de modificateurs d'accesseurs sur les propriétés ou les indexeurs est soumise aux conditions suivantes :

- Vous ne pouvez pas utiliser de modificateurs d'accesseur sur une interface ou sur une implémentation de membre d'interface explicite.
- Vous pouvez utiliser les modificateurs d'accesseur uniquement si la propriété ou l'indexeur dispose à la fois d'accesseurs `set` et `get`. Dans ce cas, le modificateur est autorisé uniquement sur l'un des deux accesseurs.
- Si la propriété ou l'indexeur possède un modificateur override, le modificateur d'accesseur doit correspondre à l'accesseur de l'accesseur substitué, le cas échant.
- Le niveau d'accessibilité sur l'accesseur doit être plus restrictif que le niveau d'accessibilité sur la propriété ou l'indexeur eux-mêmes.

Modificateurs d'accès sur les accesseurs de substitution

Quand vous substituez une propriété ou un indexeur, les accesseurs remplacés doivent être accessibles au code de substitution. Par ailleurs, l'accessibilité de la propriété/l'indexeur et de ses accesseurs doivent correspondre à la propriété/l'indexeur et à ses accesseurs substitués. Par exemple :

C#

```
public class Parent
{
    public virtual int TestProperty
    {
        // Notice the accessor accessibility level.
        protected set { }

        // No access modifier is used here.
        get { return 0; }
    }
}
public class Kid : Parent
{
    public override int TestProperty
    {
        // Use the same accessibility level as in the overridden accessor.
        protected set { }

        // Cannot use access modifier here.
```

```
        get { return 0; }
    }
}
```

Implémentation des interfaces

Quand vous utilisez un accesseur pour implémenter une interface, l'accesseur peut ne pas avoir de modificateur d'accès. Toutefois, si vous implémentez l'interface à l'aide d'un accesseur, tel que `get`, l'autre accesseur peut avoir un modificateur d'accès, comme dans l'exemple suivant :

C#

```
public interface ISomeInterface
{
    int TestProperty
    {
        // No access modifier allowed here
        // because this is an interface.
        get;
    }
}

public class TestClass : ISomeInterface
{
    public int TestProperty
    {
        // Cannot use access modifier here because
        // this is an interface implementation.
        get { return 10; }

        // Interface property does not have set accessor,
        // so access modifier is allowed.
        protected set { }
    }
}
```

Domaine d'accessibilité de l'accesseur

Si vous utilisez un modificateur d'accès sur l'accesseur, le domaine d'accessibilité de l'accesseur est déterminé par ce modificateur.

Si vous n'avez pas utilisé un modificateur d'accès sur l'accesseur, le domaine d'accessibilité de l'accesseur est déterminé par le niveau d'accessibilité de la propriété ou de l'indexeur.

Exemple

L'exemple suivant contient trois classes, `BaseClass`, `DerivedClass` et `MainClass`. Il y a deux propriétés sur `BaseClass`, `Name` et `Id` sur les deux classes. L'exemple montre comment la propriété `Id` sur `DerivedClass` peut être masquée par la propriété `Id` sur `BaseClass` quand vous utilisez un modificateur d'accès restrictif tel que protected ou private. Par conséquent, quand vous affectez des valeurs à cette propriété, la propriété sur la classe `BaseClass` est appelée à la place. Le remplacement du modificateur d'accès par public rend la propriété accessible.

L'exemple montre également qu'un modificateur d'accès restrictif, tel que `private` ou `protected`, sur l'accesseur `set` de la propriété `Name` dans `DerivedClass` empêche l'accès à l'accesseur dans la classe dérivée. Il génère une erreur lorsque vous lui donnez une affectation, ou il accède à la propriété de la classe de base du même nom, s'il est accessible.

C#

```
public class BaseClass
{
    private string _name = "Name-BaseClass";
    private string _id = "ID-BaseClass";

    public string Name
    {
        get { return _name; }
        set { }
    }

    public string Id
    {
        get { return _id; }
        set { }
    }
}

public class DerivedClass : BaseClass
{
    private string _name = "Name-DerivedClass";
    private string _id = "ID-DerivedClass";

    new public string Name
    {
        get
        {
            return _name;
        }

        // Using "protected" would make the set accessor not accessible.
```

```
        set
        {
            _name = value;
        }
    }

    // Using private on the following property hides it in the Main Class.
    // Any assignment to the property will use Id in BaseClass.
    new private string Id
    {
        get
        {
            return _id;
        }
        set
        {
            _id = value;
        }
    }
}

class MainClass
{
    static void Main()
    {
        BaseClass b1 = new BaseClass();
        DerivedClass d1 = new DerivedClass();

        b1.Name = "Mary";
        d1.Name = "John";

        b1.Id = "Mary123";
        d1.Id = "John123";  // The BaseClass.Id property is called.

        System.Console.WriteLine("Base: {0}, {1}", b1.Name, b1.Id);
        System.Console.WriteLine("Derived: {0}, {1}", d1.Name, d1.Id);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    Base: Name-BaseClass, ID-BaseClass
    Derived: John, ID-BaseClass
*/
```

Commentaires

Notez que si vous remplacez la déclaration `new private string Id` par `new public string Id`, vous obtenez la sortie suivante :

```
Name and ID in the base class: Name-BaseClass, ID-BaseClass Name and ID in the derived class: John, John123
```

Voir aussi

- Propriétés
- Indexeurs
- Modificateurs d’accès
- Propriétés init only
- Propriétés requises

Comment déclarer et utiliser des propriétés en lecture-écriture (Guide de programmation C#)

Article • 07/04/2023

Les propriétés offrent la commodité des membres de données publics sans les risques liés à un accès non protégé, non contrôlé et non vérifié aux données d'un objet. Les propriétés déclarent des *accesseurs* : lesquels sont des méthodes spéciales qui affectent et récupèrent des valeurs du membre de données sous-jacent. L'accesseur set permet aux membres de données d'être affectés, et l'accesseur get récupère des valeurs de membres de données.

L'exemple suivant montre une classe `Person` qui possède deux propriétés : `Name` (string) et `Age` (int). Étant donné que les deux propriétés fournissent des accesseurs `get` et `set`, elles sont considérées comme des propriétés en lecture/écriture.

Exemple

C#

```
class Person
{
    private string _name = "N/A";
    private int _age = 0;

    // Declare a Name property of type string:
    public string Name
    {
        get
        {
            return _name;
        }
        set
        {
            _name = value;
        }
    }

    // Declare an Age property of type int:
    public int Age
    {
        get
        {
            return _age;
```

```
        }

        set
        {
            _age = value;
        }
    }

    public override string ToString()
    {
        return "Name = " + Name + ", Age = " + Age;
    }
}

public class Wrapper
{
    private string _name = "N/A";
    public string Name
    {
        get
        {
            return _name;
        }
        private set
        {
            _name = value;
        }
    }

}

class TestPerson
{
    static void Main()
    {
        // Create a new Person object:
        Person person = new Person();

        // Print out the name and the age associated with the person:
        Console.WriteLine("Person details - {0}", person);

        // Set some values on the person object:
        person.Name = "Joe";
        person.Age = 99;
        Console.WriteLine("Person details - {0}", person);

        // Increment the Age property:
        person.Age += 1;
        Console.WriteLine("Person details - {0}", person);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

```
/* Output:
    Person details - Name = N/A, Age = 0
    Person details - Name = Joe, Age = 99
    Person details - Name = Joe, Age = 100
*/
```

Programmation fiable

Dans l'exemple précédent, les propriétés `Name` et `Age` sont publiques, et incluent un accesseur `get` et un accesseur `set`. Les accesseurs publics permettent à n'importe quel objet de lire et d'écrire ces propriétés. Toutefois, il est parfois souhaitable d'exclure l'un des accesseurs. Vous pouvez omettre l'accesseur `set` pour rendre la propriété en lecture seule :

C#

```
public string Name
{
    get
    {
        return _name;
    }
    private set
    {
        _name = value;
    }
}
```

Vous pouvez également exposer publiquement un accesseur mais rendre l'autre private ou protected. Pour plus d'informations, consultez Accessibilité de l'accesseur asymétrique.

Une fois les propriétés déclarées, vous pouvez les utiliser comme champs de la classe. Les propriétés permettent une syntaxe naturelle pour obtenir ou définir la valeur d'une propriété, comme dans les instructions suivantes :

C#

```
person.Name = "Joe";
person.Age = 99;
```

Une variable `value` spéciale est disponible dans la méthode `set` d'une propriété. Cette variable contient la valeur que l'utilisateur a spécifiée, par exemple :

C#

```
_name = value;
```

Notez la syntaxe correcte pour incrémenter la propriété `Age` sur un objet `Person` :

C#

```
person.Age += 1;
```

Si des méthodes `set` et `get` distinctes ont été utilisées pour modeler des propriétés, le code équivalent peut avoir la forme suivante :

C#

```
person.SetAge(person.GetAge() + 1);
```

La méthode `ToString` est substituée dans cet exemple :

C#

```
public override string ToString()
{
    return "Name = " + Name + ", Age = " + Age;
}
```

Vous pouvez remarquer que `ToString` n'est pas utilisée de façon explicite dans le programme. Elle est appelée par défaut par les appels `WriteLine`.

Voir aussi

- Propriétés
- Système de types C#

Propriétés implémentées automatiquement (Guide de programmation C#)

Article • 07/04/2023

Les propriétés implémentées automatiquement rendent la déclaration de propriété plus concise quand aucune logique supplémentaire n'est requise dans les accesseurs de propriété. Elles permettent également au code client de créer des objets. Quand vous déclarez une propriété comme indiqué dans l'exemple suivant, le compilateur crée un champ de stockage privé et anonyme uniquement accessible via les accesseurs `get` et `set` de la propriété. Dans C# 9 et versions ultérieures, les accesseurs `init` peuvent également être déclarés en tant que propriétés implémentées automatiquement.

Exemple

L'exemple suivant montre une classe simple qui a des propriétés implémentées automatiquement :

C#

```
// This class is mutable. Its data can be modified from
// outside the class.
public class Customer
{
    // Auto-implemented properties for trivial get and set
    public double TotalPurchases { get; set; }
    public string Name { get; set; }
    public int CustomerId { get; set; }

    // Constructor
    public Customer(double purchases, string name, int id)
    {
        TotalPurchases = purchases;
        Name = name;
        CustomerId = id;
    }

    // Methods
    public string GetContactInfo() { return "ContactInfo"; }
    public string GetTransactionHistory() { return "History"; }

    // .. Additional methods, events, etc.
}
```

```
class Program
{
    static void Main()
    {
        // Initialize a new object.
        Customer cust1 = new Customer(4987.63, "Northwind", 90108);

        // Modify a property.
        cust1.TotalPurchases += 499.99;
    }
}
```

Vous ne pouvez pas déclarer les propriétés implémentées automatiquement dans les interfaces. Les propriétés implémentées automatiquement déclarent un champ de stockage d'instance privée et les interfaces peuvent ne pas déclarer de champs d'instance. La déclaration d'une propriété dans une interface sans définir de corps déclare une propriété avec des accesseurs qui doivent être implémentés par chaque type qui implémente cette interface.

Vous pouvez initialiser des propriétés implémentées automatiquement de la même façon que des champs :

C#

```
public string FirstName { get; set; } = "Jane";
```

La classe qui est illustrée dans l'exemple précédent est mutable. Le code client peut modifier les valeurs dans les objets après leur création. Dans les classes complexes qui contiennent un comportement significatif (méthodes) ainsi que des données, il est souvent nécessaire d'avoir des propriétés publiques. Toutefois, pour les petites classes ou petits structs qui encapsulent simplement un ensemble de valeurs (données) et qui ont peu ou pas de comportements, vous devez utiliser l'une des options suivantes pour rendre les objets immuables :

- Déclarez uniquement un accesseur `get` (immuable partout, à l'exception du constructeur).
- Déclarez un accesseur `get` et un accesseur `init` (immuable partout, sauf pendant la construction de l'objet).
- Déclarez l'accesseur `set` comme privé (immuable pour les consommateurs).

Pour plus d'informations, consultez Comment implémenter une classe légère avec des propriétés implémentées automatiquement (Guide de programmation C#).

Voir aussi

- Utiliser des propriétés implémentées automatiquement (règle de style IDE0032)
- Propriétés
- Modificateurs

Guide pratique pour implémenter une classe légère avec des propriétés implémentées automatiquement (Guide de programmation C#)

Article • 07/04/2023

Cet exemple montre comment créer une classe légère immuable qui sert uniquement à encapsuler un jeu de propriétés implémentées automatiquement. Utilisez ce type de construction à la place d'un struct quand vous devez utiliser une sémantique de type de référence.

Vous pouvez créer une propriété immuable de l'une des manières suivantes :

- Déclarez uniquement l'accesseur get, ce qui rend la propriété immuable partout, sauf dans le constructeur du type.
- Déclarez un accesseur init au lieu d'un accesseur `set`, ce qui rend la propriété définissable uniquement dans le constructeur ou à l'aide d'un initialiseur d'objet.
- Déclarez l'accesseur set comme étant private. La propriété peut être définie dans le type, mais elle est immuable pour les consommateurs.

Vous pouvez ajouter le modificateur required à la déclaration de propriété pour forcer les appelants à définir la propriété dans le cadre de l'initialisation d'un nouvel objet.

L'exemple suivant montre comment une propriété avec un seul accesseur get diffère d'une propriété avec un ensemble get et private.

C#

```
class Contact
{
    public string Name { get; }
    public string Address { get; private set; }

    public Contact(string contactName, string contactAddress)
    {
        // Both properties are accessible in the constructor.
        Name = contactName;
        Address = contactAddress;
    }

    // Name isn't assignable here. This will generate a compile error.
    //public void ChangeName(string newName) => Name = newName;
```

```
    // Address is assignable here.
    public void ChangeAddress(string newAddress) => Address = newAddress;
}
```

Exemple

L'exemple suivant montre deux façons d'implémenter une classe immuable qui possède des propriétés implémentées automatiquement. Chaque façon déclare l'une des propriétés avec un `set` privé et l'autre avec un `get` uniquement. La première classe utilise un constructeur uniquement pour initialiser les propriétés et la deuxième classe utilise une méthode de fabrique statique qui appelle un constructeur.

C#

```
// This class is immutable. After an object is created,
// it cannot be modified from outside the class. It uses a
// constructor to initialize its properties.
class Contact
{
    // Read-only property.
    public string Name { get; }

    // Read-write property with a private set accessor.
    public string Address { get; private set; }

    // Public constructor.
    public Contact(string contactName, string contactAddress)
    {
        Name = contactName;
        Address = contactAddress;
    }
}

// This class is immutable. After an object is created,
// it cannot be modified from outside the class. It uses a
// static method and private constructor to initialize its properties.
public class Contact2
{
    // Read-write property with a private set accessor.
    public string Name { get; private set; }

    // Read-only property.
    public string Address { get; }

    // Private constructor.
    private Contact2(string contactName, string contactAddress)
    {
        Name = contactName;
        Address = contactAddress;
    }
```

```
    // Public factory method.
    public static Contact2 CreateContact(string name, string address)
    {
        return new Contact2(name, address);
    }
}

public class Program
{
    static void Main()
    {
        // Some simple data sources.
        string[] names = {"Terry Adams","Fadi Fakhouri", "Hanying Feng",
                          "Cesar Garcia", "Debra Garcia"};
        string[] addresses = {"123 Main St.", "345 Cypress Ave.", "678 1st
Ave",
                              "12 108th St.", "89 E. 42nd St."};

        // Simple query to demonstrate object creation in select clause.
        // Create Contact objects by using a constructor.
        var query1 = from i in Enumerable.Range(0, 5)
                     select new Contact(names[i], addresses[i]);

        // List elements cannot be modified by client code.
        var list = query1.ToList();
        foreach (var contact in list)
        {
            Console.WriteLine("{0}, {1}", contact.Name, contact.Address);
        }

        // Create Contact2 objects by using a static factory method.
        var query2 = from i in Enumerable.Range(0, 5)
                         select Contact2.CreateContact(names[i],
addresses[i]);

        // Console output is identical to query1.
        var list2 = query2.ToList();

        // List elements cannot be modified by client code.
        // CS0272:
        // list2[0].Name = "Eugene Zabokritski";

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
    Terry Adams, 123 Main St.
    Fadi Fakhouri, 345 Cypress Ave.
    Hanying Feng, 678 1st Ave
    Cesar Garcia, 12 108th St.
```

```
    Debra Garcia, 89 E. 42nd St.
*/
```

Le compilateur crée des champs de stockage pour chaque propriété implémentée automatiquement. Les champs ne sont pas accessibles directement à partir du code source.

Voir aussi

- Propriétés
- struct
- Initialiseurs d'objets et de collections

Méthodes (Guide de programmation C#)

Article • 07/04/2023

Une méthode est un bloc de code qui contient une série d'instructions. Un programme provoque l'exécution des instructions en appelant la méthode et en spécifiant les éventuels arguments de méthode requis. En C#, chaque instruction exécutée est effectuée dans le contexte d'une méthode.

La méthode `Main` est le point d'entrée de chaque application C# et elle est appelée par le Common Language Runtime (CLR) au démarrage du programme. Dans une application qui utilise des instructions de niveau supérieur, la méthode `Main` est générée par le compilateur et contient toutes les instructions de niveau supérieur.

> ⓘ **Notes**
>
> Cet article traite les méthodes nommées. Pour plus d'informations sur les fonctions anonymes, consultez **Expressions lambda**.

Signatures de méthode

Les méthodes sont déclarées dans une classe, struct ou interface en spécifiant le niveau d'accès comme `public` ou `private`, les modificateurs facultatifs comme `abstract` ou `sealed`, la valeur de retour, le nom de la méthode et les éventuels paramètres de méthode. Ces parties forment ensemble la signature de la méthode.

> ⓘ **Important**
>
> Un type de retour d'une méthode ne fait pas partie de la signature de la méthode à des fins de surcharge de méthode. Toutefois, il fait partie de la signature de la méthode lors de la détermination de la compatibilité entre un délégué et la méthode vers laquelle il pointe.

Les paramètres de méthode sont placés entre parenthèses et séparés par des virgules. Des parenthèses vides indiquent que la méthode ne requiert aucun paramètre. Cette classe contient quatre méthodes :

```
C#
```

```
abstract class Motorcycle
{
    // Anyone can call this.
    public void StartEngine() {/* Method statements here */ }

    // Only derived classes can call this.
    protected void AddGas(int gallons) { /* Method statements here */ }

    // Derived classes can override the base class implementation.
    public virtual int Drive(int miles, int speed) { /* Method statements
here */ return 1; }

    // Derived classes must implement this.
    public abstract double GetTopSpeed();
}
```

Accès aux méthodes

L'appel d'une méthode sur un objet revient à accéder à un champ. Après le nom de l'objet, ajoutez un point, le nom de la méthode et les parenthèses. Les arguments sont répertoriés entre les parenthèses et séparés par des virgules. Les méthodes de la classe `Motorcycle` peuvent donc être appelées comme dans l'exemple suivant :

C#

```
class TestMotorcycle : Motorcycle
{

    public override double GetTopSpeed()
    {
        return 108.4;
    }

    static void Main()
    {

        TestMotorcycle moto = new TestMotorcycle();

        moto.StartEngine();
        moto.AddGas(15);
        moto.Drive(5, 20);
        double speed = moto.GetTopSpeed();
        Console.WriteLine("My top speed is {0}", speed);
    }
}
```

Paramètres de méthode et arguments

La définition de la méthode spécifie les noms et types des paramètres requis. Quand le code appelant appelle la méthode, il fournit des valeurs concrètes appelées arguments pour chaque paramètre. Les arguments doivent être compatibles avec le type de paramètre, mais le nom de l'argument (le cas échéant) utilisé dans le code appelant ne doit pas nécessairement être le même que celui du paramètre défini dans la méthode. Par exemple :

C#

```
public void Caller()
{
    int numA = 4;
    // Call with an int variable.
    int productA = Square(numA);

    int numB = 32;
    // Call with another int variable.
    int productB = Square(numB);

    // Call with an integer literal.
    int productC = Square(12);

    // Call with an expression that evaluates to int.
    productC = Square(productA * 3);
}

int Square(int i)
{
    // Store input argument in a local variable.
    int input = i;
    return input * input;
}
```

Passer par référence et passer par valeur

Par défaut, lorsqu'une instance d'un type valeur est passée à une méthode, sa copie est passée au lieu de l'instance elle-même. Ainsi, les modifications apportées à l'argument n'ont aucun effet sur l'instance d'origine dans la méthode d'appel. Pour passer une instance de type valeur par référence, utilisez le mot clé `ref`. Pour plus d'informations, consultez Passage de paramètres de type valeur .

Quand un objet d'un type référence est passé à une méthode, une référence à l'objet est passée. Autrement dit, la méthode ne reçoit pas l'objet lui-même mais un argument qui indique l'emplacement de l'objet. Si vous modifiez un membre de l'objet à l'aide de cette référence, la modification est répercutée dans l'argument de la méthode d'appel, même si vous passez l'objet par valeur.

Vous créez un type référence à l'aide du mot clé `class`, comme le montre l'exemple suivant :

C#

```
public class SampleRefType
{
    public int value;
}
```

Maintenant, si vous passez un objet basé sur ce type à une méthode, une référence à l'objet est passée. L'exemple suivant passe un objet de type `SampleRefType` à la méthode `ModifyObject` :

C#

```
public static void TestRefType()
{
    SampleRefType rt = new SampleRefType();
    rt.value = 44;
    ModifyObject(rt);
    Console.WriteLine(rt.value);
}

static void ModifyObject(SampleRefType obj)
{
    obj.value = 33;
}
```

L'exemple produit essentiellement la même chose que l'exemple précédent, dans la mesure où il passe un argument par valeur à une méthode. Mais, puisqu'un type référence est utilisé, le résultat est différent. La modification apportée dans `ModifyObject` au champ `value` du paramètre, `obj`, modifie également le champ `value` de l'argument, `rt`, dans la méthode `TestRefType` . La méthode `TestRefType` affiche 33 en tant que sortie.

Pour plus d'informations sur le passage de types référence par référence et par valeur, consultez Passage de paramètres de type référence et Types référence.

Valeurs de retour

Les méthodes peuvent retourner une valeur à l'appelant. Si le type de retour (le type qui figure avant le nom de la méthode) n'est pas `void`, la méthode peut retourner la valeur

en utilisant return l'instruction. Une instruction avec le mot clé `return` suivi d'une valeur qui correspond au type de retour retourne cette valeur à l'appelant de la méthode.

La valeur peut être retournée à l'appelant par valeur ou par référence. Les valeurs sont retournées à l'appelant par référence si le mot clé `ref` est utilisé dans la signature de méthode et s'il suit chaque mot clé `return`. Par exemple, la signature de méthode suivante et l'instruction de retour indiquent que la méthode retourne une variable nommée `estDistance` par référence à l'appelant.

C#

```
public ref double GetEstimatedDistance()
{
    return ref estDistance;
}
```

Le mot clé `return` arrête également l'exécution de la méthode. Si le type de retour est `void`, une instruction `return` sans valeur est quand même utile pour arrêter l'exécution de la méthode. Sans le mot clé `return` , la méthode arrête de s'exécuter quand elle atteint la fin du bloc de code. Les méthodes dotées d'un type de retour non void doivent utiliser le mot clé `return` pour retourner une valeur. Par exemple, ces deux méthodes utilisent le mot clé `return` pour retourner des entiers :

C#

```
class SimpleMath
{
    public int AddTwoNumbers(int number1, int number2)
    {
        return number1 + number2;
    }

    public int SquareANumber(int number)
    {
        return number * number;
    }
}
```

Pour utiliser une valeur retournée à partir d'une méthode, la méthode d'appel peut utiliser l'appel de méthode proprement dit partout où une valeur du même type peut suffire. Vous pouvez également affecter la valeur de retour à une variable. Par exemple, les deux exemples de code suivants remplissent le même objectif :

C#

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

C#

```
result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));
// The result is 9.
Console.WriteLine(result);
```

L'utilisation d'une variable locale, dans cet exemple, `result`, pour stocker une valeur est facultative. Elle peut favoriser la lisibilité du code ou s'avérer nécessaire si vous avez besoin de stocker la valeur d'origine de l'argument pour la portée entière de la méthode.

Pour utiliser une valeur retournée par référence à partir d'une méthode, vous devez déclarer une variable ref local si vous avez l'intention de modifier sa valeur. Par exemple, si la méthode `Planet.GetEstimatedDistance` retourne une valeur Double par référence, vous pouvez la définir en tant que variable ref local avec du code semblable à celui-ci :

C#

```
ref double distance = ref Planet.GetEstimatedDistance();
```

Le retour d'un tableau multidimensionnel à partir d'une méthode, `M`, qui modifie le contenu du tableau n'est pas nécessaire si la fonction appelante a passé le tableau à `M`. Vous pouvez retourner le tableau obtenu à partir de `M` pour bénéficier d'un style approprié ou d'un flux fonctionnel de valeurs, mais cela n'est pas nécessaire, car C# passe tous les types de référence par valeur, et la valeur d'une référence de tableau est le pointeur qui désigne le tableau. Dans la méthode `M`, les modifications apportées au contenu du tableau sont observables par tout code ayant une référence au tableau, comme dans l'exemple suivant :

C#

```
static void Main(string[] args)
{
    int[,] matrix = new int[2, 2];
    FillMatrix(matrix);
    // matrix is now full of -1
}

public static void FillMatrix(int[,] matrix)
```

```
{
    for (int i = 0; i < matrix.GetLength(0); i++)
    {
        for (int j = 0; j < matrix.GetLength(1); j++)
        {
            matrix[i, j] = -1;
        }
    }
}
```

Méthodes Async

La fonctionnalité async vous permet d'appeler des méthodes asynchrones sans utiliser de rappels explicites ni fractionner manuellement votre code entre plusieurs méthodes ou expressions lambda.

Si vous marquez une méthode avec le modificateur async, vous pouvez utiliser l'opérateur await dans la méthode. Quand le contrôle atteint une expression await dans la méthode async, il retourne à l'appelant, et la progression dans la méthode est interrompue jusqu'à ce que la tâche attendue se termine. Quand la tâche est terminée, l'exécution peut reprendre dans la méthode.

> ⓘ **Notes**
>
> Une méthode asynchrone retourne à l'appelant quand elle rencontre le premier objet attendu qui n'est pas encore terminé ou quand elle atteint la fin de la méthode asynchrone, selon la première éventualité.

Une méthode asynchrone a généralement un type de retour Task<TResult>, Task, IAsyncEnumerable<T> ou `void` . Le type de retour `void` est essentiellement utilisé pour définir les gestionnaires d'événements, où un type de retour `void` est obligatoire. Une méthode async qui retourne `void` ne peut pas être attendue, et l'appelant d'une méthode retournant void ne peut intercepter aucune exception levée par la méthode. Une méthode asynchrone peut avoir n'importe quel type de retour de type tâche.

Dans l'exemple suivant, `DelayAsync` est une méthode async dont le type de retour est Task<TResult>. `DelayAsync` a une instruction `return` qui retourne un entier. Ainsi, la déclaration de méthode de `DelayAsync` doit avoir un type de retour de `Task<int>`. Étant donné que le type de retour est `Task<int>`, l'évaluation de l'expression `await` dans `DoSomethingAsync` produit un entier, comme le montre l'instruction suivante : `int result = await delayTask`.

La méthode `Main` est un exemple de méthode asynchrone dont le type de retour est Task. Elle accède à la méthode `DoSomethingAsync` et étant donné qu'elle est exprimée avec une seule ligne, elle peut omettre les mots clés `async` et `await`. Étant donné que `DoSomethingAsync` est une méthode async, la tâche pour l'appel à `DoSomethingAsync` doit être attendue, comme le montre l'instruction suivante : `await DoSomethingAsync();`.

C#

```
class Program
{
    static Task Main() => DoSomethingAsync();

    static async Task DoSomethingAsync()
    {
        Task<int> delayTask = DelayAsync();
        int result = await delayTask;

        // The previous two statements may be combined into
        // the following statement.
        //int result = await DelayAsync();

        Console.WriteLine($"Result: {result}");
    }

    static async Task<int> DelayAsync()
    {
        await Task.Delay(100);
        return 5;
    }
}
// Example output:
//   Result: 5
```

Une méthode async ne peut pas déclarer de paramètres ref ou out , mais elle peut appeler des méthodes qui comportent ces paramètres.

Pour plus d'informations sur les méthodes asynchrones, consultez Programmation asynchrone avec asynchrone et await et Types de retour asynchrones.

Définitions de corps d'expression

Il est courant d'avoir des définitions de méthode qui retournent tout simplement le résultat d'une expression immédiatement, ou qui ont une seule instruction en tant que corps de la méthode. Il existe un raccourci de syntaxe pour définir de telles méthodes en utilisant `=>`:

C#

```
public Point Move(int dx, int dy) => new Point(x + dx, y + dy);
public void Print() => Console.WriteLine(First + " " + Last);
// Works with operators, properties, and indexers too.
public static Complex operator +(Complex a, Complex b) => a.Add(b);
public string Name => First + " " + Last;
public Customer this[long id] => store.LookupCustomer(id);
```

Si la méthode retourne `void` ou est une méthode async, alors le corps de la méthode doit être une expression d'instruction (même chose avec les expressions lambda). En ce qui concerne les propriétés et indexeurs, ils doivent être en lecture seule et vous n'utilisez pas le mot clé d'accesseur `get`.

Iterators

Un itérateur exécute une itération personnalisée sur une collection, comme une liste ou un tableau. Un itérateur utilise l'instruction yield return pour retourner chaque élément un par un. Quand une instruction `yield return` est atteinte, l'emplacement actuel dans le code est mémorisé. L'exécution redémarre à partir de cet emplacement au prochain appel de l'itérateur.

Vous appelez un itérateur depuis le code client en utilisant une instruction foreach.

Le type de retour d'un itérateur peut être IEnumerable, IEnumerable<T>, IAsyncEnumerable<T>, IEnumerator ou IEnumerator<T>.

Pour plus d'informations, consultez Itérateurs.

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Guide de programmation C#
- Système de types C#
- Modificateurs d'accès
- Classes statiques et membres de classe statique
- Héritage
- Classes abstract et sealed et membres de classe
- params

- out
- ref
- Paramètres de méthodes

Fonctions locales (Guide de programmation C#)

Article • 26/04/2023

Les *fonctions locales* sont des méthodes d'un type qui sont imbriqués dans un autre membre. Elles ne peuvent être appelées qu'à partir de leur membre conteneur. Les fonctions locales peuvent être déclarées et appelées dans et à partir des éléments suivants :

- Méthodes, tout particulièrement les méthodes iterator et async
- Constructeurs
- Accesseurs de propriété
- Accesseurs d'événement
- Méthodes anonymes
- Expressions lambda
- Finaliseurs
- Autres fonctions locales

En revanche, les fonctions locales ne peuvent pas être déclarées à l'intérieur d'un membre expression-bodied.

> ⓘ **Notes**
>
> Dans certains cas, vous pouvez utiliser une expression lambda pour implémenter la fonctionnalité également prise en charge par une fonction locale. Pour obtenir une comparaison, consultez **Fonctions locales et expressions lambda**.

Les fonctions locales permettent de clarifier l'objectif de votre code. Quiconque lisant votre code peut voir que la méthode ne peut pas être appelée, sauf par la méthode conteneur. Pour les projets d'équipe, elles empêchent aussi à un autre développeur d'appeler par inadvertance la méthode directement à partir d'un autre emplacement dans la classe ou le struct.

Syntaxe des fonctions locales

Une fonction locale est définie en tant que méthode imbriquée à l'intérieur d'un membre conteneur. Sa définition présente la syntaxe suivante :

```
C#
```

```
<modifiers> <return-type> <method-name> <parameter-list>
```

> ⓘ Notes
>
> `<parameter-list>` ne doit pas contenir les paramètres nommés avec un **mot clé contextuel** `value`. Le compilateur crée la variable temporaire « value », qui contient les variables sortantes référencées, ce qui provoque ultérieurement une ambiguïté et peut également entraîner un comportement inattendu.

Vous pouvez utiliser les modificateurs suivants avec une fonction locale :

- async
- unsafe
- static Une fonction locale statique ne peut pas capturer les variables locales ou l'état de l'instance.
- extern Une fonction locale externe doit être `static`.

Toutes les variables locales définies dans le membre conteneur, y compris ses paramètres de méthode, sont accessibles dans une fonction locale non statique.

Contrairement à la définition d'une méthode, la définition d'une fonction locale ne peut pas inclure le modificateur d'accès au membre. Comme toutes les fonctions locales sont privées, l'inclusion d'un modificateur d'accès, tel que le mot clé `private`, génère l'erreur de compilateur CS0106 : « Le modificateur « private » n'est pas valide pour cet élément ».

L'exemple suivant définit une fonction locale nommée `AppendPathSeparator` qui est privée pour une méthode nommée `GetText` :

C#

```
private static string GetText(string path, string filename)
{
     var reader = File.OpenText($"{AppendPathSeparator(path)}{filename}");
     var text = reader.ReadToEnd();
     return text;

     string AppendPathSeparator(string filepath)
     {
        return filepath.EndsWith(@"\") ? filepath : filepath + @"\";
     }
}
```

À compter de C# 9.0, vous pouvez appliquer des attributs à une fonction locale, à ses paramètres et à ses paramètres de type, comme le montre l'exemple suivant :

C#

```
#nullable enable
private static void Process(string?[] lines, string mark)
{
    foreach (var line in lines)
    {
        if (IsValid(line))
        {
            // Processing logic...
        }
    }

    bool IsValid([NotNullWhen(true)] string? line)
    {
        return !string.IsNullOrEmpty(line) && line.Length >= mark.Length;
    }
}
```

L'exemple précédent utilise un attribut spécial pour aider le compilateur dans l'analyse statique dans un contexte pouvant accepter la valeur Null.

Fonctions locales et exceptions

L'une des caractéristiques intéressantes des fonctions locales est qu'elles permettent l'affichage immédiat des exceptions. Pour les méthodes d'itérateur, les exceptions apparaissent uniquement au moment où la séquence retournée est énumérée, et non à la récupération de l'itérateur. Pour les méthodes async, les exceptions levées dans une méthode async sont observées quand la tâche retournée est attendue.

L'exemple suivant définit une méthode `OddSequence` qui énumère les nombres impairs dans une plage spécifiée. Sachant qu'elle passe à la méthode d'énumérateur `OddSequence` un nombre supérieur à 100, la méthode lève une exception ArgumentOutOfRangeException. Comme le montre la sortie de l'exemple, l'exception apparaît uniquement au moment d'itérer les nombres, et non à la récupération de l'énumérateur.

C#

```
public class IteratorWithoutLocalExample
{
   public static void Main()
   {
```

```
      IEnumerable<int> xs = OddSequence(50, 110);
      Console.WriteLine("Retrieved enumerator...");

      foreach (var x in xs)  // line 11
      {
         Console.Write($"{x} ");
      }
   }

   public static IEnumerable<int> OddSequence(int start, int end)
   {
      if (start < 0 || start > 99)
         throw new ArgumentOutOfRangeException(nameof(start), "start must be
between 0 and 99.");
      if (end > 100)
         throw new ArgumentOutOfRangeException(nameof(end), "end must be
less than or equal to 100.");
      if (start >= end)
         throw new ArgumentException("start must be less than end.");

      for (int i = start; i <= end; i++)
      {
         if (i % 2 == 1)
            yield return i;
      }
   }
}
// The example displays the output like this:
//
//    Retrieved enumerator...
//    Unhandled exception. System.ArgumentOutOfRangeException: end must be
less than or equal to 100. (Parameter 'end')
//    at IteratorWithoutLocalExample.OddSequence(Int32 start, Int32
end)+MoveNext() in IteratorWithoutLocal.cs:line 22
//    at IteratorWithoutLocalExample.Main() in IteratorWithoutLocal.cs:line
11
```

Si vous placez une logique d'itérateur dans une fonction locale, des exceptions de validation d'argument sont levées lorsque vous récupérez l'énumérateur, comme le montre l'exemple suivant :

C#

```
public class IteratorWithLocalExample
{
   public static void Main()
   {
      IEnumerable<int> xs = OddSequence(50, 110);  // line 8
      Console.WriteLine("Retrieved enumerator...");

      foreach (var x in xs)
      {
```

```
            Console.Write($"{x} ");
        }
    }

    public static IEnumerable<int> OddSequence(int start, int end)
    {
        if (start < 0 || start > 99)
            throw new ArgumentOutOfRangeException(nameof(start), "start must be
between 0 and 99.");
        if (end > 100)
            throw new ArgumentOutOfRangeException(nameof(end), "end must be
less than or equal to 100.");
        if (start >= end)
            throw new ArgumentException("start must be less than end.");

        return GetOddSequenceEnumerator();

        IEnumerable<int> GetOddSequenceEnumerator()
        {
            for (int i = start; i <= end; i++)
            {
                if (i % 2 == 1)
                    yield return i;
            }
        }
    }
}
// The example displays the output like this:
//
//    Unhandled exception. System.ArgumentOutOfRangeException: end must be
less than or equal to 100. (Parameter 'end')
//    at IteratorWithLocalExample.OddSequence(Int32 start, Int32 end) in
IteratorWithLocal.cs:line 22
//    at IteratorWithLocalExample.Main() in IteratorWithLocal.cs:line 8
```

Fonctions locales et expressions lambda

À première vue, les fonctions locales et les expressions lambda sont très similaires. Souvent, le choix d'utiliser des expressions lambda ou des fonctions locales est une question de style et de préférences personnelles. Toutefois, il existe de réelles différences qui vous feront utiliser les unes ou les autres et que vous devez connaître.

Examinons les différences entre l'implémentation de l'algorithme factoriel avec une fonction locale et une expression lambda. Voici la version utilisant une fonction locale :

C#

```
public static int LocalFunctionFactorial(int n)
{
    return nthFactorial(n);
```

```
    int nthFactorial(int number) => number < 2
        ? 1
        : number * nthFactorial(number - 1);
}
```

Cette version utilise des expressions lambda :

C#

```
public static int LambdaFactorial(int n)
{
    Func<int, int> nthFactorial = default(Func<int, int>);

    nthFactorial = number => number < 2
        ? 1
        : number * nthFactorial(number - 1);

    return nthFactorial(n);
}
```

Dénomination

Les fonctions locales sont explicitement nommées comme des méthodes. Les expressions lambda sont des méthodes anonymes qui doivent être affectées à des variables d'un type `delegate`, généralement de type `Action` ou `Func`. Lorsque vous déclarez une fonction locale, le processus est semblable à l'écriture d'une méthode normale ; vous déclarez un type de retour et une signature de fonction.

Signatures de fonction et types d'expressions lambda

Les expressions lambda s'appuient sur le type de la variable `Action`/`Func` qui leur est attribué pour déterminer les types d'argument et de retour. Dans les fonctions locales, étant donné que la syntaxe ressemble beaucoup à l'écriture d'une méthode normale, les types d'argument et le type de retour font déjà partie de la déclaration de la fonction.

À compter de C# 10, certaines expressions lambda ont un *type naturel*, ce qui permet au compilateur d'inférer le type de retour et les types de paramètres de l'expression lambda.

Assignation définie

Les expressions lambda sont des objets qui sont déclarés et affectés au moment de l'exécution. Pour qu'une expression lambda soit utilisée, elle doit être définitivement

affectée : la variable `Action`/`Func` à laquelle elle sera affectée doit être déclarée et l'expression lambda doit lui être attribuée. Notez que `LambdaFactorial` doit déclarer et initialiser l'expression lambda `nthFactorial` avant de la définir. Si ce n'est pas le cas, cela entraîne une erreur de compilation due au fait que vous référencez `nthFactorial` avant de lui affecter une valeur.

Les fonctions locales sont définies au moment de la compilation. Comme elles ne sont pas affectées à des variables, elles peuvent être référencées à partir de n'importe quel emplacement de code **où elles se trouvent dans l'étendue**. Dans notre premier exemple `LocalFunctionFactorial`, nous pourrions déclarer notre fonction locale au-dessus ou au-dessous de l'instruction `return` et ne pas déclencher d'erreurs du compilateur.

Ces différences font que les algorithmes récursifs sont plus faciles à créer en utilisant des fonctions locales. Vous pouvez déclarer et définir une fonction locale qui s'appelle elle-même. Les expressions lambda doivent être déclarées et une valeur par défaut doit leur être affectée avant qu'elles puissent être réaffectées à un corps référençant la même expression lambda.

Implémentation en tant que délégué

Les expressions lambda sont converties en délégués lorsqu'elles sont déclarées. Les fonctions locales sont plus flexibles, car elles peuvent être écrites comme une méthode traditionnelle *ou* en tant que délégué. Les fonctions locales sont converties en délégués uniquement lorsque ***a utilisé*** en tant que délégué.

Si vous déclarez une fonction locale et la référencez uniquement en l'appelant comme une méthode, elle ne sera pas convertie en délégué.

Capture de la variable

Les règles d'affectation définies s'appliquent également à toutes les variables qui sont capturées par la fonction locale ou l'expression lambda. Le compilateur peut effectuer une analyse statique qui active des fonctions locales de manière à affecter définitivement les variables capturées dans la portée englobante. Examinez cet exemple :

C#

```
int M()
{
    int y;
    LocalFunction();
    return y;
```

```
    void LocalFunction() => y = 0;
}
```

Le compilateur peut déterminer que `LocalFunction` affecte `y` de manière définitive lorsqu'elle est appelée. Dans la mesure où `LocalFunction` est appelée avant l'instruction `return`, `y` est affecté de manière définitive à l'instruction `return`.

Notez que lorsqu'une fonction locale capture des variables dans l'étendue englobante, la fonction locale est implémentée en tant que type délégué.

Allocations de tas

En fonction de leur utilisation, les fonctions locales peuvent éviter les allocations de tas qui sont toujours nécessaires pour les expressions lambda. Si une fonction locale n'est jamais convertie en délégué, et qu'aucune des variables capturées par la fonction locale n'est capturée par d'autres expressions lambda ou fonctions locales qui sont converties en délégués, le compilateur peut éviter les allocations de tas.

Penchons-nous sur cet exemple asynchrone :

C#

```
public async Task<string> PerformLongRunningWorkLambda(string address, int
index, string name)
{
    if (string.IsNullOrWhiteSpace(address))
        throw new ArgumentException(message: "An address is required",
paramName: nameof(address));
    if (index < 0)
        throw new ArgumentOutOfRangeException(paramName: nameof(index),
message: "The index must be non-negative");
    if (string.IsNullOrWhiteSpace(name))
        throw new ArgumentException(message: "You must supply a name",
paramName: nameof(name));

    Func<Task<string>> longRunningWorkImplementation = async () =>
    {
        var interimResult = await FirstWork(address);
        var secondResult = await SecondStep(index, name);
        return $"The results are {interimResult} and {secondResult}.
Enjoy.";
    };

    return await longRunningWorkImplementation();
}
```

La fermeture de cette expression lambda contient les variables `address`, `index` et `name`. Dans le cas des fonctions locales, l'objet qui implémente la fermeture peut être un type `struct`. Ce type de struct serait transmis par référence à la fonction locale. Cette différence d'implémentation évite une allocation.

L'instanciation nécessaire pour les expressions lambda signifie des allocations de mémoire supplémentaires, qui peuvent être un facteur influençant les performances dans les chemins de code critiques au niveau du temps. Les fonctions locales n'entraînent pas cette charge supplémentaire. Dans l'exemple ci-dessus, la version de fonctions locale a deux allocations de moins que la version à expression lambda.

Si vous savez que votre fonction locale ne sera pas convertie en délégué et qu'aucune des variables capturées par elle n'est capturée par d'autres fonctions lambdas ou fonctions locales converties en délégués, vous pouvez garantir que votre fonction locale évite d'être allouée sur le tas en la déclarant en tant que fonction locale `static`.

> **Conseil**
>
> Activez la règle du style de code .NET **IDE0062** pour vous assurer que les fonctions locales sont toujours marquées `static`.

> **Notes**
>
> L'équivalent de cette méthode avec une fonction locale fait aussi appel à une classe pour la fermeture. Le fait que la fermeture d'une fonction locale soit implémentée en tant que `class` ou `struct` est un détail d'implémentation. Une fonction locale peut utiliser un type `struct` contrairement à une expression lambda qui utilise toujours un type `class`.

C#

```
public async Task<string> PerformLongRunningWork(string address, int index,
string name)
{
    if (string.IsNullOrWhiteSpace(address))
        throw new ArgumentException(message: "An address is required",
paramName: nameof(address));
    if (index < 0)
        throw new ArgumentOutOfRangeException(paramName: nameof(index),
message: "The index must be non-negative");
    if (string.IsNullOrWhiteSpace(name))
        throw new ArgumentException(message: "You must supply a name",
paramName: nameof(name));
```

```
    return await longRunningWorkImplementation();

    async Task<string> longRunningWorkImplementation()
    {
        var interimResult = await FirstWork(address);
        var secondResult = await SecondStep(index, name);
        return $"The results are {interimResult} and {secondResult}.
Enjoy.";
    }
}
```

Utilisation du mot clé `yield`

Ultime avantage non décrit dans cet exemple : les fonctions locales peuvent être implémentées en tant qu'itérateurs, en utilisant la syntaxe `yield return` pour produire une séquence de valeurs.

C#

```
public IEnumerable<string> SequenceToLowercase(IEnumerable<string> input)
{
    if (!input.Any())
    {
        throw new ArgumentException("There are no items to convert to
lowercase.");
    }

    return LowercaseIterator();

    IEnumerable<string> LowercaseIterator()
    {
        foreach (var output in input.Select(item => item.ToLower()))
        {
            yield return output;
        }
    }
}
```

L'instruction `yield return` n'est pas autorisée dans les expressions lambda. Pour plus d'informations, consultez Erreur du compilateur CS1621.

Alors que les fonctions locales peuvent sembler redondantes par rapport aux expressions lambda, elles ont en réalité des objectifs différents et des utilisations différentes. Les fonctions locales sont plus efficaces dans le cas où vous voulez écrire une fonction qui est appelée seulement dans le contexte d'une autre méthode.

spécification du langage C#

Pour plus d'informations, consultez la section Déclarations de fonctions locales de la spécification du langage C#.

Voir aussi

- Utiliser une fonction locale au lieu de lambda (règle de style IDE0039)
- Méthodes

Instructions de déclaration

Article • 21/06/2023

Une instruction de déclaration déclare une nouvelle variable locale, constante locale ou variable de référence locale. Pour déclarer une variable locale, spécifiez son type et indiquez son nom. Vous pouvez déclarer plusieurs variables du même type dans une même instruction, comme le montre l'exemple suivant :

```
C#

string greeting;
int a, b, c;
List<double> xs;
```

Dans une instruction de déclaration, vous pouvez également initialiser une variable avec sa valeur initiale :

```
C#

string greeting = "Hello";
int a = 3, b = 2, c = a + b;
List<double> xs = new();
```

Les exemples précédents spécifient explicitement le type d'une variable. Vous pouvez également laisser le compilateur déduire le type d'une variable à partir de son expression d'initialisation. Pour ce faire, utilisez le mot clé `var` au lieu du nom d'un type. Pour plus d'informations, consultez la section Variables locales implicitement typées.

Pour déclarer une constante locale, utilisez le const mot clé, comme le montre l'exemple suivant :

```
C#

const string Greeting = "Hello";
const double MinLimit = -10.0, MaxLimit = -MinLimit;
```

Lorsque vous déclarez une constante locale, vous devez également l'initialiser.

Pour plus d'informations sur les variables de référence locales, consultez la section Variables de référence.

Variables locales implicitement typées

Lorsque vous déclarez une variable locale, vous pouvez laisser le compilateur déduire le type de la variable à partir de l'expression d'initialisation. Pour ce faire, utilisez le mot clé `var` au lieu du nom d'un type :

C#

```
var greeting = "Hello";
Console.WriteLine(greeting.GetType());  // output: System.String

var a = 32;
Console.WriteLine(a.GetType());  // output: System.Int32

var xs = new List<double>();
Console.WriteLine(xs.GetType());  // output:
System.Collections.Generic.List`1[System.Double]
```

Comme le montre l'exemple précédent, les variables locales implicitement typées sont fortement typées.

> ⓘ Notes
>
> Lorsque vous utilisez `var` dans le **contexte pouvant accepter la valeur Null** activé et que le type d'une expression d'initialisation est un type référence, le compilateur déduit toujours un type de référence **pouvant accepter la valeur Null** même si le type d'une expression d'initialisation n'est pas compatible avec la valeur Null.

Une utilisation courante de `var` est l'expression d'invocation d'un constructeur. L'utilisation de `var` vous évite de répéter un nom de type dans une déclaration de variable et une instanciation d'objet, comme le montre l'exemple suivant :

C#

```
var xs = new List<int>();
```

À compter de C# 9.0, vous pouvez utiliser une expression new typée cible en guise d'alternative :

C#

```
List<int> xs = new();
List<int>? ys = new();
```

Lorsque vous utilisez des types anonymes, vous devez utiliser des variables locales implicitement typées. L'exemple suivant montre une expression de requête qui utilise un type anonyme pour contenir le nom et le numéro de téléphone d'un client :

```
C#
```

```
var fromPhoenix = from cust in customers
                  where cust.City == "Phoenix"
                  select new { cust.Name, cust.Phone };

foreach (var customer in fromPhoenix)
{
    Console.WriteLine($"Name={customer.Name}, Phone={customer.Phone}");
}
```

Dans l'exemple précédent, vous ne pouvez pas spécifier explicitement le type de la variable `fromPhoenix`. Le type est IEnumerable<T>, mais, dans ce cas, `T` est un type anonyme et vous ne pouvez pas fournir son nom. C'est pourquoi vous devez utiliser `var`. Pour la même raison, vous devez utiliser `var` lorsque vous déclarez la variable d'itération `customer` dans l'instruction `foreach`.

Pour plus d'informations sur les variables locales implicitement typées, consultez Variables locales implicitement typées.

Dans les critères spéciaux, le mot clé `var` est utilisé dans un modèle var.

Variables de référence

Lorsque vous déclarez une variable locale et que vous ajoutez le mot clé `ref` avant le type de la variable, vous déclarez une *variable de référence*, ou une variable locale `ref` :

```
C#
```

```
ref int alias = ref variable;
```

Une variable de référence est une variable qui fait référence à une autre variable, appelée « *référent* ». Autrement dit, une variable de référence est un *alias* à son référent. Lorsque vous affectez une valeur à une variable de référence, cette valeur est assignée au référent. Lorsque vous lisez la valeur d'une variable de référence, la valeur du référent est retournée. L'exemple suivant illustre ce comportement :

```
C#
```

```
int a = 1;
ref int alias = ref a;
Console.WriteLine($"(a, alias) is ({a}, {alias})");  // output: (a, alias)
is (1, 1)

a = 2;
Console.WriteLine($"(a, alias) is ({a}, {alias})");  // output: (a, alias)
is (2, 2)

alias = 3;
Console.WriteLine($"(a, alias) is ({a}, {alias})");  // output: (a, alias)
is (3, 3)
```

Utilisez l'opérateur d'assignation ref `= ref` pour changer le référent d'une variable de référence, comme le montre l'exemple suivant :

C#

```
void Display(int[] s) => Console.WriteLine(string.Join(" ", s));

int[] xs = { 0, 0, 0 };
Display(xs);

ref int element = ref xs[0];
element = 1;
Display(xs);

element = ref xs[^1];
element = 3;
Display(xs);
// Output:
// 0 0 0
// 1 0 0
// 1 0 3
```

Dans l'exemple précédent, la variable de référence `element` est initialisée en tant qu'alias au premier élément de tableau. Ensuite, elle est réassignée par référence (`ref`) pour faire référence au dernier élément de tableau.

Vous pouvez définir une variable locale `ref readonly`. Vous ne pouvez pas assigner une valeur à une variable `ref readonly`. Toutefois, vous pouvez réaffecter par référence (`ref`) une telle variable de référence, comme le montre l'exemple suivant :

C#

```
int[] xs = { 1, 2, 3 };

ref readonly int element = ref xs[0];
```

```
// element = 100;  error CS0131: The left-hand side of an assignment must be
a variable, property or indexer
Console.WriteLine(element);  // output: 1

element = ref xs[^1];
Console.WriteLine(element);  // output: 3
```

Vous pouvez affecter un retour de référence à une variable de référence, comme le montre l'exemple suivant :

C#

```
using System;

public class NumberStore
{
    private readonly int[] numbers = { 1, 30, 7, 1557, 381, 63, 1027, 2550,
511, 1023 };

    public ref int GetReferenceToMax()
    {
        ref int max = ref numbers[0];
        for (int i = 1; i < numbers.Length; i++)
        {
            if (numbers[i] > max)
            {
                max = ref numbers[i];
            }
        }
        return ref max;
    }

    public override string ToString() => string.Join(" ", numbers);
}

public static class ReferenceReturnExample
{
    public static void Run()
    {
        var store = new NumberStore();
        Console.WriteLine($"Original sequence: {store.ToString()}");

        ref int max = ref store.GetReferenceToMax();
        max = 0;
        Console.WriteLine($"Updated sequence:  {store.ToString()}");
        // Output:
        // Original sequence: 1 30 7 1557 381 63 1027 2550 511 1023
        // Updated sequence:  1 30 7 1557 381 63 1027 0 511 1023
    }
}
```

Dans l'exemple précédent, la méthode `GetReferenceToMax` est une méthode *returns-by-ref*. Elle ne retourne pas la valeur maximale elle-même, mais un retour de référence qui est un alias à l'élément de tableau qui contient la valeur maximale. La méthode `Run` affecte un retour de référence à la variable de référence `max`. Ensuite, en affectant une valeur à `max`, elle met à jour le stockage interne de l'instance `store`. Vous pouvez également définir une méthode `ref readonly`. Les appelants d'une méthode `ref readonly` ne peuvent pas affecter une valeur à son retour de référence.

La variable d'itération de l'instruction `foreach` peut être une variable de référence. Pour plus d'informations, consultez la section foreach de l'article Instructions d'itération.

Dans les scénarios critiques pour les performances, l'utilisation de variables et de retours de référence peut améliorer les performances en évitant les opérations de copie potentiellement coûteuses.

Le compilateur garantit qu'une variable de référence ne vit pas plus longtemps que son référent et reste valide pendant toute sa durée de vie. Pour plus d'informations, consultez la section Contextes de sécurité de référence de la Spécification du langage C#.

Pour plus d'informations sur les champs `ref`, consultez la section Champs ref de l'article Types de structure ref.

scoped ref

Le mot clé contextuel `scoped` limite la durée de vie d'une valeur. Le modificateur `scoped` limite la durée de vie de *ref-safe-to-escape* ou *safe-to-escape*, respectivement, à la méthode actuelle. En fait, l'ajout du modificateur `scoped` déclare que votre code n'allongera pas la durée de vie de la variable.

Vous pouvez appliquer `scoped` à un paramètre ou à une variable locale. Le modificateur `scoped` peut être appliqué aux paramètres et aux variables locales quand le type est un ref struct. Sinon, le modificateur `scoped` peut être appliqué uniquement aux variables de référence locales. Cela inclut les variables locales déclarées avec le modificateur `ref` et les paramètres déclarés avec les modificateurs ou `in`, `ref` ou `out`.

Le modificateur `scoped` est implicitement ajouté à `this` dans les méthodes déclarées dans un `struct`, des paramètres `out` et des paramètres `ref` quand le type est un `ref struct`.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Instructions de déclaration
- Variables et retours de référence

Pour plus d'informations sur le modificateur `scoped`, consultez la note de proposition Améliorations des structures de bas niveau.

Voir aussi

- Informations de référence sur C#
- Initialiseurs d'objets et de collections
- ref, mot clé
- Réduire les allocations de mémoire à l'aide de nouvelles fonctionnalités C#
- Préférences « var » (règles de style IDE0007 et IDE0008)

Paramètres de méthode (référence C#)

Article • 07/04/2023

En C#, les arguments peuvent être passés aux paramètres par valeur ou par référence. N'oubliez pas que les types C# peuvent être des types référence (`class`) ou des types valeur (`struct`) :

- *Passer par valeur* signifie **passer une copie de la variable** à la méthode.
- *Passer par référence* signifie **passer l'accès à la variable** à la méthode.
- Une variable d'un *type référence* contient une référence à ses données.
- Une variable d'un *type valeur* contient ses données directement.

Étant donné qu'un struct est un type valeur, quand vous passez un struct par valeur à une méthode, la méthode reçoit et traite une copie de l'argument struct. La méthode n'a pas accès au struct d'origine dans la méthode d'appel, et ne peut donc pas le modifier de quelque manière que ce soit. La méthode peut modifier uniquement la copie.

Une instance de classe est un type référence, pas un type valeur. Quand un type référence est passé par valeur à une méthode, celle-ci reçoit une copie de la référence à l'instance de classe. Autrement dit, la méthode appelée reçoit une copie de l'adresse de l'instance, et la méthode appelante conserve l'adresse d'origine de l'instance. L'instance de classe dans la méthode d'appel a une adresse, le paramètre dans la méthode appelée a une copie de l'adresse, et les deux adresses font référence au même objet. Étant donné que le paramètre contient uniquement une copie de l'adresse, la méthode appelée ne peut pas modifier l'adresse de l'instance de la classe dans la méthode. Toutefois, la méthode appelée peut utiliser la copie de l'adresse pour accéder aux membres de la classe qui ont à la fois l'adresse d'origine et la copie de la référence d'adresse. Si la méthode appelée modifie un membre de classe, l'instance de classe d'origine dans la méthode d'appel change également.

La sortie de l'exemple suivant illustre la différence. La valeur du champ `willIChange` de l'instance de classe est changée par l'appel à la méthode `ClassTaker`, car la méthode utilise l'adresse mentionnée dans le paramètre pour rechercher le champ spécifié de l'instance de classe. Le champ `willIChange` du struct de la méthode d'appel n'est pas changé par l'appel à la méthode `StructTaker`, car la valeur de l'argument est une copie du struct lui-même, et non pas une copie de son adresse. `StructTaker` change la copie, et cette dernière est perdue quand l'appel à `StructTaker` est terminé.

```
C#
```

```
class TheClass
{
    public string? willIChange;
}

struct TheStruct
{
    public string willIChange;
}

class TestClassAndStruct
{
    static void ClassTaker(TheClass c)
    {
        c.willIChange = "Changed";
    }

    static void StructTaker(TheStruct s)
    {
        s.willIChange = "Changed";
    }

    public static void Main()
    {
        TheClass testClass = new TheClass();
        TheStruct testStruct = new TheStruct();

        testClass.willIChange = "Not Changed";
        testStruct.willIChange = "Not Changed";

        ClassTaker(testClass);
        StructTaker(testStruct);

        Console.WriteLine("Class field = {0}", testClass.willIChange);
        Console.WriteLine("Struct field = {0}", testStruct.willIChange);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Class field = Changed
    Struct field = Not Changed
*/
```

Les modifications apportées à l'argument qui sont visibles à partir de l'appelant dépendent de la façon dont un argument est passé et de s'il s'agit d'un type référence ou d'un type valeur.

Passer un type valeur par valeur

Lorsque vous passez un type *valeurpar valeur* :

- Si la méthode affecte le paramètre pour faire référence à un autre objet, ces modifications **ne sont pas** visibles à partir de l'appelant.
- Si la méthode modifie l'état de l'objet auquel le paramètre fait référence, ces modifications **ne sont pas** visibles par l'appelant.

L'exemple suivant illustre le passage de paramètres de type valeur par valeur. La variable `n` est passée par valeur à la méthode `SquareIt`. Toutes les modifications qui ont lieu à l'intérieur de la méthode n'ont aucun effet sur la valeur d'origine de la variable.

C#

```
int n = 5;
System.Console.WriteLine("The value before calling the method: {0}", n);

SquareIt(n);  // Passing the variable by value.
System.Console.WriteLine("The value after calling the method: {0}", n);

// Keep the console window open in debug mode.
System.Console.WriteLine("Press any key to exit.");
System.Console.ReadKey();

static void SquareIt(int x)
// The parameter x is passed by value.
// Changes to x will not affect the original value of x.
{
    x *= x;
    System.Console.WriteLine("The value inside the method: {0}", x);
}
/* Output:
    The value before calling the method: 5
    The value inside the method: 25
    The value after calling the method: 5
*/
```

La variable `n` est un type valeur. Elle contient ses données, la valeur `5`. Quand `SquareIt` est appelée, le contenu de `n` est copié dans le paramètre `x`, qui est mis au carré à l'intérieur de la méthode. Dans `Main`, toutefois, la valeur de `n` est identique avant et après l'appel de la méthode `SquareIt`. La modification qui intervient à l'intérieur de la méthode affecte uniquement la variable locale `x`.

Passer un type valeur par référence

Lorsque vous transmettez un type *valeurpar référence* :

- Si la méthode affecte le paramètre pour faire référence à un autre objet, ces modifications **ne sont pas** visibles à partir de l'appelant.
- Si la méthode modifie l'état de l'objet auquel le paramètre fait référence, ces modifications **sont visibles** à partir de l'appelant.

L'exemple suivant est identique au précédent, sauf que l'argument est passé en tant que paramètre `ref`. La valeur de l'argument sous-jacent, `n`, est modifiée quand `x` est modifiée dans la méthode.

C#

```
int n = 5;
System.Console.WriteLine("The value before calling the method: {0}", n);

SquareIt(ref n);  // Passing the variable by reference.
System.Console.WriteLine("The value after calling the method: {0}", n);

// Keep the console window open in debug mode.
System.Console.WriteLine("Press any key to exit.");
System.Console.ReadKey();

static void SquareIt(ref int x)
// The parameter x is passed by reference.
// Changes to x will affect the original value of x.
{
    x *= x;
    System.Console.WriteLine("The value inside the method: {0}", x);
}
/* Output:
    The value before calling the method: 5
    The value inside the method: 25
    The value after calling the method: 25
*/
```

Dans cet exemple, ce n'est pas la valeur de `n` qui est passée, mais plutôt une référence à `n`. Le paramètre `x` n'est pas un int ; il s'agit d'une référence à un `int`, dans le cas présent une référence à `n`. Ainsi, quand `x` est mis au carré à l'intérieur de la méthode, ce qui est réellement mis au carré est ce à quoi `x` fait référence, c'est-à-dire `n`.

Passer un type référence par valeur

Lorsque vous passez un type *référencepar valeur* :

- Si la méthode affecte le paramètre pour faire référence à un autre objet, ces modifications **ne sont pas** visibles à partir de l'appelant.

- Si la méthode modifie l'état de l'objet auquel le paramètre fait référence, ces modifications **sont visibles** à partir de l'appelant.

L'exemple suivant illustre le passage d'un paramètre de type référence (`arr`) par valeur, à une méthode (`Change`). Sachant que le paramètre est une référence à `arr`, il est possible de modifier les valeurs des éléments du tableau. Cependant, la tentative de réassignation du paramètre à un emplacement de mémoire différent n'aboutit qu'à l'intérieur de la méthode et n'affecte pas la variable d'origine, `arr`.

C#

```
int[] arr = { 1, 4, 5 };
System.Console.WriteLine("Inside Main, before calling the method, the first
element is: {0}", arr[0]);

Change(arr);
System.Console.WriteLine("Inside Main, after calling the method, the first
element is: {0}", arr[0]);

static void Change(int[] pArray)
{
    pArray[0] = 888;  // This change affects the original element.
    pArray = new int[5] { -3, -1, -2, -3, -4 };   // This change is local.
    System.Console.WriteLine("Inside the method, the first element is: {0}",
pArray[0]);
}
/* Output:
    Inside Main, before calling the method, the first element is: 1
    Inside the method, the first element is: -3
    Inside Main, after calling the method, the first element is: 888
*/
```

Dans l'exemple précédent, le tableau `arr`, qui est un type référence, est passé à la méthode sans le paramètre `ref`. Dans ce cas, une copie de la référence, qui pointe vers `arr`, est passée à la méthode. La sortie montre que la méthode peut modifier le contenu d'un élément du tableau, dans ce cas de `1` à `888`. Cependant, si vous allouez une nouvelle portion de mémoire via l'opérateur new à l'intérieur de la méthode `Change`, la variable `pArray` fait référence à un nouveau tableau. Par conséquent, les modifications apportées à la suite de cette opération n'ont pas d'effet sur le tableau d'origine (`arr`), qui est créé à l'intérieur de `Main`. En fait, deux tableaux sont créés dans cet exemple : un à l'intérieur de `Main` et un autre à l'intérieur de la méthode `Change`.

Passer un type référence par référence

Lorsque vous passez un type *référencepar référence* :

- Si la méthode affecte le paramètre pour faire référence à un autre objet, ces modifications **sont visibles** à partir de l'appelant.
- Si la méthode modifie l'état de l'objet auquel le paramètre fait référence, ces modifications **sont visibles** à partir de l'appelant.

L'exemple suivant est identique au précédent, sauf que le mot clé `ref` est ajouté à l'en-tête et à l'appel de la méthode. Les modifications qui se produisent dans la méthode affectent la variable d'origine dans le programme appelant.

C#

```
int[] arr = { 1, 4, 5 };
System.Console.WriteLine("Inside Main, before calling the method, the first
element is: {0}", arr[0]);

Change(ref arr);
System.Console.WriteLine("Inside Main, after calling the method, the first
element is: {0}", arr[0]);

static void Change(ref int[] pArray)
{
    // Both of the following changes will affect the original variables:
    pArray[0] = 888;
    pArray = new int[5] { -3, -1, -2, -3, -4 };
    System.Console.WriteLine("Inside the method, the first element is: {0}",
pArray[0]);
}
/* Output:
    Inside Main, before calling the method, the first element is: 1
    Inside the method, the first element is: -3
    Inside Main, after calling the method, the first element is: -3
*/
```

Les modifications qui se produisent à l'intérieur de la méthode affectent le tableau d'origine dans `Main`. En fait, le tableau d'origine est réalloué à l'aide de l'opérateur `new`. Par conséquent, après l'appel à la méthode `Change`, toute référence à `arr` pointe vers le tableau à cinq éléments, qui est créé dans la méthode `Change`.

Étendue des références et des valeurs

Les méthodes peuvent stocker les valeurs des paramètres dans des champs. Lorsque les paramètres sont passés par valeur, cela est toujours fait de façon sécurisée. Les valeurs sont copiées et les types de référence sont accessibles lorsqu'ils sont stockés dans un champ. La transmission de paramètres par référence de façon sûre nécessite que le compilateur définisse quand il est sûr d'affecter une référence à une nouvelle variable. Pour chaque expression, le compilateur définit une *étendue* qui limite l'accès à une

expression ou à une variable. Le compilateur utilise deux étendues : *safe_to_escape* et *ref_safe_to_escape*.

- L'étendue *safe_to_escape* définit l'étendue dans laquelle n'importe quelle expression est accessible en toute sécurité.
- L'étendue *ref_safe_to_escape* définit l'étendue dans laquelle une *référence* à n'importe quelle expression peut être accessible ou modifiée en toute sécurité.

De manière informelle, vous pouvez considérer ces étendues comme le mécanisme permettant de vous assurer que votre code n'accède jamais ou ne modifie jamais une référence qui n'est plus valide. Une référence est valide tant qu'elle fait référence à un objet ou à un struct valide. L'étendue *safe_to_escape* définit quand une variable peut être affectée ou réaffectée. L'étendue *ref_safe_to_escape* définit quand une variable peut être affectée par *référence* ou réaffectée par *référence*. L'affectation affecte une variable à une nouvelle valeur ; *l'affectation par référence* affecte la variable pour *faire référence à* un autre emplacement de stockage.

Modificateurs

Les paramètres déclarés pour une méthode sans in, ref ou out sont passés à la méthode appelée par valeur. Les modificateurs `ref`, `in` et `out` diffèrent dans les règles d'affectation :

- L'argument d'un paramètre `ref` doit être définitivement affecté. La méthode appelée peut réaffecter ce paramètre.
- L'argument d'un paramètre `in` doit être définitivement affecté. La méthode appelée ne peut pas réaffecter ce paramètre.
- L'argument d'un paramètre `out` n'a pas besoin d'être définitivement affecté. La méthode appelée doit affecter le paramètre.

Cette section décrit les mots clés que vous pouvez utiliser pour déclarer des paramètres de méthode :

- params spécifie que ce paramètre peut prendre un nombre variable d'arguments.
- in spécifie que ce paramètre est passé par référence, mais qu'il est lu uniquement par la méthode appelée.
- ref spécifie que ce paramètre est passé par référence et qu'il peut être lu ou écrit par la méthode appelée.
- out spécifie que ce paramètre est passé par référence et qu'il est écrit par la méthode appelée.

Voir aussi

- Référence C#
- Mots clés C#
- Liste des arguments dans la Spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Paramètres de méthode (référence C#)

Article • 07/04/2023

En C#, les arguments peuvent être passés aux paramètres par valeur ou par référence. N'oubliez pas que les types C# peuvent être des types référence (`class`) ou des types valeur (`struct`) :

- *Passer par valeur* signifie **passer une copie de la variable** à la méthode.
- *Passer par référence* signifie **passer l'accès à la variable** à la méthode.
- Une variable d'un *type référence* contient une référence à ses données.
- Une variable d'un *type valeur* contient ses données directement.

Étant donné qu'un struct est un type valeur, quand vous passez un struct par valeur à une méthode, la méthode reçoit et traite une copie de l'argument struct. La méthode n'a pas accès au struct d'origine dans la méthode d'appel, et ne peut donc pas le modifier de quelque manière que ce soit. La méthode peut modifier uniquement la copie.

Une instance de classe est un type référence, pas un type valeur. Quand un type référence est passé par valeur à une méthode, celle-ci reçoit une copie de la référence à l'instance de classe. Autrement dit, la méthode appelée reçoit une copie de l'adresse de l'instance, et la méthode appelante conserve l'adresse d'origine de l'instance. L'instance de classe dans la méthode d'appel a une adresse, le paramètre dans la méthode appelée a une copie de l'adresse, et les deux adresses font référence au même objet. Étant donné que le paramètre contient uniquement une copie de l'adresse, la méthode appelée ne peut pas modifier l'adresse de l'instance de la classe dans la méthode. Toutefois, la méthode appelée peut utiliser la copie de l'adresse pour accéder aux membres de la classe qui ont à la fois l'adresse d'origine et la copie de la référence d'adresse. Si la méthode appelée modifie un membre de classe, l'instance de classe d'origine dans la méthode d'appel change également.

La sortie de l'exemple suivant illustre la différence. La valeur du champ `willIChange` de l'instance de classe est changée par l'appel à la méthode `ClassTaker`, car la méthode utilise l'adresse mentionnée dans le paramètre pour rechercher le champ spécifié de l'instance de classe. Le champ `willIChange` du struct de la méthode d'appel n'est pas changé par l'appel à la méthode `StructTaker`, car la valeur de l'argument est une copie du struct lui-même, et non pas une copie de son adresse. `StructTaker` change la copie, et cette dernière est perdue quand l'appel à `StructTaker` est terminé.

```
C#
```

```
class TheClass
{
    public string? willIChange;
}

struct TheStruct
{
    public string willIChange;
}

class TestClassAndStruct
{
    static void ClassTaker(TheClass c)
    {
        c.willIChange = "Changed";
    }

    static void StructTaker(TheStruct s)
    {
        s.willIChange = "Changed";
    }

    public static void Main()
    {
        TheClass testClass = new TheClass();
        TheStruct testStruct = new TheStruct();

        testClass.willIChange = "Not Changed";
        testStruct.willIChange = "Not Changed";

        ClassTaker(testClass);
        StructTaker(testStruct);

        Console.WriteLine("Class field = {0}", testClass.willIChange);
        Console.WriteLine("Struct field = {0}", testStruct.willIChange);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Class field = Changed
    Struct field = Not Changed
*/
```

Les modifications apportées à l'argument qui sont visibles à partir de l'appelant dépendent de la façon dont un argument est passé et de s'il s'agit d'un type référence ou d'un type valeur.

Passer un type valeur par valeur

Lorsque vous passez un type *valeurpar valeur* :

- Si la méthode affecte le paramètre pour faire référence à un autre objet, ces modifications **ne sont pas** visibles à partir de l'appelant.
- Si la méthode modifie l'état de l'objet auquel le paramètre fait référence, ces modifications **ne sont pas** visibles par l'appelant.

L'exemple suivant illustre le passage de paramètres de type valeur par valeur. La variable `n` est passée par valeur à la méthode `SquareIt`. Toutes les modifications qui ont lieu à l'intérieur de la méthode n'ont aucun effet sur la valeur d'origine de la variable.

C#

```
int n = 5;
System.Console.WriteLine("The value before calling the method: {0}", n);

SquareIt(n);  // Passing the variable by value.
System.Console.WriteLine("The value after calling the method: {0}", n);

// Keep the console window open in debug mode.
System.Console.WriteLine("Press any key to exit.");
System.Console.ReadKey();

static void SquareIt(int x)
// The parameter x is passed by value.
// Changes to x will not affect the original value of x.
{
    x *= x;
    System.Console.WriteLine("The value inside the method: {0}", x);
}
/* Output:
    The value before calling the method: 5
    The value inside the method: 25
    The value after calling the method: 5
*/
```

La variable `n` est un type valeur. Elle contient ses données, la valeur `5`. Quand `SquareIt` est appelée, le contenu de `n` est copié dans le paramètre `x`, qui est mis au carré à l'intérieur de la méthode. Dans `Main`, toutefois, la valeur de `n` est identique avant et après l'appel de la méthode `SquareIt`. La modification qui intervient à l'intérieur de la méthode affecte uniquement la variable locale `x`.

Passer un type valeur par référence

Lorsque vous transmettez un type *valeurpar référence* :

- Si la méthode affecte le paramètre pour faire référence à un autre objet, ces modifications **ne sont pas** visibles à partir de l'appelant.
- Si la méthode modifie l'état de l'objet auquel le paramètre fait référence, ces modifications **sont visibles** à partir de l'appelant.

L'exemple suivant est identique au précédent, sauf que l'argument est passé en tant que paramètre `ref`. La valeur de l'argument sous-jacent, `n`, est modifiée quand `x` est modifiée dans la méthode.

C#

```
int n = 5;
System.Console.WriteLine("The value before calling the method: {0}", n);

SquareIt(ref n);  // Passing the variable by reference.
System.Console.WriteLine("The value after calling the method: {0}", n);

// Keep the console window open in debug mode.
System.Console.WriteLine("Press any key to exit.");
System.Console.ReadKey();

static void SquareIt(ref int x)
// The parameter x is passed by reference.
// Changes to x will affect the original value of x.
{
    x *= x;
    System.Console.WriteLine("The value inside the method: {0}", x);
}
/* Output:
    The value before calling the method: 5
    The value inside the method: 25
    The value after calling the method: 25
*/
```

Dans cet exemple, ce n'est pas la valeur de `n` qui est passée, mais plutôt une référence à `n`. Le paramètre `x` n'est pas un int ; il s'agit d'une référence à un `int`, dans le cas présent une référence à `n`. Ainsi, quand `x` est mis au carré à l'intérieur de la méthode, ce qui est réellement mis au carré est ce à quoi `x` fait référence, c'est-à-dire `n`.

Passer un type référence par valeur

Lorsque vous passez un type *référencepar valeur* :

- Si la méthode affecte le paramètre pour faire référence à un autre objet, ces modifications **ne sont pas** visibles à partir de l'appelant.

- Si la méthode modifie l'état de l'objet auquel le paramètre fait référence, ces modifications **sont visibles** à partir de l'appelant.

L'exemple suivant illustre le passage d'un paramètre de type référence (`arr`) par valeur, à une méthode (`Change`). Sachant que le paramètre est une référence à `arr`, il est possible de modifier les valeurs des éléments du tableau. Cependant, la tentative de réassignation du paramètre à un emplacement de mémoire différent n'aboutit qu'à l'intérieur de la méthode et n'affecte pas la variable d'origine, `arr`.

C#

```
int[] arr = { 1, 4, 5 };
System.Console.WriteLine("Inside Main, before calling the method, the first
element is: {0}", arr[0]);

Change(arr);
System.Console.WriteLine("Inside Main, after calling the method, the first
element is: {0}", arr[0]);

static void Change(int[] pArray)
{
    pArray[0] = 888;  // This change affects the original element.
    pArray = new int[5] { -3, -1, -2, -3, -4 };   // This change is local.
    System.Console.WriteLine("Inside the method, the first element is: {0}",
pArray[0]);
}
/* Output:
    Inside Main, before calling the method, the first element is: 1
    Inside the method, the first element is: -3
    Inside Main, after calling the method, the first element is: 888
*/
```

Dans l'exemple précédent, le tableau `arr`, qui est un type référence, est passé à la méthode sans le paramètre `ref`. Dans ce cas, une copie de la référence, qui pointe vers `arr`, est passée à la méthode. La sortie montre que la méthode peut modifier le contenu d'un élément du tableau, dans ce cas de `1` à `888`. Cependant, si vous allouez une nouvelle portion de mémoire via l'opérateur new à l'intérieur de la méthode `Change`, la variable `pArray` fait référence à un nouveau tableau. Par conséquent, les modifications apportées à la suite de cette opération n'ont pas d'effet sur le tableau d'origine (`arr`), qui est créé à l'intérieur de `Main`. En fait, deux tableaux sont créés dans cet exemple : un à l'intérieur de `Main` et un autre à l'intérieur de la méthode `Change`.

Passer un type référence par référence

Lorsque vous passez un type *référencepar référence* :

- Si la méthode affecte le paramètre pour faire référence à un autre objet, ces modifications **sont visibles** à partir de l'appelant.
- Si la méthode modifie l'état de l'objet auquel le paramètre fait référence, ces modifications **sont visibles** à partir de l'appelant.

L'exemple suivant est identique au précédent, sauf que le mot clé `ref` est ajouté à l'en-tête et à l'appel de la méthode. Les modifications qui se produisent dans la méthode affectent la variable d'origine dans le programme appelant.

C#

```
int[] arr = { 1, 4, 5 };
System.Console.WriteLine("Inside Main, before calling the method, the first
element is: {0}", arr[0]);

Change(ref arr);
System.Console.WriteLine("Inside Main, after calling the method, the first
element is: {0}", arr[0]);

static void Change(ref int[] pArray)
{
    // Both of the following changes will affect the original variables:
    pArray[0] = 888;
    pArray = new int[5] { -3, -1, -2, -3, -4 };
    System.Console.WriteLine("Inside the method, the first element is: {0}",
pArray[0]);
}
/* Output:
    Inside Main, before calling the method, the first element is: 1
    Inside the method, the first element is: -3
    Inside Main, after calling the method, the first element is: -3
*/
```

Les modifications qui se produisent à l'intérieur de la méthode affectent le tableau d'origine dans `Main`. En fait, le tableau d'origine est réalloué à l'aide de l'opérateur `new`. Par conséquent, après l'appel à la méthode `Change`, toute référence à `arr` pointe vers le tableau à cinq éléments, qui est créé dans la méthode `Change`.

Étendue des références et des valeurs

Les méthodes peuvent stocker les valeurs des paramètres dans des champs. Lorsque les paramètres sont passés par valeur, cela est toujours fait de façon sécurisée. Les valeurs sont copiées et les types de référence sont accessibles lorsqu'ils sont stockés dans un champ. La transmission de paramètres par référence de façon sûre nécessite que le compilateur définisse quand il est sûr d'affecter une référence à une nouvelle variable. Pour chaque expression, le compilateur définit une *étendue* qui limite l'accès à une

expression ou à une variable. Le compilateur utilise deux étendues : *safe_to_escape* et *ref_safe_to_escape*.

- L'étendue *safe_to_escape* définit l'étendue dans laquelle n'importe quelle expression est accessible en toute sécurité.
- L'étendue *ref_safe_to_escape* définit l'étendue dans laquelle une *référence* à n'importe quelle expression peut être accessible ou modifiée en toute sécurité.

De manière informelle, vous pouvez considérer ces étendues comme le mécanisme permettant de vous assurer que votre code n'accède jamais ou ne modifie jamais une référence qui n'est plus valide. Une référence est valide tant qu'elle fait référence à un objet ou à un struct valide. L'étendue *safe_to_escape* définit quand une variable peut être affectée ou réaffectée. L'étendue *ref_safe_to_escape* définit quand une variable peut être affectée par *référence* ou réaffectée par *référence*. L'affectation affecte une variable à une nouvelle valeur ; *l'affectation par référence* affecte la variable pour *faire référence à* un autre emplacement de stockage.

Modificateurs

Les paramètres déclarés pour une méthode sans in, ref ou out sont passés à la méthode appelée par valeur. Les modificateurs `ref`, `in` et `out` diffèrent dans les règles d'affectation :

- L'argument d'un paramètre `ref` doit être définitivement affecté. La méthode appelée peut réaffecter ce paramètre.
- L'argument d'un paramètre `in` doit être définitivement affecté. La méthode appelée ne peut pas réaffecter ce paramètre.
- L'argument d'un paramètre `out` n'a pas besoin d'être définitivement affecté. La méthode appelée doit affecter le paramètre.

Cette section décrit les mots clés que vous pouvez utiliser pour déclarer des paramètres de méthode :

- params spécifie que ce paramètre peut prendre un nombre variable d'arguments.
- in spécifie que ce paramètre est passé par référence, mais qu'il est lu uniquement par la méthode appelée.
- ref spécifie que ce paramètre est passé par référence et qu'il peut être lu ou écrit par la méthode appelée.
- out spécifie que ce paramètre est passé par référence et qu'il est écrit par la méthode appelée.

Voir aussi

- Référence C#
- Mots clés C#
- Liste des arguments dans la Spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Paramètres de méthode (référence C#)

Article • 07/04/2023

En C#, les arguments peuvent être passés aux paramètres par valeur ou par référence. N'oubliez pas que les types C# peuvent être des types référence (`class`) ou des types valeur (`struct`) :

- *Passer par valeur* signifie **passer une copie de la variable** à la méthode.
- *Passer par référence* signifie **passer l'accès à la variable** à la méthode.
- Une variable d'un *type référence* contient une référence à ses données.
- Une variable d'un *type valeur* contient ses données directement.

Étant donné qu'un struct est un type valeur, quand vous passez un struct par valeur à une méthode, la méthode reçoit et traite une copie de l'argument struct. La méthode n'a pas accès au struct d'origine dans la méthode d'appel, et ne peut donc pas le modifier de quelque manière que ce soit. La méthode peut modifier uniquement la copie.

Une instance de classe est un type référence, pas un type valeur. Quand un type référence est passé par valeur à une méthode, celle-ci reçoit une copie de la référence à l'instance de classe. Autrement dit, la méthode appelée reçoit une copie de l'adresse de l'instance, et la méthode appelante conserve l'adresse d'origine de l'instance. L'instance de classe dans la méthode d'appel a une adresse, le paramètre dans la méthode appelée a une copie de l'adresse, et les deux adresses font référence au même objet. Étant donné que le paramètre contient uniquement une copie de l'adresse, la méthode appelée ne peut pas modifier l'adresse de l'instance de la classe dans la méthode. Toutefois, la méthode appelée peut utiliser la copie de l'adresse pour accéder aux membres de la classe qui ont à la fois l'adresse d'origine et la copie de la référence d'adresse. Si la méthode appelée modifie un membre de classe, l'instance de classe d'origine dans la méthode d'appel change également.

La sortie de l'exemple suivant illustre la différence. La valeur du champ `willIChange` de l'instance de classe est changée par l'appel à la méthode `ClassTaker`, car la méthode utilise l'adresse mentionnée dans le paramètre pour rechercher le champ spécifié de l'instance de classe. Le champ `willIChange` du struct de la méthode d'appel n'est pas changé par l'appel à la méthode `StructTaker`, car la valeur de l'argument est une copie du struct lui-même, et non pas une copie de son adresse. `StructTaker` change la copie, et cette dernière est perdue quand l'appel à `StructTaker` est terminé.

```
C#
```

```
class TheClass
{
    public string? willIChange;
}

struct TheStruct
{
    public string willIChange;
}

class TestClassAndStruct
{
    static void ClassTaker(TheClass c)
    {
        c.willIChange = "Changed";
    }

    static void StructTaker(TheStruct s)
    {
        s.willIChange = "Changed";
    }

    public static void Main()
    {
        TheClass testClass = new TheClass();
        TheStruct testStruct = new TheStruct();

        testClass.willIChange = "Not Changed";
        testStruct.willIChange = "Not Changed";

        ClassTaker(testClass);
        StructTaker(testStruct);

        Console.WriteLine("Class field = {0}", testClass.willIChange);
        Console.WriteLine("Struct field = {0}", testStruct.willIChange);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Class field = Changed
    Struct field = Not Changed
*/
```

Les modifications apportées à l'argument qui sont visibles à partir de l'appelant dépendent de la façon dont un argument est passé et de s'il s'agit d'un type référence ou d'un type valeur.

Passer un type valeur par valeur

Lorsque vous passez un type *valeurpar valeur* :

- Si la méthode affecte le paramètre pour faire référence à un autre objet, ces modifications **ne sont pas** visibles à partir de l'appelant.
- Si la méthode modifie l'état de l'objet auquel le paramètre fait référence, ces modifications **ne sont pas** visibles par l'appelant.

L'exemple suivant illustre le passage de paramètres de type valeur par valeur. La variable `n` est passée par valeur à la méthode `SquareIt`. Toutes les modifications qui ont lieu à l'intérieur de la méthode n'ont aucun effet sur la valeur d'origine de la variable.

C#

```
int n = 5;
System.Console.WriteLine("The value before calling the method: {0}", n);

SquareIt(n);  // Passing the variable by value.
System.Console.WriteLine("The value after calling the method: {0}", n);

// Keep the console window open in debug mode.
System.Console.WriteLine("Press any key to exit.");
System.Console.ReadKey();

static void SquareIt(int x)
// The parameter x is passed by value.
// Changes to x will not affect the original value of x.
{
    x *= x;
    System.Console.WriteLine("The value inside the method: {0}", x);
}
/* Output:
    The value before calling the method: 5
    The value inside the method: 25
    The value after calling the method: 5
*/
```

La variable `n` est un type valeur. Elle contient ses données, la valeur `5`. Quand `SquareIt` est appelée, le contenu de `n` est copié dans le paramètre `x`, qui est mis au carré à l'intérieur de la méthode. Dans `Main`, toutefois, la valeur de `n` est identique avant et après l'appel de la méthode `SquareIt`. La modification qui intervient à l'intérieur de la méthode affecte uniquement la variable locale `x`.

Passer un type valeur par référence

Lorsque vous transmettez un type *valeurpar référence* :

- Si la méthode affecte le paramètre pour faire référence à un autre objet, ces modifications **ne sont pas** visibles à partir de l'appelant.
- Si la méthode modifie l'état de l'objet auquel le paramètre fait référence, ces modifications **sont visibles** à partir de l'appelant.

L'exemple suivant est identique au précédent, sauf que l'argument est passé en tant que paramètre `ref`. La valeur de l'argument sous-jacent, `n`, est modifiée quand `x` est modifiée dans la méthode.

C#

```
int n = 5;
System.Console.WriteLine("The value before calling the method: {0}", n);

SquareIt(ref n);  // Passing the variable by reference.
System.Console.WriteLine("The value after calling the method: {0}", n);

// Keep the console window open in debug mode.
System.Console.WriteLine("Press any key to exit.");
System.Console.ReadKey();

static void SquareIt(ref int x)
// The parameter x is passed by reference.
// Changes to x will affect the original value of x.
{
    x *= x;
    System.Console.WriteLine("The value inside the method: {0}", x);
}
/* Output:
    The value before calling the method: 5
    The value inside the method: 25
    The value after calling the method: 25
*/
```

Dans cet exemple, ce n'est pas la valeur de `n` qui est passée, mais plutôt une référence à `n`. Le paramètre `x` n'est pas un int ; il s'agit d'une référence à un `int`, dans le cas présent une référence à `n`. Ainsi, quand `x` est mis au carré à l'intérieur de la méthode, ce qui est réellement mis au carré est ce à quoi `x` fait référence, c'est-à-dire `n`.

Passer un type référence par valeur

Lorsque vous passez un type *référencepar valeur* :

- Si la méthode affecte le paramètre pour faire référence à un autre objet, ces modifications **ne sont pas** visibles à partir de l'appelant.

- Si la méthode modifie l'état de l'objet auquel le paramètre fait référence, ces modifications **sont visibles** à partir de l'appelant.

L'exemple suivant illustre le passage d'un paramètre de type référence (`arr`) par valeur, à une méthode (`Change`). Sachant que le paramètre est une référence à `arr`, il est possible de modifier les valeurs des éléments du tableau. Cependant, la tentative de réassignation du paramètre à un emplacement de mémoire différent n'aboutit qu'à l'intérieur de la méthode et n'affecte pas la variable d'origine, `arr`.

C#

```
int[] arr = { 1, 4, 5 };
System.Console.WriteLine("Inside Main, before calling the method, the first
element is: {0}", arr[0]);

Change(arr);
System.Console.WriteLine("Inside Main, after calling the method, the first
element is: {0}", arr[0]);

static void Change(int[] pArray)
{
    pArray[0] = 888;  // This change affects the original element.
    pArray = new int[5] { -3, -1, -2, -3, -4 };   // This change is local.
    System.Console.WriteLine("Inside the method, the first element is: {0}",
pArray[0]);
}
/* Output:
    Inside Main, before calling the method, the first element is: 1
    Inside the method, the first element is: -3
    Inside Main, after calling the method, the first element is: 888
*/
```

Dans l'exemple précédent, le tableau `arr`, qui est un type référence, est passé à la méthode sans le paramètre `ref`. Dans ce cas, une copie de la référence, qui pointe vers `arr`, est passée à la méthode. La sortie montre que la méthode peut modifier le contenu d'un élément du tableau, dans ce cas de `1` à `888`. Cependant, si vous allouez une nouvelle portion de mémoire via l'opérateur new à l'intérieur de la méthode `Change`, la variable `pArray` fait référence à un nouveau tableau. Par conséquent, les modifications apportées à la suite de cette opération n'ont pas d'effet sur le tableau d'origine (`arr`), qui est créé à l'intérieur de `Main`. En fait, deux tableaux sont créés dans cet exemple : un à l'intérieur de `Main` et un autre à l'intérieur de la méthode `Change`.

Passer un type référence par référence

Lorsque vous passez un type *référencepar référence* :

- Si la méthode affecte le paramètre pour faire référence à un autre objet, ces modifications **sont visibles** à partir de l'appelant.
- Si la méthode modifie l'état de l'objet auquel le paramètre fait référence, ces modifications **sont visibles** à partir de l'appelant.

L'exemple suivant est identique au précédent, sauf que le mot clé `ref` est ajouté à l'en-tête et à l'appel de la méthode. Les modifications qui se produisent dans la méthode affectent la variable d'origine dans le programme appelant.

C#

```
int[] arr = { 1, 4, 5 };
System.Console.WriteLine("Inside Main, before calling the method, the first
element is: {0}", arr[0]);

Change(ref arr);
System.Console.WriteLine("Inside Main, after calling the method, the first
element is: {0}", arr[0]);

static void Change(ref int[] pArray)
{
    // Both of the following changes will affect the original variables:
    pArray[0] = 888;
    pArray = new int[5] { -3, -1, -2, -3, -4 };
    System.Console.WriteLine("Inside the method, the first element is: {0}",
pArray[0]);
}
/* Output:
    Inside Main, before calling the method, the first element is: 1
    Inside the method, the first element is: -3
    Inside Main, after calling the method, the first element is: -3
*/
```

Les modifications qui se produisent à l'intérieur de la méthode affectent le tableau d'origine dans `Main`. En fait, le tableau d'origine est réalloué à l'aide de l'opérateur `new`. Par conséquent, après l'appel à la méthode `Change`, toute référence à `arr` pointe vers le tableau à cinq éléments, qui est créé dans la méthode `Change`.

Étendue des références et des valeurs

Les méthodes peuvent stocker les valeurs des paramètres dans des champs. Lorsque les paramètres sont passés par valeur, cela est toujours fait de façon sécurisée. Les valeurs sont copiées et les types de référence sont accessibles lorsqu'ils sont stockés dans un champ. La transmission de paramètres par référence de façon sûre nécessite que le compilateur définisse quand il est sûr d'affecter une référence à une nouvelle variable. Pour chaque expression, le compilateur définit une *étendue* qui limite l'accès à une

expression ou à une variable. Le compilateur utilise deux étendues : *safe_to_escape* et *ref_safe_to_escape*.

- L'étendue *safe_to_escape* définit l'étendue dans laquelle n'importe quelle expression est accessible en toute sécurité.
- L'étendue *ref_safe_to_escape* définit l'étendue dans laquelle une *référence* à n'importe quelle expression peut être accessible ou modifiée en toute sécurité.

De manière informelle, vous pouvez considérer ces étendues comme le mécanisme permettant de vous assurer que votre code n'accède jamais ou ne modifie jamais une référence qui n'est plus valide. Une référence est valide tant qu'elle fait référence à un objet ou à un struct valide. L'étendue *safe_to_escape* définit quand une variable peut être affectée ou réaffectée. L'étendue *ref_safe_to_escape* définit quand une variable peut être affectée par *référence* ou réaffectée par *référence*. L'affectation affecte une variable à une nouvelle valeur ; *l'affectation par référence* affecte la variable pour *faire référence à* un autre emplacement de stockage.

Modificateurs

Les paramètres déclarés pour une méthode sans in, ref ou out sont passés à la méthode appelée par valeur. Les modificateurs `ref`, `in` et `out` diffèrent dans les règles d'affectation :

- L'argument d'un paramètre `ref` doit être définitivement affecté. La méthode appelée peut réaffecter ce paramètre.
- L'argument d'un paramètre `in` doit être définitivement affecté. La méthode appelée ne peut pas réaffecter ce paramètre.
- L'argument d'un paramètre `out` n'a pas besoin d'être définitivement affecté. La méthode appelée doit affecter le paramètre.

Cette section décrit les mots clés que vous pouvez utiliser pour déclarer des paramètres de méthode :

- params spécifie que ce paramètre peut prendre un nombre variable d'arguments.
- in spécifie que ce paramètre est passé par référence, mais qu'il est lu uniquement par la méthode appelée.
- ref spécifie que ce paramètre est passé par référence et qu'il peut être lu ou écrit par la méthode appelée.
- out spécifie que ce paramètre est passé par référence et qu'il est écrit par la méthode appelée.

Voir aussi

- Référence C#
- Mots clés C#
- Liste des arguments dans la Spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Paramètres de méthode (référence C#)

Article • 07/04/2023

En C#, les arguments peuvent être passés aux paramètres par valeur ou par référence. N'oubliez pas que les types C# peuvent être des types référence (`class`) ou des types valeur (`struct`) :

- *Passer par valeur* signifie **passer une copie de la variable** à la méthode.
- *Passer par référence* signifie **passer l'accès à la variable** à la méthode.
- Une variable d'un *type référence* contient une référence à ses données.
- Une variable d'un *type valeur* contient ses données directement.

Étant donné qu'un struct est un type valeur, quand vous passez un struct par valeur à une méthode, la méthode reçoit et traite une copie de l'argument struct. La méthode n'a pas accès au struct d'origine dans la méthode d'appel, et ne peut donc pas le modifier de quelque manière que ce soit. La méthode peut modifier uniquement la copie.

Une instance de classe est un type référence, pas un type valeur. Quand un type référence est passé par valeur à une méthode, celle-ci reçoit une copie de la référence à l'instance de classe. Autrement dit, la méthode appelée reçoit une copie de l'adresse de l'instance, et la méthode appelante conserve l'adresse d'origine de l'instance. L'instance de classe dans la méthode d'appel a une adresse, le paramètre dans la méthode appelée a une copie de l'adresse, et les deux adresses font référence au même objet. Étant donné que le paramètre contient uniquement une copie de l'adresse, la méthode appelée ne peut pas modifier l'adresse de l'instance de la classe dans la méthode. Toutefois, la méthode appelée peut utiliser la copie de l'adresse pour accéder aux membres de la classe qui ont à la fois l'adresse d'origine et la copie de la référence d'adresse. Si la méthode appelée modifie un membre de classe, l'instance de classe d'origine dans la méthode d'appel change également.

La sortie de l'exemple suivant illustre la différence. La valeur du champ `willIChange` de l'instance de classe est changée par l'appel à la méthode `ClassTaker`, car la méthode utilise l'adresse mentionnée dans le paramètre pour rechercher le champ spécifié de l'instance de classe. Le champ `willIChange` du struct de la méthode d'appel n'est pas changé par l'appel à la méthode `StructTaker`, car la valeur de l'argument est une copie du struct lui-même, et non pas une copie de son adresse. `StructTaker` change la copie, et cette dernière est perdue quand l'appel à `StructTaker` est terminé.

```
C#
```

```
class TheClass
{
    public string? willIChange;
}

struct TheStruct
{
    public string willIChange;
}

class TestClassAndStruct
{
    static void ClassTaker(TheClass c)
    {
        c.willIChange = "Changed";
    }

    static void StructTaker(TheStruct s)
    {
        s.willIChange = "Changed";
    }

    public static void Main()
    {
        TheClass testClass = new TheClass();
        TheStruct testStruct = new TheStruct();

        testClass.willIChange = "Not Changed";
        testStruct.willIChange = "Not Changed";

        ClassTaker(testClass);
        StructTaker(testStruct);

        Console.WriteLine("Class field = {0}", testClass.willIChange);
        Console.WriteLine("Struct field = {0}", testStruct.willIChange);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Class field = Changed
    Struct field = Not Changed
*/
```

Les modifications apportées à l'argument qui sont visibles à partir de l'appelant dépendent de la façon dont un argument est passé et de s'il s'agit d'un type référence ou d'un type valeur.

Passer un type valeur par valeur

Lorsque vous passez un type *valeurpar valeur* :

- Si la méthode affecte le paramètre pour faire référence à un autre objet, ces modifications **ne sont pas** visibles à partir de l'appelant.
- Si la méthode modifie l'état de l'objet auquel le paramètre fait référence, ces modifications **ne sont pas** visibles par l'appelant.

L'exemple suivant illustre le passage de paramètres de type valeur par valeur. La variable `n` est passée par valeur à la méthode `SquareIt`. Toutes les modifications qui ont lieu à l'intérieur de la méthode n'ont aucun effet sur la valeur d'origine de la variable.

C#

```
int n = 5;
System.Console.WriteLine("The value before calling the method: {0}", n);

SquareIt(n);  // Passing the variable by value.
System.Console.WriteLine("The value after calling the method: {0}", n);

// Keep the console window open in debug mode.
System.Console.WriteLine("Press any key to exit.");
System.Console.ReadKey();

static void SquareIt(int x)
// The parameter x is passed by value.
// Changes to x will not affect the original value of x.
{
    x *= x;
    System.Console.WriteLine("The value inside the method: {0}", x);
}
/* Output:
    The value before calling the method: 5
    The value inside the method: 25
    The value after calling the method: 5
*/
```

La variable `n` est un type valeur. Elle contient ses données, la valeur `5`. Quand `SquareIt` est appelée, le contenu de `n` est copié dans le paramètre `x`, qui est mis au carré à l'intérieur de la méthode. Dans `Main`, toutefois, la valeur de `n` est identique avant et après l'appel de la méthode `SquareIt`. La modification qui intervient à l'intérieur de la méthode affecte uniquement la variable locale `x`.

Passer un type valeur par référence

Lorsque vous transmettez un type *valeurpar référence* :

- Si la méthode affecte le paramètre pour faire référence à un autre objet, ces modifications **ne sont pas** visibles à partir de l'appelant.
- Si la méthode modifie l'état de l'objet auquel le paramètre fait référence, ces modifications **sont visibles** à partir de l'appelant.

L'exemple suivant est identique au précédent, sauf que l'argument est passé en tant que paramètre `ref`. La valeur de l'argument sous-jacent, `n`, est modifiée quand `x` est modifiée dans la méthode.

C#

```
int n = 5;
System.Console.WriteLine("The value before calling the method: {0}", n);

SquareIt(ref n);  // Passing the variable by reference.
System.Console.WriteLine("The value after calling the method: {0}", n);

// Keep the console window open in debug mode.
System.Console.WriteLine("Press any key to exit.");
System.Console.ReadKey();

static void SquareIt(ref int x)
// The parameter x is passed by reference.
// Changes to x will affect the original value of x.
{
    x *= x;
    System.Console.WriteLine("The value inside the method: {0}", x);
}
/* Output:
    The value before calling the method: 5
    The value inside the method: 25
    The value after calling the method: 25
*/
```

Dans cet exemple, ce n'est pas la valeur de `n` qui est passée, mais plutôt une référence à `n`. Le paramètre `x` n'est pas un int ; il s'agit d'une référence à un `int`, dans le cas présent une référence à `n`. Ainsi, quand `x` est mis au carré à l'intérieur de la méthode, ce qui est réellement mis au carré est ce à quoi `x` fait référence, c'est-à-dire `n`.

Passer un type référence par valeur

Lorsque vous passez un type *référencepar valeur* :

- Si la méthode affecte le paramètre pour faire référence à un autre objet, ces modifications **ne sont pas** visibles à partir de l'appelant.

- Si la méthode modifie l'état de l'objet auquel le paramètre fait référence, ces modifications **sont visibles** à partir de l'appelant.

L'exemple suivant illustre le passage d'un paramètre de type référence (`arr`) par valeur, à une méthode (`Change`). Sachant que le paramètre est une référence à `arr`, il est possible de modifier les valeurs des éléments du tableau. Cependant, la tentative de réassignation du paramètre à un emplacement de mémoire différent n'aboutit qu'à l'intérieur de la méthode et n'affecte pas la variable d'origine, `arr`.

C#

```
int[] arr = { 1, 4, 5 };
System.Console.WriteLine("Inside Main, before calling the method, the first
element is: {0}", arr[0]);

Change(arr);
System.Console.WriteLine("Inside Main, after calling the method, the first
element is: {0}", arr[0]);

static void Change(int[] pArray)
{
    pArray[0] = 888;  // This change affects the original element.
    pArray = new int[5] { -3, -1, -2, -3, -4 };   // This change is local.
    System.Console.WriteLine("Inside the method, the first element is: {0}",
pArray[0]);
}
/* Output:
    Inside Main, before calling the method, the first element is: 1
    Inside the method, the first element is: -3
    Inside Main, after calling the method, the first element is: 888
*/
```

Dans l'exemple précédent, le tableau `arr`, qui est un type référence, est passé à la méthode sans le paramètre `ref`. Dans ce cas, une copie de la référence, qui pointe vers `arr`, est passée à la méthode. La sortie montre que la méthode peut modifier le contenu d'un élément du tableau, dans ce cas de `1` à `888`. Cependant, si vous allouez une nouvelle portion de mémoire via l'opérateur new à l'intérieur de la méthode `Change`, la variable `pArray` fait référence à un nouveau tableau. Par conséquent, les modifications apportées à la suite de cette opération n'ont pas d'effet sur le tableau d'origine (`arr`), qui est créé à l'intérieur de `Main`. En fait, deux tableaux sont créés dans cet exemple : un à l'intérieur de `Main` et un autre à l'intérieur de la méthode `Change`.

Passer un type référence par référence

Lorsque vous passez un type *référencepar référence* :

- Si la méthode affecte le paramètre pour faire référence à un autre objet, ces modifications **sont visibles** à partir de l'appelant.
- Si la méthode modifie l'état de l'objet auquel le paramètre fait référence, ces modifications **sont visibles** à partir de l'appelant.

L'exemple suivant est identique au précédent, sauf que le mot clé `ref` est ajouté à l'en-tête et à l'appel de la méthode. Les modifications qui se produisent dans la méthode affectent la variable d'origine dans le programme appelant.

C#

```
int[] arr = { 1, 4, 5 };
System.Console.WriteLine("Inside Main, before calling the method, the first
element is: {0}", arr[0]);

Change(ref arr);
System.Console.WriteLine("Inside Main, after calling the method, the first
element is: {0}", arr[0]);

static void Change(ref int[] pArray)
{
    // Both of the following changes will affect the original variables:
    pArray[0] = 888;
    pArray = new int[5] { -3, -1, -2, -3, -4 };
    System.Console.WriteLine("Inside the method, the first element is: {0}",
pArray[0]);
}
/* Output:
    Inside Main, before calling the method, the first element is: 1
    Inside the method, the first element is: -3
    Inside Main, after calling the method, the first element is: -3
*/
```

Les modifications qui se produisent à l'intérieur de la méthode affectent le tableau d'origine dans `Main`. En fait, le tableau d'origine est réalloué à l'aide de l'opérateur `new`. Par conséquent, après l'appel à la méthode `Change`, toute référence à `arr` pointe vers le tableau à cinq éléments, qui est créé dans la méthode `Change`.

Étendue des références et des valeurs

Les méthodes peuvent stocker les valeurs des paramètres dans des champs. Lorsque les paramètres sont passés par valeur, cela est toujours fait de façon sécurisée. Les valeurs sont copiées et les types de référence sont accessibles lorsqu'ils sont stockés dans un champ. La transmission de paramètres par référence de façon sûre nécessite que le compilateur définisse quand il est sûr d'affecter une référence à une nouvelle variable. Pour chaque expression, le compilateur définit une *étendue* qui limite l'accès à une

expression ou à une variable. Le compilateur utilise deux étendues : *safe_to_escape* et *ref_safe_to_escape*.

- L'étendue *safe_to_escape* définit l'étendue dans laquelle n'importe quelle expression est accessible en toute sécurité.
- L'étendue *ref_safe_to_escape* définit l'étendue dans laquelle une *référence* à n'importe quelle expression peut être accessible ou modifiée en toute sécurité.

De manière informelle, vous pouvez considérer ces étendues comme le mécanisme permettant de vous assurer que votre code n'accède jamais ou ne modifie jamais une référence qui n'est plus valide. Une référence est valide tant qu'elle fait référence à un objet ou à un struct valide. L'étendue *safe_to_escape* définit quand une variable peut être affectée ou réaffectée. L'étendue *ref_safe_to_escape* définit quand une variable peut être affectée par *référence* ou réaffectée par *référence*. L'affectation affecte une variable à une nouvelle valeur ; *l'affectation par référence* affecte la variable pour *faire référence à* un autre emplacement de stockage.

Modificateurs

Les paramètres déclarés pour une méthode sans in, ref ou out sont passés à la méthode appelée par valeur. Les modificateurs `ref`, `in` et `out` diffèrent dans les règles d'affectation :

- L'argument d'un paramètre `ref` doit être définitivement affecté. La méthode appelée peut réaffecter ce paramètre.
- L'argument d'un paramètre `in` doit être définitivement affecté. La méthode appelée ne peut pas réaffecter ce paramètre.
- L'argument d'un paramètre `out` n'a pas besoin d'être définitivement affecté. La méthode appelée doit affecter le paramètre.

Cette section décrit les mots clés que vous pouvez utiliser pour déclarer des paramètres de méthode :

- params spécifie que ce paramètre peut prendre un nombre variable d'arguments.
- in spécifie que ce paramètre est passé par référence, mais qu'il est lu uniquement par la méthode appelée.
- ref spécifie que ce paramètre est passé par référence et qu'il peut être lu ou écrit par la méthode appelée.
- out spécifie que ce paramètre est passé par référence et qu'il est écrit par la méthode appelée.

Voir aussi

- Référence C#
- Mots clés C#
- Liste des arguments dans la Spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Variables locales implicitement typées (Guide de programmation C#)

Article • 21/03/2023

Les variables locales peuvent être déclarées sans donner de type explicite. Le mot clé `var` indique au compilateur de déduire le type de la variable à partir de l'expression située à droite de l'instruction d'initialisation. Le type déduit peut être un type intégré, un type anonyme, un type défini par l'utilisateur ou un type défini dans la bibliothèque de classes .NET. Pour plus d'informations sur l'initialisation des tableaux avec `var`, consultez Tableaux implicitement typés.

Les exemples suivants montrent différentes manières de déclarer des variables locales avec `var` :

C#

```
// i is compiled as an int
var i = 5;

// s is compiled as a string
var s = "Hello";

// a is compiled as int[]
var a = new[] { 0, 1, 2 };

// expr is compiled as IEnumerable<Customer>
// or perhaps IQueryable<Customer>
var expr =
    from c in customers
    where c.City == "London"
    select c;

// anon is compiled as an anonymous type
var anon = new { Name = "Terry", Age = 34 };

// list is compiled as List<int>
var list = new List<int>();
```

Il est important de comprendre que le mot clé `var` n'est pas « variant » et qu'il n'indique pas que la variable est peu typée ou à liaison tardive. Cela signifie simplement que le compilateur détermine et assigne le type qui convient le mieux.

Le mot clé `var` peut être utilisé dans les contextes suivants :

- Dans des variables locales (variables déclarées dans la portée de la méthode), comme indiqué dans l'exemple précédent.
- Dans une instruction d'initialisation for

```
C#

for (var x = 1; x < 10; x++)
```

- Dans une instruction d'initialisation foreach

```
C#

foreach (var item in list) {...}
```

- Dans une instruction using

```
C#

using (var file = new StreamReader("C:\\myfile.txt")) {...}
```

Pour plus d'informations, consultez Guide pratique pour utiliser des tableaux et des variables locales implicitement typés dans une expression de requête.

Types var et anonymes

Dans de nombreux cas, l'utilisation de `var` est facultative et sert uniquement à simplifier la syntaxe. Toutefois, lorsqu'une variable est initialisée avec un type anonyme, vous devez déclarer la variable en tant que `var` si vous savez déjà que vous aurez besoin d'accéder aux propriétés de l'objet. Il s'agit d'un scénario courant avec les expressions de requête LINQ. Pour plus d'informations, consultez Types anonymes.

Du point de vue de votre code source, un type anonyme n'a pas de nom. Par conséquent, si une variable de requête a été initialisée avec `var`, la seule façon d'accéder aux propriétés de la séquence d'objets retournée consiste à utiliser `var` comme type pour la variable d'itération de l'instruction `foreach`.

```
C#

class ImplicitlyTypedLocals2
{
    static void Main()
    {
```

```
        string[] words = { "aPPLE", "BlUeBeRrY", "cHeRry" };

        // If a query produces a sequence of anonymous types,
        // then use var in the foreach statement to access the properties.
        var upperLowerWords =
             from w in words
             select new { Upper = w.ToUpper(), Lower = w.ToLower() };

        // Execute the query
        foreach (var ul in upperLowerWords)
        {
            Console.WriteLine("Uppercase: {0}, Lowercase: {1}", ul.Upper,
ul.Lower);
        }
    }
}
/* Outputs:
    Uppercase: APPLE, Lowercase: apple
    Uppercase: BLUEBERRY, Lowercase: blueberry
    Uppercase: CHERRY, Lowercase: cherry
 */
```

Remarques

Les restrictions suivantes s'appliquent aux déclarations de variables implicitement typées :

- `var` peut être utilisé uniquement lorsqu'une variable locale est déclarée et initialisée dans la même instruction. La variable ne peut pas être initialisée vers la valeur Null, vers un groupe de méthodes ou vers une fonction anonyme.
- `var` ne peut pas être utilisé dans les champs situés dans la portée de la classe.
- Les variables déclarées à l'aide de `var` ne peuvent pas être utilisées dans l'expression d'initialisation. En d'autres termes, l'expression `int i = (i = 20);` est légale, mais l'expression génère une erreur de compilation : `var i = (i = 20);`
- Il n'est pas possible d'initialiser plusieurs variables implicitement typées dans la même instruction.
- Si un type nommé `var` se trouve dans la portée, le mot clé `var` est résolu en ce nom de type et n'est pas considéré comme faisant partie d'une déclaration de variable locale implicitement typée.

Le typage implicite avec le mot clé `var` ne peut être appliqué qu'aux variables comprises dans la portée de la méthode locale. Le typage implicite n'est pas disponible pour les champs de classe, car le compilateur C# rencontrerait un paradoxe logique

pendant le traitement du code : le compilateur a besoin de connaître le type du champ, mais il ne peut pas déterminer le type tant que l'expression d'assignation n'est pas analysée, et l'expression ne peut pas être évaluée sans connaître le type. Prenez le code suivant :

```
C#

private var bookTitles;
```

`bookTitles` est un champ de classe de type `var`. Le champ n'ayant aucune expression à évaluer, le compilateur ne peut pas déduire le type que `bookTitles` est censé avoir. De plus, l'ajout d'une expression au champ (comme vous le feriez pour une variable locale) est également insuffisant :

```
C#

private var bookTitles = new List<string>();
```

Quand le compilateur rencontre des champs pendant la compilation du code, il enregistre le type de chaque champ avant de traiter toutes les expressions associées. Le compilateur rencontre le même paradoxe en essayant d'analyser `bookTitles` : il doit connaître le type du champ, mais il détermine normalement le type de `var` en analysant l'expression, ce qui est impossible sans connaître le type au préalable.

`var` peut également être utile avec les expressions de requête dans lesquelles il est difficile de déterminer le type construit exact de la variable de requête. Cela peut se produire avec les opérations de regroupement et de classement.

Le mot clé `var` peut également être utile lorsque le type de la variable est fastidieux à taper, ou bien lorsqu'il est évident ou lorsqu'il n'aide pas à la lisibilité du code. Par exemple, `var` est utile avec les types génériques imbriqués tels que ceux utilisés avec les opérations de groupe. Dans la requête suivante, le type de la variable de requête est `IEnumerable<IGrouping<string, Student>>`. Tant que les personnes qui doivent gérer votre code comprennent ceci, le typage implicite peut tout à fait être utilisé pour rendre votre code plus concis et simplifier sa gestion.

```
C#

// Same as previous example except we use the entire last name as a key.
// Query variable is an IEnumerable<IGrouping<string, Student>>
var studentQuery3 =
    from student in students
    group student by student.Last;
```

L'utilisation de `var` permet de simplifier votre code, mais son utilisation doit être limitée aux cas où il est nécessaire ou lorsqu'il rend votre code plus facile à lire. Pour plus d'informations sur le moment où utiliser `var` correctement, consultez la section Variables locales typées implicitement dans l'article Instructions de codage C#.

Voir aussi

- Référence C#
- Tableaux implicitement typés
- Comment : utiliser des tableaux et des variables locales implicitement typés dans une expression de requête
- Types anonymes
- Initialiseurs d'objets et de collections
- var
- LINQ en C#
- LINQ (Language Integrated Query)
- Instructions d'itération
- using, instruction

Comment utiliser des tableaux et des variables locales implicitement typés dans une expression de requête (Guide de programmation C#)

Article • 07/04/2023

Vous pouvez utiliser des variables locales implicitement typées chaque fois que vous voulez que le compilateur détermine le type d'une variable locale. Vous devez utiliser des variables locales implicitement typées pour stocker des types anonymes, qui sont souvent utilisés dans les expressions de requête. Les exemples suivants illustrent des utilisations facultatives et obligatoires de variables locales implicitement typées dans des requêtes.

Les variables locales implicitement typées sont déclarées à l'aide du mot clé contextuel var. Pour plus d'informations, consultez Variables locales implicitement typées et Tableaux implicitement typés.

Exemples

L'exemple suivant illustre un scénario courant dans lequel le mot clé `var` est obligatoire : une expression de requête qui produit une séquence de types anonymes. Dans ce scénario, la variable de requête et la variable d'itération figurant dans l'instruction `foreach` doivent être implicitement typées avec `var`, car vous n'avez pas accès à un nom de type pour le type anonyme. Pour plus d'informations sur les types anonymes, consultez Types anonymes.

C#

```
private static void QueryNames(char firstLetter)
{
    // Create the query. Use of var is required because
    // the query produces a sequence of anonymous types:
    // System.Collections.Generic.IEnumerable<????>.
    var studentQuery =
        from student in students
        where student.FirstName[0] == firstLetter
        select new { student.FirstName, student.LastName };

    // Execute the query and display the results.
    foreach (var anonType in studentQuery)
    {
```

```
        Console.WriteLine("First = {0}, Last = {1}", anonType.FirstName,
anonType.LastName);
    }
}
```

Dans l'exemple suivant, le mot clé `var` est utilisé dans un cas similaire, à la différence que l'utilisation de `var` est facultative. Comme `student.LastName` est une chaîne, l'exécution de la requête retourne une séquence de chaînes. De ce fait, le type de `queryId` pourrait être déclaré en tant que `System.Collections.Generic.IEnumerable<string>` au lieu de `var`. Le mot clé `var` est utilisé pour des raisons pratiques. Dans l'exemple, la variable d'itération figurant dans l'instruction `foreach` est explicitement typée en tant que chaîne, mais elle aurait pu être déclarée à l'aide de `var`. Comme le type de la variable d'itération n'est pas un type anonyme, l'utilisation de `var` est facultative et non obligatoire. Pour rappel, `var` n'est pas en soi un type, mais une instruction indiquant au compilateur de déduire et d'assigner le type.

C#

```
// Variable queryId could be declared by using
// System.Collections.Generic.IEnumerable<string>
// instead of var.
var queryId =
    from student in students
    where student.Id > 111
    select student.LastName;

// Variable str could be declared by using var instead of string.
foreach (string str in queryId)
{
    Console.WriteLine("Last name: {0}", str);
}
```

Voir aussi

- Guide de programmation C#
- Méthodes d'extension
- LINQ (Language Integrated Query)
- LINQ en C#

Méthodes d’extension (Guide de programmation C#)

Article • 10/05/2023

Les méthodes d'extension vous permettent d'« ajouter » des méthodes à des types existants sans créer un type dérivé, ni recompiler ou modifier le type d'origine. Les méthodes d'extension sont des méthodes statiques appelées comme s'il s'agissait de méthodes d'instance sur le type étendu. Pour le code client écrit en C#, F# et Visual Basic, il n’y a aucune différence apparente lors de l’appel entre une méthode d’extension et les méthodes définies dans un type.

Les méthodes d’extension les plus courantes sont les opérateurs de requête standard LINQ qui ajoutent des fonctionnalités de requête aux types System.Collections.IEnumerable et System.Collections.Generic.IEnumerable<T> existants. Pour utiliser les opérateurs de requête standard, introduisez-les d'abord dans la portée avec une directive `using System.Linq`. Puis, tout type qui implémente IEnumerable<T> semble avoir des méthodes d'instance telles que GroupBy, OrderBy, Average, etc. Vous pouvez consulter ces méthodes supplémentaires dans la saisie semi-automatique des instructions IntelliSense quand vous tapez un « point » après une instance d’un type IEnumerable<T> tel que List<T> ou Array.

Exemple OrderBy

L'exemple suivant indique comment appeler la méthode `OrderBy` d'opérateur de requête standard sur un tableau d'entiers. L'expression entre parenthèses est une expression lambda. De nombreux opérateurs de requête standard prennent des expressions lambda comme paramètres, mais ce n’est pas requis pour les méthodes d’extension. Pour plus d’informations, consultez Expressions lambda.

C#

```
class ExtensionMethods2
{

    static void Main()
    {
        int[] ints = { 10, 45, 15, 39, 21, 26 };
        var result = ints.OrderBy(g => g);
        foreach (var i in result)
        {
            System.Console.Write(i + " ");
        }
```

```
    }
}
//Output: 10 15 21 26 39 45
```

Les méthodes d'extension sont définies comme méthodes statiques mais sont appelées en utilisant la syntaxe de méthode d'instance. Leur premier paramètre spécifie le type sur lequel la méthode opère. Le paramètre est précédé de ce modificateur. Les méthodes d'extension sont uniquement dans la portée lorsque vous importez explicitement l'espace de noms dans votre code source avec une directive `using`.

L'exemple suivant présente une méthode d'extension définie pour la classe System.String. Elle est définie à l'intérieur d'une classe statique, non imbriquée et non générique :

C#

```
namespace ExtensionMethods
{
    public static class MyExtensions
    {
        public static int WordCount(this string str)
        {
            return str.Split(new char[] { ' ', '.', '?' },
                             StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
```

La méthode d'extension `WordCount` peut être mise à portée avec cette directive `using` :

C#

```
using ExtensionMethods;
```

Elle peut être appelée à partir d'une application à l'aide de cette syntaxe :

C#

```
string s = "Hello Extension Methods";
int i = s.WordCount();
```

Vous appelez la méthode d'extension dans votre code avec la syntaxe de méthode d'instance. Le langage intermédiaire (IL) généré par le compilateur traduit votre code dans un appel sur la méthode statique. Le principe d'encapsulation n'est pas réellement

violé. Les méthodes d'extension ne peuvent pas accéder aux variables privées dans le type qu'elles étendent.

La classe `MyExtensions` et la méthode `WordCount` sont `static` et elles sont accessibles comme tous les autres membres `static`. La méthode `WordCount` peut être appelée comme d'autres méthodes `static` comme suit :

```
C#

string s = "Hello Extension Methods";
int i = MyExtensions.WordCount(s);
```

Le code C# précédent :

- Déclare et affecte un nouveau `string` nommé `s` avec la valeur `"Hello Extension Methods"`.
- Appelle l'argument `MyExtensions.WordCount` donné `s`

Pour plus d'informations, consultez Comment implémenter et appeler une méthode d'extension personnalisée.

En général, vous appellerez probablement les méthodes d'extension beaucoup plus souvent que vous n'implémenterez les vôtres. Comme les méthodes d'extension sont appelées à l'aide de la syntaxe de méthode d'instance, aucune connaissance particulière n'est requise pour les utiliser depuis le code client. Pour activer des méthodes d'extension pour un type particulier, ajoutez simplement une directive `using` pour l'espace de noms dans lequel les méthodes sont définies. Par exemple, pour utiliser les opérateurs de requête standard, ajoutez la directive `using` à votre code :

```
C#

using System.Linq;
```

(Vous devrez peut-être également ajouter une référence à System.Core.dll.) Vous remarquerez que les opérateurs de requête standard apparaissent désormais dans IntelliSense en tant que méthodes supplémentaires disponibles pour la plupart des types IEnumerable<T>.

Liaison de méthodes d'extension à la compilation

Vous pouvez utiliser des méthodes d'extension pour étendre une classe ou une interface, mais pas pour les remplacer. Une méthode d'extension avec le même nom et la même signature qu'une méthode d'interface ou de classe ne sera jamais appelée. Au moment de la compilation, les méthodes d'extension ont toujours la priorité la plus faible par rapport aux méthodes d'instance définies dans le type lui-même. En d'autres termes, si un type a une méthode nommée `Process(int i)` et que vous avez une méthode d'extension avec la même signature, le compilateur créera toujours une liaison avec la méthode d'instance. Lorsque le compilateur rencontre un appel de méthode, il recherche d'abord une correspondance dans les méthodes d'instance du type. Si aucune correspondance n'est trouvée, il recherche toutes les méthodes d'extension définies pour le type et crée une liaison avec la première méthode d'extension qu'il trouve. L'exemple suivant montre comment le compilateur détermine quelle méthode d'extension ou méthode d'instance est choisie pour créer une liaison.

Exemple

L'exemple suivant montre les règles que le compilateur C# suit pour déterminer s'il faut lier un appel de méthode à une méthode d'instance sur le type, ou à une méthode d'extension. Le classe statique `Extensions` contient des méthodes d'extension définies pour tout type qui implémente `IMyInterface`. Les classes `A`, `B` et `C` implémentent toutes l'interface.

La méthode d'extension `MethodB` n'est jamais appelée car son nom et sa signature correspondent exactement aux méthodes déjà implémentées par les classes.

Lorsque le compilateur ne trouve pas de méthode d'instance avec une signature correspondante, il crée une liaison avec une méthode d'extension correspondante, s'il en existe une.

C#

```
// Define an interface named IMyInterface.
namespace DefineIMyInterface
{
    using System;

    public interface IMyInterface
    {
        // Any class that implements IMyInterface must define a method
        // that matches the following signature.
        void MethodB();
    }
}

// Define extension methods for IMyInterface.
```

```
namespace Extensions
{
    using System;
    using DefineIMyInterface;

    // The following extension methods can be accessed by instances of any
    // class that implements IMyInterface.
    public static class Extension
    {
        public static void MethodA(this IMyInterface myInterface, int i)
        {
            Console.WriteLine
                ("Extension.MethodA(this IMyInterface myInterface, int i)");
        }

        public static void MethodA(this IMyInterface myInterface, string s)
        {
            Console.WriteLine
                ("Extension.MethodA(this IMyInterface myInterface, string
s)");
        }

        // This method is never called in ExtensionMethodsDemo1, because
each
        // of the three classes A, B, and C implements a method named
MethodB
        // that has a matching signature.
        public static void MethodB(this IMyInterface myInterface)
        {
            Console.WriteLine
                ("Extension.MethodB(this IMyInterface myInterface)");
        }
    }
}

// Define three classes that implement IMyInterface, and then use them to
test
// the extension methods.
namespace ExtensionMethodsDemo1
{
    using System;
    using Extensions;
    using DefineIMyInterface;

    class A : IMyInterface
    {
        public void MethodB() { Console.WriteLine("A.MethodB()"); }
    }

    class B : IMyInterface
    {
        public void MethodB() { Console.WriteLine("B.MethodB()"); }
        public void MethodA(int i) { Console.WriteLine("B.MethodA(int i)");
}
    }
```

```
    class C : IMyInterface
    {
        public void MethodB() { Console.WriteLine("C.MethodB()"); }
        public void MethodA(object obj)
        {
            Console.WriteLine("C.MethodA(object obj)");
        }
    }

    class ExtMethodDemo
    {
        static void Main(string[] args)
        {
            // Declare an instance of class A, class B, and class C.
            A a = new A();
            B b = new B();
            C c = new C();

            // For a, b, and c, call the following methods:
            //      -- MethodA with an int argument
            //      -- MethodA with a string argument
            //      -- MethodB with no argument.

            // A contains no MethodA, so each call to MethodA resolves to
            // the extension method that has a matching signature.
            a.MethodA(1);           // Extension.MethodA(IMyInterface, int)
            a.MethodA("hello");     // Extension.MethodA(IMyInterface,
string)

            // A has a method that matches the signature of the following
call
            // to MethodB.
            a.MethodB();            // A.MethodB()

            // B has methods that match the signatures of the following
            // method calls.
            b.MethodA(1);           // B.MethodA(int)
            b.MethodB();            // B.MethodB()

            // B has no matching method for the following call, but
            // class Extension does.
            b.MethodA("hello");     // Extension.MethodA(IMyInterface,
string)

            // C contains an instance method that matches each of the
following
            // method calls.
            c.MethodA(1);           // C.MethodA(object)
            c.MethodA("hello");     // C.MethodA(object)
            c.MethodB();            // C.MethodB()
        }
    }
}
/* Output:
```

```
    Extension.MethodA(this IMyInterface myInterface, int i)
    Extension.MethodA(this IMyInterface myInterface, string s)
    A.MethodB()
    B.MethodA(int i)
    B.MethodB()
    Extension.MethodA(this IMyInterface myInterface, string s)
    C.MethodA(object obj)
    C.MethodA(object obj)
    C.MethodB()
 */
```

Modes d'utilisation courants

Fonctionnalités des collections

Dans le passé, il était courant de créer des « classes de collection » qui implémentaient l'interface System.Collections.Generic.IEnumerable<T> pour un type donné et contenaient des fonctionnalités qui agissaient sur des collections de ce type. Bien qu'il n'y ait rien de mal à créer ce type d'objet de collection, la même fonctionnalité peut être obtenue à l'aide d'une extension sur System.Collections.Generic.IEnumerable<T>. Les extensions ont l'avantage de permettre à la fonctionnalité d'être appelée à partir de n'importe quelle collection, telle que System.Array ou System.Collections.Generic.List<T> qui implémente System.Collections.Generic.IEnumerable<T> sur ce type. Vous trouverez un exemple de ceci à l'aide d'un tableau d'Int32 plus haut dans cet article.

Fonctionnalité Layer-Specific

Lors de l'utilisation d'une architecture Onion ou d'une autre conception d'application en couches, il est commun d'avoir un ensemble d'entités de domaine ou d'objets de transfert de données qui peuvent être utilisés pour communiquer entre les limites de l'application. Ces objets ne contiennent généralement aucune fonctionnalité, ou seulement des fonctionnalités minimales qui s'appliquent à toutes les couches de l'application. Les méthodes d'extension peuvent être utilisées pour ajouter des fonctionnalités spécifiques à chaque couche d'application sans charger l'objet avec des méthodes non nécessaires ou non souhaitées dans d'autres couches.

C#

```
public class DomainEntity
{
    public int Id { get; set; }
    public string FirstName { get; set; }
```

```
    public string LastName { get; set; }
}

static class DomainEntityExtensions
{
    static string FullName(this DomainEntity value)
        => $"{value.FirstName} {value.LastName}";
}
```

Extension des types prédéfinis

Plutôt que de créer des objets lorsque des fonctionnalités réutilisables doivent être créées, nous pouvons souvent étendre un type existant, tel qu'un type .NET ou CLR. Par exemple, si nous n'utilisons pas de méthodes d'extension, nous pouvons créer une classe `Engine` ou `Query` pour exécuter une requête sur SQL Server qui peut être appelé à plusieurs emplacements dans notre code. Toutefois, nous pouvons à la place étendre la classe System.Data.SqlClient.SqlConnection à l'aide de méthodes d'extension pour exécuter cette requête à partir de n'importe où nous avons une connexion à SQL Server. D'autres exemples consistent à ajouter des fonctionnalités courantes à la classe System.String, d'étendre les fonctionnalités de traitement des données des objets System.IO.File et System.IO.Stream d'objets System.Exception pour des fonctionnalités de gestion des erreurs spécifiques. Ces types de cas d'usage ne sont limités que par votre imagination et votre bon sens.

L'extension de types prédéfinis peut être difficile avec les types `struct`, car ils sont transmis par valeur aux méthodes. Cela signifie que toutes les modifications apportées au struct sont apportées à une copie du struct. Ces modifications ne sont pas visibles une fois la méthode d'extension terminée. Vous pouvez ajouter le modificateur `ref` au premier argument d'une méthode d'extension. L'ajout du modificateur `ref` signifie que le premier argument est passé par référence. Cela vous permet d'écrire des méthodes d'extension qui modifient l'état du struct en cours d'extension.

Instructions générales

Bien qu'il soit toujours préférable d'ajouter des fonctionnalités en modifiant le code d'un objet ou en dérivant un nouveau type chaque fois qu'il est possible et raisonnable de le faire, les méthodes d'extension sont devenues une option essentielle pour créer des fonctionnalités réutilisables dans l'écosystème .NET. Dans les cas où la source d'origine n'est pas sous votre contrôle, lorsqu'un objet dérivé est inapproprié ou impossible, ou lorsque la fonctionnalité ne doit pas être exposée au-delà de son étendue applicable, les méthodes d'extension sont un excellent choix.

Pour plus d'informations sur les types dérivés, consultez Héritage.

Lors de l'utilisation d'une méthode d'extension pour étendre un type dont vous ne contrôlez pas le code source, vous risquez d'interrompre votre méthode d'extension en cas de modification dans l'implémentation du type.

Si vous implémentez des méthodes d'extension pour un type donné, prenez en compte les points suivants :

- Une méthode d'extension ne sera jamais appelée si elle a la même signature qu'une méthode définie dans le type.
- Les méthodes d'extension sont mises en portée au niveau de l'espace de noms. Par exemple, si vous avez plusieurs classes statiques qui contiennent des méthodes d'extension dans un espace de noms unique nommé `Extensions`, elles seront toutes mises en portée par la directive `using Extensions;`.

Pour une bibliothèque de classes que vous avez implémentée, vous ne devez pas utiliser de méthodes d'extension pour éviter d'incrémenter le numéro de version d'un assembly. Si vous souhaitez ajouter une fonctionnalité importante à une bibliothèque dont le code source vous appartient, vous devez suivre les directives .NET relatives à la gestion de version des assemblys. Pour plus d'informations, consultez Versioning des assemblys.

Voir aussi

- Guide de programmation C#
- Exemples de programmation parallèle (il s'agit de nombreux exemples de méthodes d'extension)
- Expressions lambda
- Vue d'ensemble des opérateurs de requête standard
- Règles de conversion pour les paramètres Instance et leur impact
- Interopérabilité des méthodes d'extension entre les langages
- Méthodes d'extension et délégués curryfiés
- Extension method Binding and Error reporting

Guide pratique pour implémenter et appeler une méthode d'extension personnalisée (Guide de programmation C#)

Article • 10/05/2023

Cette rubrique montre comment implémenter vos propres méthodes d'extension pour n'importe quel type .NET. Le code client peut utiliser vos méthodes d'extension en ajoutant une référence à la DLL qui les contient, et en ajoutant une directive using qui spécifie l'espace de noms dans lequel les méthodes d'extension sont définies.

Pour définir et appeler la méthode d'extension

1. Définissez une classe statiquepour contenir la méthode d'extension.

   La classe doit être visible par le code client. Pour plus d'informations sur les règles d'accessibilité, consultez Modificateurs d'accès.

2. Implémentez la méthode d'extension en tant que méthode statique avec au moins la même visibilité que la classe conteneur.

3. Le premier paramètre de la méthode spécifie le type sur lequel la méthode opère. Il doit être précédé du modificateur this.

4. Dans le code appelant, ajoutez une directive `using` pour spécifier l'espace de noms qui contient la classe de méthode d'extension.

5. Appelez les méthodes comme s'il s'agissait de méthodes d'instance sur le type.

   Notez que le premier paramètre n'est pas spécifié par le code appelant, car il représente le type sur lequel l'opérateur est appliqué, et le compilateur connaît déjà le type de votre objet. Vous devez fournir des arguments uniquement pour les paramètres 2 à `n`.

Exemple

L'exemple suivant implémente une méthode d'extension nommée `WordCount` dans la classe `CustomExtensions.StringExtension`. La méthode opère sur la classe String, qui est spécifiée comme premier paramètre de méthode. L'espace de noms `CustomExtensions`

est importé dans l'espace de noms d'application, et la méthode est appelée à l'intérieur de la méthode `Main`.

C#

```
using System.Linq;
using System.Text;
using System;

namespace CustomExtensions
{
    // Extension methods must be defined in a static class.
    public static class StringExtension
    {
        // This is the extension method.
        // The first parameter takes the "this" modifier
        // and specifies the type for which the method is defined.
        public static int WordCount(this string str)
        {
            return str.Split(new char[] {' ', '.','?'},
StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
namespace Extension_Methods_Simple
{
    // Import the extension method namespace.
    using CustomExtensions;
    class Program
    {
        static void Main(string[] args)
        {
            string s = "The quick brown fox jumped over the lazy dog.";
            // Call the method as if it were an
            // instance method on the type. Note that the first
            // parameter is not specified by the calling code.
            int i = s.WordCount();
            System.Console.WriteLine("Word count of s is {0}", i);
        }
    }
}
```

Sécurité .NET

Les méthodes d'extension ne présentent aucune faille de sécurité spécifique. Elles ne peuvent jamais être utilisées pour emprunter l'identité des méthodes existantes sur un type, car toutes les collisions de noms sont résolues en faveur de l'instance ou de la méthode statique définie par le type lui-même. Les méthodes d'extension ne peuvent pas accéder à des données privées dans la classe étendue.

Voir aussi

- Guide de programmation C#
- Méthodes d'extension
- LINQ (Language Integrated Query)
- Classes statiques et membres de classe statique
- protected
- internal
- public
- this
- namespace

Guide pratique pour créer une méthode pour une énumération (Guide de programmation C#)

Article • 07/04/2023

Vous pouvez utiliser des méthodes d'extension pour ajouter des fonctionnalités propres à un type enum particulier.

Exemple

Dans l'exemple suivant, l'énumération `Grades` représente les notes qu'un étudiant peut obtenir dans une classe. Une méthode d'extension nommée `Passing` est ajoutée au type `Grades` pour que chaque instance de ce type « sache » maintenant si elle représente une note au-dessus de la moyenne.

C#

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Linq;

namespace EnumExtension
{
    // Define an extension method in a non-nested static class.
    public static class Extensions
    {
        public static Grades minPassing = Grades.D;
        public static bool Passing(this Grades grade)
        {
            return grade >= minPassing;
        }
    }

    public enum Grades { F = 0, D=1, C=2, B=3, A=4 };
    class Program
    {
        static void Main(string[] args)
        {
            Grades g1 = Grades.D;
            Grades g2 = Grades.F;
            Console.WriteLine("First {0} a passing grade.", g1.Passing() ?
"is" : "is not");
            Console.WriteLine("Second {0} a passing grade.", g2.Passing() ?
"is" : "is not");
```

```
            Extensions.minPassing = Grades.C;
            Console.WriteLine("\r\nRaising the bar!\r\n");
            Console.WriteLine("First {0} a passing grade.", g1.Passing() ?
"is" : "is not");
            Console.WriteLine("Second {0} a passing grade.", g2.Passing() ?
"is" : "is not");
        }
    }
  }
/* Output:
    First is a passing grade.
    Second is not a passing grade.

    Raising the bar!

    First is not a passing grade.
    Second is not a passing grade.
 */
```

Notez que la classe `Extensions` contient également une variable statique qui est mise à jour de manière dynamique, et que la valeur de retour de la méthode d'extension reflète la valeur actuelle de cette variable. Cela démontre que, dans les coulisses, les méthodes d'extension sont appelées directement sur la classe statique dans laquelle elles sont définies.

Voir aussi

- Guide de programmation C#
- Méthodes d'extension

Arguments nommés et facultatifs (Guide de programmation C#)

Article • 09/05/2023

Les *arguments nommés* vous permettent de spécifier un argument pour un paramètre en faisant correspondre l'argument avec son nom plutôt qu'avec sa position dans la liste de paramètres. Les *arguments facultatifs* vous permettent d'omettre des arguments pour certains paramètres. Les deux techniques peuvent être utilisées avec les méthodes, les indexeurs, les constructeurs et les délégués.

Quand vous utilisez des arguments nommés et facultatifs, ils sont évalués selon leur ordre d'affichage dans la liste d'arguments, et non dans la liste de paramètres.

Les paramètres nommés et facultatifs vous permettent de fournir des arguments pour les paramètres sélectionnés. Cette fonctionnalité facilite considérablement les appels aux interfaces COM telles que les API Microsoft Office Automation.

Arguments nommés

Avec les arguments nommés, vous n'avez plus à faire correspondre l'ordre des arguments avec l'ordre des paramètres dans les listes de paramètres des méthodes appelées. L'argument pour chaque paramètre peut être spécifié par le nom du paramètre. Par exemple, une fonction qui imprime les détails d'une commande (par exemple, le nom du vendeur, le nom du produit et le numéro de commande) peut être appelée en envoyant des arguments par position, dans l'ordre défini par la fonction.

C#

```
PrintOrderDetails("Gift Shop", 31, "Red Mug");
```

Si vous ne vous souvenez pas de l'ordre des paramètres, mais que vous connaissez leur nom, vous pouvez envoyer les arguments dans n'importe quel ordre.

C#

```
PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift
Shop");
PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop", orderNum:
31);
```

Les arguments nommés améliorent également la lisibilité de votre code en identifiant ce que chaque argument représente. Dans l'exemple de méthode ci-dessous, `sellerName` ne peut pas être une valeur Null ou un espace blanc. `sellerName` et `productName` étant tous deux des types de chaînes, au lieu d'envoyer les arguments par position, il est plus logique d'utiliser des arguments nommés pour lever l'ambiguïté entre les deux et réduire les risques de confusion pour toute personne lisant le code.

Les arguments nommés, quand ils sont utilisés avec des arguments de position, sont valides tant que :

- Ils ne sont pas suivis d'arguments de position, ou

  ```
  C#

  PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
  ```

- ils sont utilisés dans la position correcte. Dans l'exemple ci-dessous, le paramètre `orderNum` est à la position correcte, mais il n'est pas explicitement nommé.

  ```
  C#

  PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red Mug");
  ```

Les arguments positionnels qui suivent des arguments nommés dans le désordre ne sont pas valides.

```
C#

// This generates CS1738: Named argument specifications must appear after
all fixed arguments have been specified.
PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");
```

Exemple

Le code suivant implémente les exemples de cette section, ainsi que d'autres exemples.

```
C#

class NamedExample
{
    static void Main(string[] args)
    {
        // The method can be called in the normal way, by using positional
arguments.
        PrintOrderDetails("Gift Shop", 31, "Red Mug");
```

```
        // Named arguments can be supplied for the parameters in any order.
        PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName:
"Gift Shop");
        PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop",
orderNum: 31);

        // Named arguments mixed with positional arguments are valid
        // as long as they are used in their correct position.
        PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
        PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red
Mug");
        PrintOrderDetails("Gift Shop", orderNum: 31, "Red Mug");

        // However, mixed arguments are invalid if used out-of-order.
        // The following statements will cause a compiler error.
        // PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");
        // PrintOrderDetails(31, sellerName: "Gift Shop", "Red Mug");
        // PrintOrderDetails(31, "Red Mug", sellerName: "Gift Shop");
    }

    static void PrintOrderDetails(string sellerName, int orderNum, string
productName)
    {
        if (string.IsNullOrWhiteSpace(sellerName))
        {
            throw new ArgumentException(message: "Seller name cannot be null
or empty.", paramName: nameof(sellerName));
        }

        Console.WriteLine($"Seller: {sellerName}, Order #: {orderNum},
Product: {productName}");
    }
}
```

Arguments facultatifs

La définition d'une méthode, d'un constructeur, d'un indexeur ou d'un délégué peut spécifier que ses paramètres sont obligatoires ou facultatifs. Chaque appel doit fournir des arguments pour tous les paramètres obligatoires, mais peut omettre les arguments des paramètres facultatifs.

Dans sa définition, chaque paramètre facultatif a une valeur par défaut. Si aucun argument n'est envoyé pour ce paramètre, la valeur par défaut est utilisée. Une valeur par défaut doit être l'un des types d'expressions suivants :

- une expression constante ;
- une expression de la forme `new ValType()`, où `ValType` est un type valeur (par exemple, enum ou struct) ;

- une expression de la forme default(ValType), où `ValType` est un type valeur.

Les paramètres facultatifs sont définis à la fin de la liste de paramètres, après tous les paramètres obligatoires. Si l'appelant fournit un argument pour l'un des paramètres d'une série de paramètres facultatifs, il doit fournir des arguments pour tous les paramètres facultatifs précédents. Les intervalles séparés par des virgules ne sont pas autorisés dans la liste d'arguments. Par exemple, dans le code suivant, la méthode d'instance `ExampleMethod` est définie avec un paramètre obligatoire et deux paramètres facultatifs.

C#

```
public void ExampleMethod(int required, string optionalstr = "default
string",
    int optionalint = 10)
```

L'appel suivant à `ExampleMethod` provoque une erreur du compilateur, car un argument est fourni pour le troisième paramètre, mais pas pour le deuxième.

C#

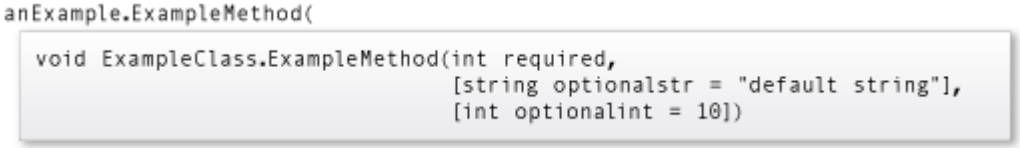
```
//anExample.ExampleMethod(3, ,4);
```

Toutefois, si vous connaissez le nom du troisième paramètre, vous pouvez utiliser un argument nommé pour effectuer la tâche.

C#

```
anExample.ExampleMethod(3, optionalint: 4);
```

IntelliSense indique les paramètres facultatifs par des crochets, comme illustré dans l'exemple suivant :



> ⓘ **Notes**
>
> Vous pouvez aussi déclarer des paramètres facultatifs à l'aide de la classe **OptionalAttribute** .NET. Les paramètres `OptionalAttribute` ne nécessitent pas de valeur par défaut.

Exemple

Dans l'exemple suivant, le constructeur associé à `ExampleClass` a un seul paramètre, qui est facultatif. La méthode d'instance `ExampleMethod` a un paramètre obligatoire (`required`) et deux paramètres facultatifs (`optionalstr` et `optionalint`). Le code dans `Main` montre les différentes façons dont le constructeur et la méthode peuvent être appelés.

C#

```
namespace OptionalNamespace
{
    class OptionalExample
    {
        static void Main(string[] args)
        {
            // Instance anExample does not send an argument for the
constructor's
            // optional parameter.
            ExampleClass anExample = new ExampleClass();
            anExample.ExampleMethod(1, "One", 1);
            anExample.ExampleMethod(2, "Two");
            anExample.ExampleMethod(3);

            // Instance anotherExample sends an argument for the
constructor's
            // optional parameter.
            ExampleClass anotherExample = new ExampleClass("Provided name");
            anotherExample.ExampleMethod(1, "One", 1);
            anotherExample.ExampleMethod(2, "Two");
            anotherExample.ExampleMethod(3);

            // The following statements produce compiler errors.

            // An argument must be supplied for the first parameter, and it
            // must be an integer.
            //anExample.ExampleMethod("One", 1);
            //anExample.ExampleMethod();

            // You cannot leave a gap in the provided arguments.
            //anExample.ExampleMethod(3, ,4);
            //anExample.ExampleMethod(3, 4);

            // You can use a named parameter to make the previous
            // statement work.
            anExample.ExampleMethod(3, optionalint: 4);
        }
    }

    class ExampleClass
    {
        private string _name;
```

```
        // Because the parameter for the constructor, name, has a default
        // value assigned to it, it is optional.
        public ExampleClass(string name = "Default name")
        {
            _name = name;
        }

        // The first parameter, required, has no default value assigned
        // to it. Therefore, it is not optional. Both optionalstr and
        // optionalint have default values assigned to them. They are
optional.
        public void ExampleMethod(int required, string optionalstr =
"default string",
            int optionalint = 10)
        {
            Console.WriteLine(
                $"{_name}: {required}, {optionalstr}, and {optionalint}.");
        }
    }

    // The output from this example is the following:
    // Default name: 1, One, and 1.
    // Default name: 2, Two, and 10.
    // Default name: 3, default string, and 10.
    // Provided name: 1, One, and 1.
    // Provided name: 2, Two, and 10.
    // Provided name: 3, default string, and 10.
    // Default name: 3, default string, and 4.
}
```

Le code précédent montre un certain nombre d'exemples où les paramètres facultatifs ne sont pas appliqués correctement. Le premier montre qu'un argument doit être fourni pour le premier paramètre, qui est obligatoire.

interfaces COM

Avec la prise en charge des objets dynamiques, les arguments nommés et facultatifs améliorent grandement l'interopérabilité avec les API COM telles que les API Office Automation.

Par exemple, la méthode AutoFormat de l'interface Range de Microsoft Office Excel comporte sept paramètres, tous facultatifs. Ces paramètres sont indiqués dans l'illustration suivante :

```
excelApp.get_Range("A1", "B4").AutoFormat(
```

```
dynamic Range.AutoFormat([Excel.XlRangeAutoFormat Format = 1],
[object Number = Type.Missing], [object Font = Type.Missing],
[object Alignment = Type.Missing], [object Border = Type.Missing],
[object Pattern = Type.Missing], [object Width = Type.Missing])
```

Toutefois, vous pouvez considérablement simplifier l'appel à `AutoFormat` en utilisant les arguments nommés et facultatifs. Les arguments nommés et facultatifs vous permettent d'omettre l'argument d'un paramètre facultatif si vous ne souhaitez pas changer la valeur par défaut du paramètre. Dans l'appel suivant, une valeur est spécifiée pour un seul des sept paramètres.

C#

```
var excelApp = new Microsoft.Office.Interop.Excel.Application();
excelApp.Workbooks.Add();
excelApp.Visible = true;

var myFormat =

Microsoft.Office.Interop.Excel.XlRangeAutoFormat.xlRangeAutoFormatAccounting
1;

excelApp.Range["A1", "B4"].AutoFormat( Format: myFormat );
```

Pour obtenir plus d'informations et des exemples, consultez Guide pratique pour utiliser des arguments nommés et facultatifs dans la programmation Office et Guide pratique pour accéder aux objets Office Interop à l'aide des fonctionnalités C#.

Résolution de surcharge

L'utilisation d'arguments nommés et facultatifs affecte la résolution de surcharge des manières suivantes :

- Une méthode, un indexeur ou un constructeur est un candidat pour l'exécution si chacun de ses paramètres est facultatif ou correspond, par nom ou par position, à un seul argument dans l'instruction appelante, et que cet argument peut être converti vers le type du paramètre.
- Si plusieurs candidats sont trouvés, les règles de résolution de surcharge des conversions préférées sont appliquées aux arguments qui sont explicitement spécifiés. Les arguments omis pour les paramètres facultatifs sont ignorés.
- Si deux candidats sont jugés de qualité équivalente, la préférence va à celui qui n'a pas de paramètres facultatifs pour lesquels des arguments ont été omis dans l'appel. La résolution de surcharges préfère généralement les candidats qui ont moins de paramètres.

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Comment : utiliser des arguments nommés et facultatifs dans la programmation Office

Article • 09/03/2023

Les arguments nommés et les arguments facultatifs, rendent la programmation en C# plus pratique, plus souple et plus lisible. De plus, ces fonctionnalités facilitent considérablement l'accès aux interfaces COM, telles que les API Microsoft Office Automation.

> **ⓘ Important**
>
> VSTO est basé sur **.NET Framework**. Les compléments COM peuvent également être écrits avec .NET Framework. Les compléments Office ne peuvent pas être créés avec **.NET Core et .NET 5+**, les dernières versions de .NET. En effet, .NET Core et .NET 5+ ne peuvent pas fonctionner avec .NET Framework dans le même processus, et cela peut entraîner des échecs de chargement des compléments. Vous pouvez continuer d'utiliser .NET Framework pour écrire des compléments VSTO et COM pour Office. Microsoft ne mettra pas à jour VSTO ou la plateforme de compléments COM pour utiliser .NET Core ou .NET 5+. Vous pouvez utiliser .NET Core et .NET 5+, dont ASP.NET Core, pour créer le côté serveur des **compléments web Office**.

Dans l'exemple suivant, la méthode ConvertToTable a 16 paramètres qui représentent les caractéristiques d'une table, comme le nombre de colonnes et de lignes, la mise en forme, les bordures, les polices et les couleurs. Ces 16 paramètres sont facultatifs, car la plupart du temps, il n'est pas nécessaire de spécifier des valeurs pour chacun d'eux. Mais sans arguments nommés et facultatifs, vous devez fournir une valeur ou une valeur d'espace réservé. Avec les arguments nommés et facultatifs, vous spécifiez des valeurs uniquement pour les paramètres nécessaires à votre projet.

Pour que vous puissiez effectuer ces procédures, Microsoft Office Word doit être installé sur votre ordinateur.

> **ⓘ Notes**
>
> Il est possible que pour certains des éléments de l'interface utilisateur de Visual Studio, votre ordinateur affiche des noms ou des emplacements différents de ceux indiqués dans les instructions suivantes. L'édition de Visual Studio dont vous

disposez et les paramètres que vous utilisez déterminent ces éléments. Pour plus d'informations, consultez **Personnalisation de l'IDE**.

Créer une application console

Démarrez Visual Studio. Dans le menu **Fichier** , pointez sur **Nouveau**, puis sélectionnez **Projet**. Dans le volet **Catégories de modèles**, développez **C#**, puis sélectionnez **Windows**. Regardez en haut du volet **Modèles** pour vérifier que **.NET Framework 4**, ou version ultérieure, apparaît dans la zone **Framework cible**. Dans le volet **Modèles**, sélectionnez **Application console**. Attribuez un nom à votre projet dans le champ **Nom**. Sélectionnez **OK**. Le nouveau projet s'affiche dans l'**Explorateur de solutions**.

Ajouter une référence

Dans l'**Explorateur de solutions**, cliquez avec le bouton droit sur le nom de votre projet, puis sélectionnez **Ajouter une référence**. La boîte de dialogue **Ajouter une référence** s'affiche. Dans la page **.NET**, sélectionnez **Microsoft.Office.Interop.Word** dans la liste **Nom du composant**. Sélectionnez **OK**.

Ajouter les directives using nécessaires

Dans l'**Explorateur de solutions**, cliquez avec le bouton droit sur le fichier *Program.cs*, puis sélectionnez **Afficher le code**. Ajoutez les directives `using` suivantes en début du fichier de code :

```
C#
```

```
using Word = Microsoft.Office.Interop.Word;
```

Afficher du texte dans un document Word

Dans la classe `Program` du fichier *Program.cs*, ajoutez la méthode suivante pour créer une application Word et un document Word. La méthode Add comprend quatre paramètres facultatifs. Cet exemple utilise leurs valeurs par défaut. Par conséquent, aucun argument n'est nécessaire dans l'instruction appelante.

```
C#
```

```
static void DisplayInWord()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;
    // docs is a collection of all the Document objects currently
    // open in Word.
    Word.Documents docs = wordApp.Documents;

    // Add a document to the collection and name it doc.
    Word.Document doc = docs.Add();
}
```

Ajoutez le code suivant à la fin de la méthode pour définir l'emplacement d'affichage du texte dans le document, ainsi que le texte à afficher :

C#

```
// Define a range, a contiguous area in the document, by specifying
// a starting and ending character position. Currently, the document
// is empty.
Word.Range range = doc.Range(0, 0);

// Use the InsertAfter method to insert a string at the end of the
// current range.
range.InsertAfter("Testing, testing, testing. . .");
```

Exécution de l'application

Ajoutez l'instruction suivante à Main :

C#

```
DisplayInWord();
```

Appuyez sur `Ctrl`+`F5` pour exécuter le projet. Un document Word contenant le texte spécifié s'affiche.

Convertir le texte en tableau

Utilisez la méthode `ConvertToTable` pour inclure le texte dans un tableau. La méthode comporte 16 paramètres optionnels. IntelliSense place les paramètres facultatifs entre crochets, comme illustré dans l'exemple suivant.

```
range.ConvertToTable(
```

```
Word.Table Range.ConvertToTable([ref object Separator = Type.Missing], [ref object NumRows =
Type.Missing], [ref object NumColumns = Type.Missing], [ref object InitialColumnWidth =
Type.Missing], [ref object Format = Type.Missing], [ref object ApplyBorders = Type.Missing],
[ref object ApplyShading = Type.Missing], [ref object ApplyFont = Type.Missing], [ref object
ApplyColor = Type.Missing], [ref object ApplyHeadingsRows = Type.Missing], [ref object
ApplyLastRow = Type.Missing], [ref object ApplyFirstColumn = Type.Missing], [ref object
ApplyLastColumn = Type.Missing], [ref object AutoFit = Type.Missing], [ref object
AutoFitBehavior = Type.Missing], [ref object DefaultTableBehavior = Type.Missing])
```

Les arguments nommés et facultatifs permettent de spécifier des valeurs uniquement pour les paramètres que vous voulez modifier. Ajoutez le code suivant à la fin de la méthode `DisplayInWord` pour créer un tableau. L'argument spécifie que les virgules présentes dans le texte de la chaîne comprise dans `range` séparent les cellules du tableau.

C#

```
// Convert to a simple table. The table will have a single row with
// three columns.
range.ConvertToTable(Separator: ",");
```

Appuyez sur `Ctrl`+`F5` pour exécuter le projet.

Tester d'autres paramètres

Modifiez le tableau de sorte qu'il comporte une colonne et trois lignes, remplacez la dernière ligne dans `DisplayInWord` par l'instruction suivante, puis appuyez sur `CTRL`+`F5` .

C#

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1);
```

Spécifiez un format prédéfini pour le tableau, remplacez la dernière ligne dans `DisplayInWord` par l'instruction suivante, puis appuyez sur `CTRL`+`F5`. Le format peut être n'importe quelle constante WdTableFormat.

C#

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1,
    Format: Word.WdTableFormat.wdTableFormatElegant);
```

Exemple

Le code suivant contient l'exemple complet :

C#

```
using System;
using Word = Microsoft.Office.Interop.Word;

namespace OfficeHowTo
{
    class WordProgram
    {
        static void Main(string[] args)
        {
            DisplayInWord();
        }

        static void DisplayInWord()
        {
            var wordApp = new Word.Application();
            wordApp.Visible = true;
            // docs is a collection of all the Document objects currently
            // open in Word.
            Word.Documents docs = wordApp.Documents;

            // Add a document to the collection and name it doc.
            Word.Document doc = docs.Add();

            // Define a range, a contiguous area in the document, by
specifying
            // a starting and ending character position. Currently, the
document
            // is empty.
            Word.Range range = doc.Range(0, 0);

            // Use the InsertAfter method to insert a string at the end of
the
            // current range.
            range.InsertAfter("Testing, testing, testing. . .");

            // You can comment out any or all of the following statements to
            // see the effect of each one in the Word document.

            // Next, use the ConvertToTable method to put the text into a
table.
            // The method has 16 optional parameters. You only have to
specify
            // values for those you want to change.

            // Convert to a simple table. The table will have a single row
with
            // three columns.
            range.ConvertToTable(Separator: ",");

            // Change to a single column with three rows..
```

```
            range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns:
1);

            // Format the table.
            range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns:
1,
                Format: Word.WdTableFormat.wdTableFormatElegant);
        }
    }
}
```

Constructeurs (Guide de programmation C#)

Article • 07/04/2023

Chaque fois qu'une instance de classe ou qu'un struct est créé, son constructeur est appelé. Une classe ou un struct peut avoir plusieurs constructeurs qui prennent des arguments différents. Les constructeurs permettent au programmeur de définir des valeurs par défaut, de limiter l'instanciation et d'écrire un code flexible et facile à lire. Pour obtenir plus d'informations et d'exemples, consultez Constructeurs d'instances et Utilisation de constructeurs.

Plusieurs actions font partie de l'initialisation d'une nouvelle instance. Ces actions ont lieu dans l'ordre suivant :

1. *Les champs d'instance sont définis sur 0*. Cela est généralement effectué par le runtime.
2. *Les initialiseurs de champs s'exécutent*. Initialiseurs de champs dans l'exécution de type le plus dérivé.
3. *Les initialiseurs de champs de type de base s'exécutent*. Initialiseurs de champs commençant par la base directe via chaque type de base vers System.Object.
4. *Les constructeurs d'instances de base s'exécutent*. Tous les constructeurs d'instances, en commençant par Object.Object via chaque classe de base vers la classe de base directe.
5. *Le constructeur d'instance s'exécute*. Le constructeur d'instance pour le type s'exécute.
6. *Les initialiseurs d'objets s'exécutent*. Si l'expression inclut des initialiseurs d'objets, ceux-ci s'exécutent après l'exécution du constructeur d'instance. Les initialiseurs d'objets s'exécutent dans l'ordre textuel.

Les actions précédentes se produisent lorsqu'une nouvelle instance est initialisée. Si une nouvelle instance de `struct` est définie sur sa valeur `default`, tous les champs d'instance sont définis sur 0.

Si le constructeur statique n'a pas été exécuté, le constructeur statique s'exécute avant l'une des actions du constructeur d'instance.

Syntaxe du constructeur

Un constructeur est une méthode dont le nom est le même que celui de son type. Sa signature de méthode inclut uniquement un modificateur d'accès facultatif, le nom de la

méthode et sa liste de paramètres ; elle n'inclut pas de type de retour. L'exemple suivant affiche le constructeur d'une classe nommée `Person`.

C#

```
public class Person
{
   private string last;
   private string first;

   public Person(string lastName, string firstName)
   {
      last = lastName;
      first = firstName;
   }

   // Remaining implementation of Person class.
}
```

Si un constructeur peut être implémenté comme une seule instruction, vous pouvez utiliser une définition de corps d'expression. L'exemple suivant définit une classe `Location` dont le constructeur a un seul paramètre de chaîne nommé *name*. La définition de corps d'expression assigne l'argument au champ `locationName`.

C#

```
public class Location
{
   private string locationName;

   public Location(string name) => Name = name;

   public string Name
   {
      get => locationName;
      set => locationName = value;
   }
}
```

Constructeurs statiques

Les exemples précédents ont tous montré des constructeurs d'instances qui permettent de créer un objet. Une classe ou un struct peut également avoir un constructeur statique qui initialise les membres statiques du type. Les constructeurs statiques sont sans paramètre. Si vous ne fournissez pas de constructeur statique pour initialiser les champs

statiques, le compilateur C# initialise tous les champs statiques vers leur valeur par défaut telle que répertoriée dans l'article Valeurs par défaut des types C#.

L'exemple suivant utilise un constructeur statique pour initialiser un champ statique.

C#

```
public class Adult : Person
{
   private static int minimumAge;

   public Adult(string lastName, string firstName) : base(lastName,
firstName)
   { }

   static Adult()
   {
      minimumAge = 18;
   }

   // Remaining implementation of Adult class.
}
```

Vous pouvez également définir un constructeur statique avec une définition de corps d'expression, comme l'illustre l'exemple suivant.

C#

```
public class Child : Person
{
   private static int maximumAge;

   public Child(string lastName, string firstName) : base(lastName,
firstName)
   { }

   static Child() => maximumAge = 18;

   // Remaining implementation of Child class.
}
```

Pour obtenir plus d'informations et d'exemples, consultez Constructeurs statiques.

Dans cette section

Utilisation de constructeurs

Constructeurs d'instances

Constructeurs privés

Constructeurs statiques

Comment : écrire un constructeur de copie

Voir aussi

- Guide de programmation C#
- Système de types C#
- Finaliseurs
- static
- Why Do Initializers Run In The Opposite Order As Constructors? Part One

Utilisation de constructeurs (Guide de programmation C#)

Article • 01/06/2023

Lorsqu'une classe ou un struct est instancié, son constructeur est appelé. Les constructeurs portent le même nom que la classe ou le struct, et ils initialisent généralement les membres de données du nouvel objet.

Dans l'exemple suivant, une classe nommée `Taxi` est définie en utilisant un constructeur simple. Cette classe est ensuite instanciée à l'aide de l'opérateur new. Le constructeur `Taxi` est appelé par l'opérateur `new` immédiatement après l'allocation de la mémoire pour le nouvel objet.

C#

```
public class Taxi
{
    public bool IsInitialized;

    public Taxi()
    {
        IsInitialized = true;
    }
}

class TestTaxi
{
    static void Main()
    {
        Taxi t = new Taxi();
        Console.WriteLine(t.IsInitialized);
    }
}
```

Un constructeur qui ne prend pas de paramètres est appelé *constructeur sans paramètre*. Les constructeurs sans paramètre sont appelés chaque fois qu'un objet est instancié à l'aide de l'opérateur `new` et qu'aucun argument n'est fourni à `new`. C# 12 introduit les *constructeurs principaux*. Un constructeur principal spécifie les paramètres qui doivent être fournis pour initialiser un nouvel objet. Pour plus d'informations, consultez Constructeurs d'instances.

À moins d'être statiques, les classes sans constructeur se voient attribuer un constructeur public sans paramètre par le compilateur C#, afin d'activer l'instanciation

de classe. Pour plus d'informations, consultez la page Classes statiques et membres de classes statiques.

Vous pouvez empêcher qu'une classe soit instanciée en rendant le constructeur privé, comme suit :

C#

```
class NLog
{
    // Private Constructor:
    private NLog() { }

    public static double e = Math.E;  //2.71828...
}
```

Pour plus d'informations, consultez Constructeurs privés.

Les constructeurs pour les types struct ressemblent aux constructeurs de classe. Lorsqu'un type de struct est instancié avec `new`, un constructeur est appelé. Lorsque `struct` est défini sur sa valeur `default`, le runtime initialise toutes les mémoires dans le struct sur 0. Avant C# 10, `structs` ne peut pas contenir de constructeur explicite sans paramètre, car un constructeur est fourni automatiquement par le compilateur. Pour plus d'informations, consultez la section Initialisation de struct et valeurs par défaut de l'article Types de structure.

Le code suivant utilise le constructeur sans paramètre pour Int32, afin que vous soyez assuré que l'entier est initialisé :

C#

```
int i = new int();
Console.WriteLine(i);
```

Toutefois, le code suivant provoque une erreur de compilateur, car il n'utilise pas `new` et il essaie d'utiliser un objet qui n'a pas été initialisé :

C#

```
int i;
Console.WriteLine(i);
```

Les objets basés sur des `structs` (notamment tous les types numériques intégrés) peuvent également être initialisés ou assignés, puis utilisés, comme dans l'exemple

suivant :

```
C#
```

```
int a = 44;  // Initialize the value type...
int b;
b = 33;      // Or assign it before using it.
Console.WriteLine("{0}, {1}", a, b);
```

Les classes et les structs peuvent définir des constructeurs qui prennent des paramètres, y compris les constructeurs principaux. Les constructeurs qui prennent des paramètres doivent être appelés à l'aide d'une instruction `new` ou d'une instruction base. Les classes et les structs peuvent également définir plusieurs constructeurs, et ni les classes ni les structs ne sont nécessaires pour définir un constructeur sans paramètre. Par exemple :

```
C#
```

```
public class Employee
{
    public int Salary;

    public Employee() { }

    public Employee(int annualSalary)
    {
        Salary = annualSalary;
    }

    public Employee(int weeklySalary, int numberOfWeeks)
    {
        Salary = weeklySalary * numberOfWeeks;
    }
}
```

Cette classe peut être créée à l'aide de l'une ou l'autre des instructions suivantes :

```
C#
```

```
Employee e1 = new Employee(30000);
Employee e2 = new Employee(500, 52);
```

Un constructeur peut utiliser le mot clé `base` pour appeler le constructeur d'une classe de base. Par exemple :

```
C#
```

```
public class Manager : Employee
{
    public Manager(int annualSalary)
        : base(annualSalary)
    {
        //Add further instructions here.
    }
}
```

Dans cet exemple, le constructeur de la classe de base est appelé avant que le bloc du constructeur ne soit exécuté. Le mot clé `base` peut être utilisé avec ou sans paramètres. Tous les paramètres du constructeur peuvent être utilisés comme paramètres pour `base` ou comme partie d'une expression. Pour plus d'informations, consultez base.

Dans une classe dérivée, si un constructeur de classe de base n'est pas appelé explicitement à l'aide du mot clé `base`, le constructeur sans paramètre, s'il existe, est appelé implicitement. Les déclarations de constructeur suivantes sont en fait les mêmes :

C#

```
public Manager(int initialData)
{
    //Add further instructions here.
}
```

C#

```
public Manager(int initialData)
    : base()
{
    //Add further instructions here.
}
```

Si une classe de base n'offre pas de constructeur sans paramètre, la classe dérivée doit faire un appel explicite à un constructeur de base à l'aide de `base`.

Un constructeur peut appeler un autre constructeur dans le même objet à l'aide du mot clé this. Comme `base`, `this` peut être utilisé avec ou sans paramètres, et tous les paramètres dans le constructeur sont disponibles comme paramètres pour `this` ou comme partie d'une expression. Par exemple, le deuxième constructeur de l'exemple précédent peut être récrit à l'aide de `this` :

C#

```
public Employee(int weeklySalary, int numberOfWeeks)
    : this(weeklySalary * numberOfWeeks)
{
}
```

L'utilisation du mot clé `this` dans l'exemple précédent provoque l'appel de ce constructeur :

C#

```
public Employee(int annualSalary)
{
    Salary = annualSalary;
}
```

Les constructeurs peuvent être marqués comme public, private, protected, internal, protected internal ou private protected. Ces modificateurs d'accès définissent la façon dont les utilisateurs de la classe peuvent construire la classe. Pour plus d'informations, consultez Modificateurs d'accès.

Un constructeur peut être déclaré statique à l'aide du mot clé static. Les constructeurs statiques sont appelés automatiquement, juste avant que des champs statiques soient accessibles, et ils sont utilisés pour initialiser des membres de classe statique. Pour plus d'informations, consultez Constructeurs statiques.

Spécification du langage C#

Pour plus d'informations, consultez Constructeurs d'instances et Constructeurs statiques dans la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Guide de programmation C#
- Système de types C#
- Constructeurs
- Finaliseurs

Constructeurs d'instances (Guide de programmation C#)

Article • 02/06/2023

Vous déclarez un constructeur d'instance pour spécifier le code qui est exécuté lorsque vous créez une instance d'un type avec l'expression new . Pour initialiser une classe statique ou des variables statiques dans une classe non statique, vous pouvez définir un constructeur statique.

Comme le montre l'exemple suivant, vous pouvez déclarer plusieurs constructeurs d'instances dans un type :

C#

```
class Coords
{
    public Coords()
        : this(0, 0)
    {  }

    public Coords(int x, int y)
    {
        X = x;
        Y = y;
    }

    public int X { get; set; }
    public int Y { get; set; }

    public override string ToString() => $"({X},{Y})";
}

class Example
{
    static void Main()
    {
        var p1 = new Coords();
        Console.WriteLine($"Coords #1 at {p1}");
        // Output: Coords #1 at (0,0)

        var p2 = new Coords(5, 3);
        Console.WriteLine($"Coords #2 at {p2}");
        // Output: Coords #2 at (5,3)
    }
}
```

Dans l'exemple précédent, le premier constructeur sans paramètre appelle le deuxième constructeur avec les deux arguments égaux à `0`. Pour ce faire, utilisez le mot clé `this`.

Lorsque vous déclarez un constructeur d'instance dans une classe dérivée, vous pouvez appeler un constructeur d'une classe de base. Pour ce faire, utilisez le mot clé `base`, comme le montre l'exemple suivant :

C#

```
abstract class Shape
{
    public const double pi = Math.PI;
    protected double x, y;

    public Shape(double x, double y)
    {
        this.x = x;
        this.y = y;
    }

    public abstract double Area();
}

class Circle : Shape
{
    public Circle(double radius)
        : base(radius, 0)
    {  }

    public override double Area() => pi * x * x;
}

class Cylinder : Circle
{
    public Cylinder(double radius, double height)
        : base(radius)
    {
        y = height;
    }

    public override double Area() => (2 * base.Area()) + (2 * pi * x * y);
}

class Example
{
    static void Main()
    {
        double radius = 2.5;
        double height = 3.0;

        var ring = new Circle(radius);
        Console.WriteLine($"Area of the circle = {ring.Area():F2}");
        // Output: Area of the circle = 19.63
```

```
        var tube = new Cylinder(radius, height);
        Console.WriteLine($"Area of the cylinder = {tube.Area():F2}");
        // Output: Area of the cylinder = 86.39
    }
}
```

Constructeurs sans paramètre

Si une *classe* n'a aucun constructeur d'instance explicite, C# fournit un constructeur sans paramètre que vous pouvez utiliser pour instancier une instance de cette classe, comme le montre l'exemple suivant :

C#

```
public class Person
{
    public int age;
    public string name = "unknown";
}

class Example
{
    static void Main()
    {
        var person = new Person();
        Console.WriteLine($"Name: {person.name}, Age: {person.age}");
        // Output:  Name: unknown, Age: 0
    }
}
```

Ce constructeur initialise les champs et propriétés de l'instance en fonction des initialiseurs correspondants. Si un champ ou une propriété n'a pas d'initialiseur, sa valeur est définie sur la valeur par défaut du type du champ ou de la propriété. Si vous déclarez au moins un constructeur d'instance dans une classe, C# ne fournit pas de constructeur sans paramètre.

Un type de *structure* fournit toujours un constructeur sans paramètre comme suit :

- Dans C# 9.0 et antérieur, il s'agit d'un constructeur sans paramètre implicite qui produit la valeur par défaut d'un type.
- Dans C# 10 et ultérieur, il s'agit d'un constructeur sans paramètre implicite qui produit la valeur par défaut d'un type ou d'un constructeur sans paramètre déclaré explicitement. Pour plus d'informations, consultez la section Initialisation de struct et valeurs par défaut de l'article Types de structure.

Constructeurs principaux

À compter de C# 12, vous pouvez déclarer un *constructeur principal* dans les classes et les structs. Vous placez tous les paramètres entre parenthèses en suivant le nom de type :

C#

```
public class NamedItem(string name)
{
    public string Name => name;
}
```

Les paramètres d'un constructeur principal sont dans l'étendue dans le corps entier du type déclarant. Ils peuvent initialiser des propriétés ou des champs. Ils peuvent être utilisés comme variables dans des méthodes ou des fonctions locales. Ils peuvent être passés à un constructeur de base.

Un constructeur principal indique que ces paramètres sont nécessaires pour n'importe quelle instance d'un type. Tout constructeur écrit explicitement doit utiliser la syntaxe d'initialiseur `this(...)` pour appeler le constructeur principal. Cela garantit que les paramètres du constructeur principal sont définitivement attribués par tous les constructeurs. Pour tout type `class`, y compris les types `record class`, le constructeur sans paramètre implicite n'est pas émis lorsqu'un constructeur principal est présent. Pour tout type `struct`, y compris les types `record struct`, le constructeur sans paramètre implicite est toujours émis et initialise toujours tous les champs, y compris les paramètres du constructeur principal, au modèle 0 bits. Si vous écrivez un constructeur explicite sans paramètre, il doit appeler le constructeur principal. Dans ce cas, vous pouvez spécifier une valeur différente pour les paramètres du constructeur principal. Le code suivant montre des exemples de constructeurs principaux.

C#

```
// name isn't captured in Widget.
// width, height, and depth are captured as private fields
public class Widget(string name, int width, int height, int depth) :
NamedItem(name)
{
    public Widget() : this("N/A", 1,1,1) {} // unnamed unit cube

    public int WidthInCM => width;
    public int HeightInCM => height;
    public int DepthInCM => depth;

    public int Volume => width * height * depth;
}
```

Dans les types `class` et `struct`, les paramètres du constructeur principal sont disponibles n'importe où dans le corps du type. Ils peuvent être utilisés en tant que champs membres. Lorsqu'un paramètre de constructeur principal est utilisé, le compilateur capture le paramètre de constructeur dans un champ privé avec un nom généré par le compilateur. Si un paramètre de constructeur principal n'est pas utilisé dans le corps du type, aucun champ privé n'est capturé. Cette règle empêche l'allocation accidentelle de deux copies d'un paramètre de constructeur principal passé à un constructeur de base.

Si le type inclut le modificateur `record`, le compilateur synthétise plutôt une propriété publique portant le même nom que le paramètre du constructeur principal. Pour les types `record class`, si un paramètre de constructeur principal utilise le même nom qu'un constructeur principal de base, cette propriété est une propriété publique du type de base `record class`. Il n'est pas dupliqué dans le type dérivé `record class`. Ces propriétés ne sont pas générées pour les types non `record`.

Voir aussi

- Guide de programmation C#
- Classes, structs et enregistrements
- Constructeurs
- Finaliseurs
- base
- this
- Spécification des caractéristiques des constructeurs principaux

Constructeurs privés (Guide de programmation C#)

Article • 07/04/2023

Un constructeur privé est un constructeur d'instance spécial. Il est généralement utilisé dans les classes qui contiennent uniquement des membres statiques. Si une classe a un ou plusieurs constructeurs privés, mais n'a aucun constructeur public, les autres classes (à l'exception des classes imbriquées) ne peuvent pas créer d'instances de cette classe. Par exemple :

C#

```
class NLog
{
    // Private Constructor:
    private NLog() { }

    public static double e = Math.E;  //2.71828...
}
```

La déclaration du constructeur vide empêche la génération automatique d'un constructeur sans paramètre. Notez que si vous n'utilisez pas de modificateur d'accès avec le constructeur, le constructeur est toujours privé par défaut. En règle générale, le modificateur private est explicitement utilisé pour spécifier que la classe ne peut pas être instanciée.

Les constructeurs privés servent à empêcher la création d'instances d'une classe quand il n'y a pas de champs ou de méthodes d'instance (par exemple, la classe Math) ou quand une méthode est appelée pour obtenir une instance d'une classe. Si toutes les méthodes dans la classe sont statiques, définissez la classe entière comme statique. Pour plus d'informations, consultez Classes statiques et membres de classe statique.

Exemple

L'exemple suivant montre une classe qui utilise un constructeur privé.

C#

```
public class Counter
{
    private Counter() { }

    public static int currentCount;
```

```
    public static int IncrementCount()
    {
        return ++currentCount;
    }
}

class TestCounter
{
    static void Main()
    {
        // If you uncomment the following statement, it will generate
        // an error because the constructor is inaccessible:
        // Counter aCounter = new Counter();   // Error

        Counter.currentCount = 100;
        Counter.IncrementCount();
        Console.WriteLine("New count: {0}", Counter.currentCount);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: New count: 101
```

Notez que si vous décommentez l'instruction suivante dans l'exemple, une erreur est générée, car le constructeur a un niveau de protection qui le rend inaccessible :

C#

```
// Counter aCounter = new Counter();   // Error
```

Voir aussi

- Guide de programmation C#
- Système de types C#
- Constructeurs
- Finaliseurs
- private
- public

Constructeurs statiques (Guide de programmation C#)

Article • 07/04/2023

Un constructeur statique est utilisé pour initialiser toutes les données statiques ou effectuer une action particulière qui ne doit être effectuée qu'une seule fois. Il est automatiquement appelé avant la création de la première instance ou le référencement d'un membre statique. Un constructeur statique est appelé au maximum une fois.

C#

```
class SimpleClass
{
    // Static variable that must be initialized at run time.
    static readonly long baseline;

    // Static constructor is called at most one time, before any
    // instance constructor is invoked or member is accessed.
    static SimpleClass()
    {
        baseline = DateTime.Now.Ticks;
    }
}
```

Plusieurs actions font partie de l'initialisation statique. Ces actions ont lieu dans l'ordre suivant :

1. *Les champs statiques sont définis sur 0*. Cela est généralement effectué par le runtime.
2. *Les initialiseurs de champs statiques s'exécutent*. Initialiseurs de champs statiques dans l'exécution de type le plus dérivé.
3. *Les initialiseurs de champs statiques de type de base s'exécutent*. Initialiseurs de champs statiques commençant par la base directe via chaque type de base vers System.Object.
4. *Les constructeurs statiques de base s'exécutent*. Tous les constructeurs statiques, en commençant par Object.Object via chaque classe de base vers la classe de base directe.
5. *Le constructeur statique s'exécute*. Le constructeur statique pour le type s'exécute.

Remarques

Les constructeurs statiques ont les propriétés suivantes :

- Un constructeur statique n'utilise ni modificateurs d'accès ni paramètres.
- Une classe ou struct ne peut pas avoir de constructeur statique.
- Les constructeurs statiques ne peuvent pas être hérités ou surchargés.
- Un constructeur statique ne peut pas être appelé directement et est uniquement destiné à être appelé par le Common Language Runtime (CLR). Il est appelé automatiquement.
- L'utilisateur n'a aucun contrôle sur le moment d'exécution du constructeur statique dans le programme.
- Un constructeur statique est automatiquement appelé. Il initialise la classe avant la création de la première instance ou les membres statiques déclarés dans cette classe (et non ses classes de base) sont référencés. Un constructeur statique s'exécute avant un constructeur d'instance. Si les initialiseurs de variable de champ statique sont présents dans la classe du constructeur statique, ils sont exécutés dans l'ordre textuel dans lequel ils apparaissent dans la déclaration de classe. Les initialiseurs s'exécutent immédiatement avant l'exécution du constructeur statique.
- Si vous ne fournissez pas de constructeur statique pour initialiser les champs statiques, tous les champs statiques sont initialisés à leur valeur par défaut, conformément au Valeurs par défaut des types C#.
- Si un constructeur statique lève une exception, le runtime ne l'appelle pas une deuxième fois et le type reste non initialisé pour la durée de vie du domaine d'application. En règle générale, une exception TypeInitializationException est levée lorsqu'un constructeur statique ne parvient pas à instancier un type ou quand une exception non gérée se produit dans un constructeur statique. Pour les constructeurs statiques qui ne sont pas définis explicitement dans le code source, la résolution des problèmes peut nécessiter l'inspection du code en langage intermédiaire (IL).
- La présence d'un constructeur statique empêche l'ajout de l'attribut de type BeforeFieldInit. Cela limite l'optimisation du runtime.
- Un champ déclaré en tant que `static readonly` ne peut être attribué que dans le cadre de sa déclaration ou dans un constructeur statique. Si un constructeur statique explicite n'est pas obligatoire, initialisez les champs statiques lors de la déclaration plutôt que via un constructeur statique pour une meilleure optimisation du runtime.
- Le runtime appelle un constructeur statique au maximum une fois dans un seul domaine d'application. Cet appel est effectué dans une région verrouillée en fonction du type spécifique de la classe. Aucun mécanisme de verrouillage supplémentaire n'est nécessaire dans le corps d'un constructeur statique. Pour éviter le risque d'interblocages, ne bloquez pas le thread actuel dans les constructeurs statiques et les initialiseurs. Par exemple, n'attendez pas les tâches, les threads, attendez les descripteurs ou les événements, n'acquérez pas de

verrous et n'exécutez pas d'opérations parallèles bloquantes telles que des boucles parallèles, `Parallel.Invoke` et des requêtes LINQ parallèles.

> ⓘ **Notes**
>
> S'il n'est pas directement accessible, la présence d'un constructeur statique explicite doit être documentée pour faciliter la résolution des exceptions de l'initialisation.

Utilisation

- Généralement, on utilise des constructeurs statiques quand la classe utilise un fichier journal et que le constructeur sert à écrire des entrées dans ce fichier.
- Les constructeurs statiques sont également utiles pour la création de classes wrapper pour du code non managé, quand le constructeur peut appeler la méthode `LoadLibrary`.
- Les constructeurs statiques sont également pratiques pour appliquer les contrôles d'exécution sur le paramètre de type qui ne peuvent pas être effectués au moment de la compilation par le biais de contraintes de paramètre.

Exemple

Dans cet exemple, la classe `Bus` possède un constructeur statique. Quand la première instance de `Bus` est créée (`bus1`), le constructeur statique est appelé pour initialiser la classe. L'exemple de sortie vérifie que le constructeur statique s'exécute une seule fois, même si deux instances de `Bus` sont créées, et qu'il s'exécute avant le constructeur d'instance.

C#

```
public class Bus
{
    // Static variable used by all Bus instances.
    // Represents the time the first bus of the day starts its route.
    protected static readonly DateTime globalStartTime;

    // Property for the number of each bus.
    protected int RouteNumber { get; set; }

    // Static constructor to initialize the static variable.
    // It is invoked before the first instance constructor is run.
    static Bus()
    {
        globalStartTime = DateTime.Now;
```

```
        // The following statement produces the first line of output,
        // and the line occurs only once.
        Console.WriteLine("Static constructor sets global start time to
{0}",
            globalStartTime.ToLongTimeString());
    }

    // Instance constructor.
    public Bus(int routeNum)
    {
        RouteNumber = routeNum;
        Console.WriteLine("Bus #{0} is created.", RouteNumber);
    }

    // Instance method.
    public void Drive()
    {
        TimeSpan elapsedTime = DateTime.Now - globalStartTime;

        // For demonstration purposes we treat milliseconds as minutes to
simulate
        // actual bus times. Do not do this in your actual bus schedule
program!
        Console.WriteLine("{0} is starting its route {1:N2} minutes after
global start time {2}.",
                          this.RouteNumber,
                          elapsedTime.Milliseconds,
                          globalStartTime.ToShortTimeString());
    }
}

class TestBus
{
    static void Main()
    {
        // The creation of this instance activates the static constructor.
        Bus bus1 = new Bus(71);

        // Create a second bus.
        Bus bus2 = new Bus(72);

        // Send bus1 on its way.
        bus1.Drive();

        // Wait for bus2 to warm up.
        System.Threading.Thread.Sleep(25);

        // Send bus2 on its way.
        bus2.Drive();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
```

```
}
/* Sample output:
    Static constructor sets global start time to 3:57:08 PM.
    Bus #71 is created.
    Bus #72 is created.
    71 is starting its route 6.00 minutes after global start time 3:57 PM.
    72 is starting its route 31.00 minutes after global start time 3:57 PM.
*/
```

spécification du langage C#

Pour plus d'informations, voir la section Constructeurs statiques de la spécification du langage C#.

Voir aussi

- Guide de programmation C#
- Système de types C#
- Constructeurs
- Classes statiques et membres de classe statique
- Finaliseurs
- Instructions de conception des constructeurs
- Avertissement de sécurité - CA2121 : Les constructeurs statiques doivent être privés
- Initialiseurs de module

Guide pratique pour écrire un constructeur de copie (Guide de programmation C#)

Article • 01/06/2023

Les enregistrements C# fournissent un constructeur de copie pour les objets, mais pour les classes, vous devez en écrire un vous-même.

> **ⓘ Important**
>
> L'écriture de constructeurs de copie qui fonctionnent pour tous les types dérivés d'une hiérarchie de classes peut être difficile. Si votre classe n'est pas `sealed`, vous devez fortement envisager de créer une hiérarchie de types `record class` pour utiliser le constructeur de copie synthétisé par le compilateur.

Exemple

Dans l'exemple suivant, la `Person` classe définit un constructeur de copie qui prend comme argument une instance de `Person`. Les valeurs des propriétés de l'argument sont assignées aux propriétés de la nouvelle instance de `Person`. Le code contient un autre constructeur de copie qui envoie les propriétés `Name` et `Age` de l'instance que vous voulez copier au constructeur d'instance de la classe. La classe `Person` est `sealed`, de sorte qu'aucun type dérivé ne peut être déclaré qui pourrait introduire des erreurs en copiant uniquement la classe de base.

C#

```
public sealed class Person
{
    // Copy constructor.
    public Person(Person previousPerson)
    {
        Name = previousPerson.Name;
        Age = previousPerson.Age;
    }

    //// Alternate copy constructor calls the instance constructor.
    //public Person(Person previousPerson)
    //    : this(previousPerson.Name, previousPerson.Age)
    //{
    //}
```

```
    // Instance constructor.
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    public int Age { get; set; }

    public string Name { get; set; }

    public string Details()
    {
        return Name + " is " + Age.ToString();
    }
}

class TestPerson
{
    static void Main()
    {
        // Create a Person object by using the instance constructor.
        Person person1 = new Person("George", 40);

        // Create another Person object, copying person1.
        Person person2 = new Person(person1);

        // Change each person's age.
        person1.Age = 39;
        person2.Age = 41;

        // Change person2's name.
        person2.Name = "Charles";

        // Show details to verify that the name and age fields are distinct.
        Console.WriteLine(person1.Details());
        Console.WriteLine(person2.Details());

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:
// George is 39
// Charles is 41
```

Voir aussi

- ICloneable
- Enregistrements

- Guide de programmation C#
- Système de types C#
- Constructeurs
- Finaliseurs

Finaliseurs (Guide de programmation C#)

Article • 21/03/2023

Les finaliseurs (historiquement appelés **destructeurs**) sont utilisés pour effectuer le nettoyage final nécessaire lorsqu'une instance de classe est collectée par le récupérateur de mémoire. Dans la plupart des cas, vous pouvez éviter d'écrire un finaliseur à l'aide des classes System.Runtime.InteropServices.SafeHandle ou dérivées pour envelopper tout descripteur non géré.

Remarques

- Les finaliseurs ne peuvent pas être définis dans des structs. Ils sont utilisés uniquement avec les classes.
- Une classe ne peut avoir qu'un seul finaliseur.
- Les finaliseurs ne peuvent pas être hérités ou surchargés.
- Les finaliseurs ne peuvent pas être appelés. Ils sont appelés automatiquement.
- Un finaliseur ne prend pas de modificateur et n'a pas de paramètre.

Par exemple, voici une déclaration de finaliseur pour la classe `Car`.

C#

```
class Car
{
    ~Car()  // finalizer
    {
        // cleanup statements...
    }
}
```

Un finaliseur peut aussi être implémenté en tant que définition de corps d'expression, comme l'illustre l'exemple suivant.

C#

```
public class Destroyer
{
   public override string ToString() => GetType().Name;

   ~Destroyer() => Console.WriteLine($"The {ToString()} finalizer is
executing.");
}
```

Le finaliseur appelle implicitement Finalize sur la classe de base de l'objet. Ainsi, un appel à un finaliseur est traduit implicitement en ce code :

C#

```
protected override void Finalize()
{
    try
    {
        // Cleanup statements...
    }
    finally
    {
        base.Finalize();
    }
}
```

Cette conception signifie que la méthode `Finalize` est appelée de manière récursive pour toutes les instances de la chaîne d'héritage, de la plus dérivée à la moins dérivée.

> ⓘ Notes
>
> Les finaliseurs vides ne doivent pas être utilisés. Quand une classe contient un finaliseur, une entrée est créée dans la file d'attente `Finalize`. Cette file d'attente est traitée par le récupérateur de mémoire. Lorsque le GC traite la file d'attente, il appelle chaque finaliseur. Les finaliseurs inutiles, y compris les finaliseurs vides, les finaliseurs qui appellent uniquement le finaliseur de classe de base ou les finaliseurs qui appellent uniquement des méthodes émises sous conditions, provoquent une perte inutile de performances.

Le programmeur n'a aucun contrôle sur le moment où le finaliseur est appelé ; le récupérateur de mémoire décide quand l'appeler. Le récupérateur de mémoire recherche les objets qui ne sont plus utilisés par l'application. S'il considère qu'un objet peut être finalisé, il appelle le finaliseur (s'il y en a un) et libère la mémoire utilisée pour stocker l'objet. Il est possible de forcer le nettoyage de mémoire en appelant Collect, mais la plupart du temps, cet appel doit être évité, car il peut créer des problèmes de performances.

> ⓘ Notes
>
> L'exécution ou non des finaliseurs dans le cadre de l'arrêt de l'application est spécifique à chaque **implémentation de .NET**. Lorsqu'une application se termine,

.NET Framework effectue tous les efforts raisonnables pour appeler les finaliseurs des objets qui n'ont pas encore été collectés par la mémoire, à moins que ce nettoyage n'ait été supprimé (par un appel à la méthode `GC.SuppressFinalize` de bibliothèque, par exemple). .NET 5 (y compris .NET Core) et les versions ultérieures n'appellent pas de finaliseurs dans le cadre de l'arrêt de l'application. Pour plus d'informations, consultez le problème GitHub **dotnet/csharpstandard #291**.

Si vous devez effectuer un nettoyage fiable lors de la sortie d'une application, inscrivez un gestionnaire pour l'événement System.AppDomain.ProcessExit. Ce gestionnaire garantit que IDisposable.Dispose() (ou, IAsyncDisposable.DisposeAsync()) a été appelé pour tous les objets qui nécessitent un nettoyage avant la sortie de l'application. Étant donné que vous ne pouvez pas appeler *Finalize* directement et que vous ne pouvez pas garantir que le récupérateur de mémoire appelle tous les finaliseurs avant la sortie, vous devez utiliser `Dispose` ou `DisposeAsync` pour vous assurer que les ressources sont libérées.

Utiliser des finaliseurs pour libérer des ressources

En règle générale, C# ne nécessite pas autant de gestion de la mémoire de la part du développeur que les langages qui ne ciblent pas un runtime avec nettoyage de la mémoire. Cela est dû au fait que le récupérateur de mémoire .NET gère implicitement l'allocation et la libération de mémoire pour vos objets. Toutefois, quand votre application encapsule des ressources non gérées, telles que des fenêtres, des fichiers et des connexions réseau, vous devez utiliser des finaliseurs pour libérer ces ressources. Quand l'objet peut être finalisé, le récupérateur de mémoire exécute la méthode `Finalize` de l'objet.

Libération explicite de ressources

Si votre application utilise une ressource externe coûteuse, nous vous recommandons également de proposer un moyen de libérer explicitement la ressource avant que le récupérateur de mémoire ne libère l'objet. Pour libérer la ressource, implémentez une méthode `Dispose` à partir de l'interface IDisposable qui effectue le nettoyage nécessaire pour l'objet. Cela peut améliorer considérablement les performances de l'application. Même avec ce contrôle explicite sur les ressources, le finaliseur devient une protection pour nettoyer les ressources si l'appel à la méthode `Dispose` échoue.

Pour plus d'informations sur le nettoyage des ressources, consultez les articles suivants :

- Nettoyage de ressources non managées
- Implémentation d'une méthode Dispose
- Implémentation d'une méthode DisposeAsync
- using, instruction

Exemple

L'exemple suivant crée trois classes qui forment une chaîne d'héritage. La classe `First` est la classe de base, `Second` est dérivée de `First`, et `Third` est dérivée de `Second`. Toutes trois ont des finaliseurs. Dans `Main`, une instance de la classe la plus dérivée est créée. La sortie de ce code dépend de l'implémentation de .NET que l'application cible :

- .NET Framework : la sortie indique que les finaliseurs pour les trois classes sont appelés automatiquement lorsque l'application se termine, dans l'ordre du plus dérivé au moins dérivé.
- .NET 5 (y compris .NET Core) ou une version ultérieure : il n'existe aucune sortie, car cette implémentation de .NET n'appelle pas de finaliseurs lorsque l'application se termine.

C#

```
class First
{
    ~First()
    {
        System.Diagnostics.Trace.WriteLine("First's finalizer is called.");
    }
}

class Second : First
{
    ~Second()
    {
        System.Diagnostics.Trace.WriteLine("Second's finalizer is called.");
    }
}

class Third : Second
{
    ~Third()
    {
        System.Diagnostics.Trace.WriteLine("Third's finalizer is called.");
    }
}

/*
Test with code like the following:
    Third t = new Third();
```

```
    t = null;

When objects are finalized, the output would be:
Third's finalizer is called.
Second's finalizer is called.
First's finalizer is called.
*/
```

spécification du langage C#

Pour plus d'informations, consultez la section Finaliseurs de la Spécification du langage C#.

Voir aussi

- IDisposable
- Guide de programmation C#
- Constructeurs
- Garbage collection

Initialiseurs d’objet et de collection (Guide de programmation C#)

Article • 18/07/2023

C# vous permet d’instancier un objet ou une collection et d’effectuer des affectations de membres dans une seule instruction.

Initialiseurs d’objet

Les initialiseurs d'objet vous permettent d'affecter des valeurs aux champs ou propriétés accessibles d'un objet, au moment de sa création, sans devoir appeler un constructeur suivi de ses lignes d'instructions d'assignation. La syntaxe de l'initialiseur de l'objet vous permet de spécifier les arguments d'un constructeur ou de les omettre (et la syntaxe de parenthèses). L'exemple suivant montre comment utiliser l'initialiseur de l'objet de type nommé, `Cat`, et comment appeler le constructeur sans paramètre. Notez l'utilisation de propriétés implémentées automatiquement dans la classe `Cat`. Pour plus d’informations, consultez Propriétés implémentées automatiquement.

C#

```
public class Cat
{
    // Auto-implemented properties.
    public int Age { get; set; }
    public string? Name { get; set; }

    public Cat()
    {
    }

    public Cat(string name)
    {
        this.Name = name;
    }
}
```

C#

```
Cat cat = new Cat { Age = 10, Name = "Fluffy" };
Cat sameCat = new Cat("Fluffy"){ Age = 10 };
```

La syntaxe des initialiseurs d’objet vous permet de créer une instance qui assigne l’objet nouvellement créé, avec ses propriétés, à la variable dans l’assignation.

Les initialiseurs d'objet peuvent, en plus d'affecter des champs et des propriétés, définir des indexeurs. Prenez l'exemple de cette classe `Matrix` de base :

C#

```
public class Matrix
{
    private double[,] storage = new double[3, 3];

    public double this[int row, int column]
    {
        // The embedded array will throw out of range exceptions as
appropriate.
        get { return storage[row, column]; }
        set { storage[row, column] = value; }
    }
}
```

Vous pouvez initialiser la matrice identity avec le code suivant :

C#

```
var identity = new Matrix
{
    [0, 0] = 1.0,
    [0, 1] = 0.0,
    [0, 2] = 0.0,

    [1, 0] = 0.0,
    [1, 1] = 1.0,
    [1, 2] = 0.0,

    [2, 0] = 0.0,
    [2, 1] = 0.0,
    [2, 2] = 1.0,
};
```

Tout indexeur accessible qui contient un setter accessible peut être utilisé comme l'une des expressions d'un initialiseur d'objet, indépendamment du nombre ou des types d'arguments. Les arguments d'index forment le côté gauche de l'affectation, et la valeur le côté droit de l'expression. Par exemple, les éléments suivants sont tous valides si `IndexersExample` a les indexeurs appropriés :

C#

```
var thing = new IndexersExample
{
    name = "object one",
    [1] = '1',
```

```
    [2] = '4',
    [3] = '9',
    Size = Math.PI,
    ['C',4] = "Middle C"
}
```

Pour le code précédent à compiler, le type `IndexersExample` doit avoir les membres suivants :

C#

```
public string name;
public double Size { set { ... }; }
public char this[int i] { set { ... }; }
public string this[char c, int i] {  set { ... }; }
```

Initialiseurs d'objet avec des types anonymes

Les initialiseurs d'objet peuvent être utilisés dans tous les contextes, mais ils sont particulièrement utiles dans les expressions de requête LINQ. Les expressions de requête utilisent souvent des types anonymes, qui peuvent uniquement être initialisés à l'aide d'un initialiseur d'objet, comme indiqué dans la déclaration suivante.

C#

```
var pet = new { Age = 10, Name = "Fluffy" };
```

Les types anonymes permettent à la clause `select` d'une expression de requête lINQ de transformer les objets de la séquence d'origine en objets dont la valeur et la forme peuvent différer de l'original. Cela s'avère utile si vous souhaitez stocker uniquement une partie des informations de chaque objet d'une séquence. Dans l'exemple suivant, supposons qu'un objet de produit (`p`) contient de nombreux champs et méthodes et que vous souhaitez uniquement créer une séquence d'objets qui contiennent le nom de produit et le prix unitaire.

C#

```
var productInfos =
    from p in products
    select new { p.ProductName, p.UnitPrice };
```

Lorsque cette requête est exécutée, la variable `productInfos` contient une séquence d'objets qui sont accessibles dans une instruction `foreach` comme indiqué dans cet

exemple :

```
C#

foreach(var p in productInfos){...}
```

Chaque objet dans le nouveau type anonyme a deux propriétés publiques qui reçoivent les mêmes noms que les propriétés ou champs de l'objet d'origine. Vous pouvez également renommer un champ lorsque vous créez un type anonyme. L'exemple suivant renomme le champ `UnitPrice` en `Price`.

```
C#

select new {p.ProductName, Price = p.UnitPrice};
```

Initialiseurs d'objet avec le modificateur `required`

Vous utilisez le mot clé `required` pour forcer les appelants à définir la valeur d'une propriété ou d'un champ à l'aide d'un initialiseur d'objet. Les propriétés requises n'ont pas besoin d'être définies en tant que paramètres de constructeur. Le compilateur garantit que tous les appelants initialisent ces valeurs.

```
C#

public class Pet
{
    public required int Age;
    public string Name;
}

// `Age` field is necessary to be initialized.
// You don't need to initialize `Name` property
var pet = new Pet() { Age = 10};

// Compiler error:
// Error CS9035 Required member 'Pet.Age' must be set in the object
initializer or attribute constructor.
// var pet = new Pet();
```

C'est une pratique courante que de garantir que votre objet est correctement initialisé, en particulier lorsque vous avez plusieurs champs ou propriétés à gérer et que vous ne souhaitez pas tous les inclure dans le constructeur.

Initialiseurs d'objet avec l'accesseur `init`

Assurez-vous que personne ne modifie l'objet conçu à l'aide d'un accesseur `init`. Il permet de restreindre le paramètre de la valeur de la propriété.

C#

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; init; }
}

// Last name property is necessary to be initialized.
// The `FirstName` property can be modified after initialization.
var pet = new Person() { FirstName = "Joe", LastName = "Doe"};

// You can assign the FirstName property to a different value.
pet.FirstName = "Jane";

// Compiler error:
// Error CS8852  Init - only property or indexer 'Person.LastName' can only
be assigned in an object initializer,
//               or on 'this' or 'base' in an instance constructor or an
'init' accessor.
// pet.LastName = "Kowalski";
```

Les propriétés d'initialisation uniquement requises prennent en charge les structures immuables tout en autorisant la syntaxe naturelle pour les utilisateurs du type.

Initialiseurs de collection

Les initialiseurs de collection vous permettent de spécifier un ou plusieurs initialiseurs d'élément quand vous initialisez un type de collection qui implémente IEnumerable et qui utilise `Add` comme méthode d'instance ou méthode d'extension avec la signature appropriée. Les initialiseurs d'élément peuvent être une valeur simple, une expression ou un initialiseur d'objet. En utilisant un initialiseur de collection, il n'est pas nécessaire de spécifier plusieurs appels ; le compilateur les ajoute automatiquement.

L'exemple suivant montre deux initialiseurs de collection simples :

C#

```
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
List<int> digits2 = new List<int> { 0 + 1, 12 % 3, MakeInt() };
```

L'initialiseur de collection suivant utilise des initialiseurs d'objets pour initialiser les objets de la classe `Cat` définis dans un exemple précédent. Notez que les initialiseurs d'objets individuels sont placés entre accolades et séparés par une virgule.

C#

```
List<Cat> cats = new List<Cat>
{
    new Cat{ Name = "Sylvester", Age=8 },
    new Cat{ Name = "Whiskers", Age=2 },
    new Cat{ Name = "Sasha", Age=14 }
};
```

Vous pouvez spécifier Null comme élément dans un initialiseur de collection si la méthode `Add` de la collection l'autorise.

C#

```
List<Cat?> moreCats = new List<Cat?>
{
    new Cat{ Name = "Furrytail", Age=5 },
    new Cat{ Name = "Peaches", Age=4 },
    null
};
```

Vous pouvez spécifier des éléments indexés si la collection prend en charge l'indexation en lecture/écriture.

C#

```
var numbers = new Dictionary<int, string>
{
    [7] = "seven",
    [9] = "nine",
    [13] = "thirteen"
};
```

L'exemple précédent génère du code qui appelle Item[TKey] pour définir les valeurs. Vous pouvez également initialiser des dictionnaires et d'autres conteneurs associatifs à l'aide de la syntaxe suivante. Notez que la syntaxe de l'indexeur, avec des parenthèses et une affectation, est remplacée par un objet avec plusieurs valeurs :

C#

```
var moreNumbers = new Dictionary<int, string>
{
    {19, "nineteen" },
```

```
    {23, "twenty-three" },
    {42, "forty-two" }
};
```

Cet exemple d'initialiseur appelle Add(TKey, TValue) pour ajouter les trois éléments dans le dictionnaire. Ces deux manières d'initialiser les collections associatives ont un comportement légèrement différent en raison des appels de méthode générés par le compilateur. Les deux variantes fonctionnent avec la classe `Dictionary`. D'autres types peuvent uniquement prendre en charge l'une ou l'autre en fonction de leur API publique.

Initialiseurs d'objets avec initialisation de propriété en lecture seule de collection

Certaines classes peuvent avoir des propriétés de collection où la propriété est en lecture seule, comme la propriété `Cats` de `CatOwner` dans le cas suivant :

C#

```
public class CatOwner
{
    public IList<Cat> Cats { get; } = new List<Cat>();
}
```

Vous ne pourrez pas utiliser la syntaxe d'initialiseur de collection décrite jusqu'à présent, car une nouvelle liste ne peut pas être attribuée à la propriété :

C#

```
CatOwner owner = new CatOwner
{
    Cats = new List<Cat>
    {
        new Cat{ Name = "Sylvester", Age=8 },
        new Cat{ Name = "Whiskers", Age=2 },
        new Cat{ Name = "Sasha", Age=14 }
    }
};
```

Toutefois, de nouvelles entrées peuvent être ajoutées à `Cats` l'aide de la syntaxe d'initialisation en omettant la création de la liste (`new List<Cat>`), comme indiqué ci-dessous :

C#

```
CatOwner owner = new CatOwner
{
    Cats =
    {
        new Cat{ Name = "Sylvester", Age=8 },
        new Cat{ Name = "Whiskers", Age=2 },
        new Cat{ Name = "Sasha", Age=14 }
    }
};
```

L'ensemble d'entrées à ajouter apparaît simplement entouré d'accolades. Ce qui précède est identique à l'écriture :

C#

```
CatOwner owner = new CatOwner();
owner.Cats.Add(new Cat{ Name = "Sylvester", Age=8 });
owner.Cats.Add(new Cat{ Name = "Whiskers", Age=2 });
owner.Cats.Add(new Cat{ Name = "Sasha", Age=14 });
```

Exemples

L'exemple suivant combine les concepts d'initialiseurs d'objet et de collection.

C#

```
public class InitializationSample
{
    public class Cat
    {
        // Auto-implemented properties.
        public int Age { get; set; }
        public string? Name { get; set; }

        public Cat() { }

        public Cat(string name)
        {
            Name = name;
        }
    }

    public static void Main()
    {
        Cat cat = new Cat { Age = 10, Name = "Fluffy" };
        Cat sameCat = new Cat("Fluffy"){ Age = 10 };

        List<Cat> cats = new List<Cat>
        {
```

```
            new Cat { Name = "Sylvester", Age = 8 },
            new Cat { Name = "Whiskers", Age = 2 },
            new Cat { Name = "Sasha", Age = 14 }
        };

        List<Cat?> moreCats = new List<Cat?>
        {
            new Cat { Name = "Furrytail", Age = 5 },
            new Cat { Name = "Peaches", Age = 4 },
            null
        };

        // Display results.
        System.Console.WriteLine(cat.Name);

        foreach (Cat c in cats)
            System.Console.WriteLine(c.Name);

        foreach (Cat? c in moreCats)
            if (c != null)
                System.Console.WriteLine(c.Name);
            else
                System.Console.WriteLine("List element has null value.");
    }
    // Output:
    //Fluffy
    //Sylvester
    //Whiskers
    //Sasha
    //Furrytail
    //Peaches
    //List element has null value.
}
```

L'exemple suivant montre un objet qui implémente IEnumerable et qui contient une méthode `Add` avec plusieurs paramètres. Il utilise un initialiseur de collection avec plusieurs éléments dans la liste correspondant à la signature de la méthode `Add`.

C#

```
    public class FullExample
    {
        class FormattedAddresses : IEnumerable<string>
        {
            private List<string> internalList = new List<string>();
            public IEnumerator<string> GetEnumerator() =>
internalList.GetEnumerator();

            System.Collections.IEnumerator
System.Collections.IEnumerable.GetEnumerator() =>
internalList.GetEnumerator();
```

```
            public void Add(string firstname, string lastname,
                string street, string city,
                string state, string zipcode) => internalList.Add(
                $@"{firstname} {lastname}
{street}
{city}, {state} {zipcode}"
                );
        }

        public static void Main()
        {
            FormattedAddresses addresses = new FormattedAddresses()
            {
                {"John", "Doe", "123 Street", "Topeka", "KS", "00000" },
                {"Jane", "Smith", "456 Street", "Topeka", "KS", "00000" }
            };

            Console.WriteLine("Address Entries:");

            foreach (string addressEntry in addresses)
            {
                Console.WriteLine("\r\n" + addressEntry);
            }
        }

        /*
         * Prints:

            Address Entries:

            John Doe
            123 Street
            Topeka, KS 00000

            Jane Smith
            456 Street
            Topeka, KS 00000
         */
    }
```

Les méthodes `Add` peuvent utiliser le mot clé `params` pour sélectionner un nombre variable d'arguments, comme indiqué dans l'exemple suivant. Cet exemple montre également l'implémentation personnalisée d'un indexeur pour initialiser une collection à l'aide d'index.

C#

```
public class DictionaryExample
{
    class RudimentaryMultiValuedDictionary<TKey, TValue> :
IEnumerable<KeyValuePair<TKey, List<TValue>>> where TKey : notnull
    {
```

```
        private Dictionary<TKey, List<TValue>> internalDictionary = new
Dictionary<TKey, List<TValue>>();

        public IEnumerator<KeyValuePair<TKey, List<TValue>>> GetEnumerator()
=> internalDictionary.GetEnumerator();

        System.Collections.IEnumerator
System.Collections.IEnumerable.GetEnumerator() =>
internalDictionary.GetEnumerator();

        public List<TValue> this[TKey key]
        {
            get => internalDictionary[key];
            set => Add(key, value);
        }

        public void Add(TKey key, params TValue[] values) => Add(key,
(IEnumerable<TValue>)values);

        public void Add(TKey key, IEnumerable<TValue> values)
        {
            if (!internalDictionary.TryGetValue(key, out List<TValue>?
storedValues))
                internalDictionary.Add(key, storedValues = new List<TValue>
());

            storedValues.AddRange(values);
        }
    }

    public static void Main()
    {
        RudimentaryMultiValuedDictionary<string, string>
rudimentaryMultiValuedDictionary1
            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                {"Group1", "Bob", "John", "Mary" },
                {"Group2", "Eric", "Emily", "Debbie", "Jesse" }
            };
        RudimentaryMultiValuedDictionary<string, string>
rudimentaryMultiValuedDictionary2
            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                ["Group1"] = new List<string>() { "Bob", "John", "Mary" },
                ["Group2"] = new List<string>() { "Eric", "Emily", "Debbie",
"Jesse" }
            };
        RudimentaryMultiValuedDictionary<string, string>
rudimentaryMultiValuedDictionary3
            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                {"Group1", new string []{ "Bob", "John", "Mary" } },
                { "Group2", new string[]{ "Eric", "Emily", "Debbie", "Jesse"
} }
            };
```

```
        Console.WriteLine("Using first multi-valued dictionary created with
a collection initializer:");

        foreach (KeyValuePair<string, List<string>> group in
rudimentaryMultiValuedDictionary1)
        {
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");

            foreach (string member in group.Value)
            {
                Console.WriteLine(member);
            }
        }

        Console.WriteLine("\r\nUsing second multi-valued dictionary created
with a collection initializer using indexing:");

        foreach (KeyValuePair<string, List<string>> group in
rudimentaryMultiValuedDictionary2)
        {
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");

            foreach (string member in group.Value)
            {
                Console.WriteLine(member);
            }
        }
        Console.WriteLine("\r\nUsing third multi-valued dictionary created
with a collection initializer using indexing:");

        foreach (KeyValuePair<string, List<string>> group in
rudimentaryMultiValuedDictionary3)
        {
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");

            foreach (string member in group.Value)
            {
                Console.WriteLine(member);
            }
        }
    }

    /*
     * Prints:

        Using first multi-valued dictionary created with a collection
initializer:

        Members of group Group1:
        Bob
        John
        Mary

        Members of group Group2:
```

```
        Eric
        Emily
        Debbie
        Jesse

        Using second multi-valued dictionary created with a collection
initializer using indexing:

        Members of group Group1:
        Bob
        John
        Mary

        Members of group Group2:
        Eric
        Emily
        Debbie
        Jesse

        Using third multi-valued dictionary created with a collection
initializer using indexing:

        Members of group Group1:
        Bob
        John
        Mary

        Members of group Group2:
        Eric
        Emily
        Debbie
        Jesse
    */
}
```

Voir aussi

- Utiliser des initialiseurs d'objet (règle de style IDE0017)
- Utiliser des initialiseurs de collection (règle de style IDE0028)
- Guide de programmation C#
- LINQ en C#
- Types anonymes

Guide pratique pour initialiser des objets à l'aide d'un initialiseur d'objet (Guide de programmation C#)

Article • 07/04/2023

Vous pouvez utiliser des initialiseurs d'objets pour initialiser des objets de type de façon déclarative sans appeler explicitement un constructeur pour le type.

Les exemples suivants montrent comment utiliser des initialiseurs d'objets avec des objets nommés. Le compilateur traite les initialiseurs d'objets en accédant d'abord au constructeur d'instance sans paramètre, puis en traitant les initialisations des membres. Ainsi, si le constructeur sans paramètre est déclaré comme `private` dans la classe, les initialiseurs d'objets qui nécessitent un accès public échoueront.

Vous devez utiliser un initialiseur d'objet si vous définissez un type anonyme. Pour plus d'informations, consultez Guide pratique pour renvoyer des sous-ensembles de propriétés d'éléments dans une requête.

Exemple

L'exemple suivant montre comment initialiser un nouveau type `StudentName` à l'aide d'initialiseurs d'objets. Cet exemple définit des propriétés dans le type `StudentName` :

C#

```
public class HowToObjectInitializers
{
    public static void Main()
    {
        // Declare a StudentName by using the constructor that has two
parameters.
        StudentName student1 = new StudentName("Craig", "Playstead");

        // Make the same declaration by using an object initializer and
sending
        // arguments for the first and last names. The parameterless
constructor is
        // invoked in processing this declaration, not the constructor that
has
        // two parameters.
        StudentName student2 = new StudentName
        {
            FirstName = "Craig",
            LastName = "Playstead"
```

```
        };

        // Declare a StudentName by using an object initializer and sending
        // an argument for only the ID property. No corresponding
constructor is
        // necessary. Only the parameterless constructor is used to process
object
        // initializers.
        StudentName student3 = new StudentName
        {
            ID = 183
        };

        // Declare a StudentName by using an object initializer and sending
        // arguments for all three properties. No corresponding constructor
is
        // defined in the class.
        StudentName student4 = new StudentName
        {
            FirstName = "Craig",
            LastName = "Playstead",
            ID = 116
        };

        Console.WriteLine(student1.ToString());
        Console.WriteLine(student2.ToString());
        Console.WriteLine(student3.ToString());
        Console.WriteLine(student4.ToString());
    }
    // Output:
    // Craig  0
    // Craig  0
    //   183
    // Craig  116

    public class StudentName
    {
        // This constructor has no parameters. The parameterless constructor
        // is invoked in the processing of object initializers.
        // You can test this by changing the access modifier from public to
        // private. The declarations in Main that use object initializers
will
        // fail.
        public StudentName() { }

        // The following constructor has parameters for two of the three
        // properties.
        public StudentName(string first, string last)
        {
            FirstName = first;
            LastName = last;
        }

        // Properties.
        public string? FirstName { get; set; }
```

```
        public string? LastName { get; set; }
        public int ID { get; set; }

        public override string ToString() => FirstName + "   " + ID;
    }
}
```

Des initialiseurs d'objet peuvent être utilisés pour définir des indexeurs dans un objet. L'exemple suivant définit une classe `BaseballTeam` qui utilise un indexeur pour obtenir et définir des joueurs à des positions différentes. L'initialiseur peut affecter des joueurs en fonction de l'abréviation de la position ou du chiffre utilisé pour désigner la position sur les feuilles de match :

C#

```
public class HowToIndexInitializer
{
    public class BaseballTeam
    {
        private string[] players = new string[9];
        private readonly List<string> positionAbbreviations = new
List<string>
        {
            "P", "C", "1B", "2B", "3B", "SS", "LF", "CF", "RF"
        };

        public string this[int position]
        {
            // Baseball positions are 1 - 9.
            get { return players[position-1]; }
            set { players[position-1] = value; }
        }
        public string this[string position]
        {
            get { return players[positionAbbreviations.IndexOf(position)]; }
            set { players[positionAbbreviations.IndexOf(position)] = value;
}
        }
    }

    public static void Main()
    {
        var team = new BaseballTeam
        {
            ["RF"] = "Mookie Betts",
            [4] = "Jose Altuve",
            ["CF"] = "Mike Trout"
        };

        Console.WriteLine(team["2B"]);
    }
}
```

Voir aussi

- Guide de programmation C#
- Initialiseurs d’objets et de collections

Guide pratique pour initialiser un dictionnaire avec un initialiseur de collection (Guide de programmation C#)

Article • 29/06/2023

Un Dictionary<TKey,TValue> contient une collection de paires clé/valeur. Sa méthode Add prend deux paramètres, un pour la clé et un pour la valeur. Une façon d'initialiser un Dictionary<TKey,TValue> ou toute collection dont la méthode `Add` prend plusieurs paramètres consiste à mettre chaque jeu de paramètres entre accolades, comme illustré dans l'exemple suivant. Une autre option consiste à utiliser un initialiseur d'index, également illustré dans l'exemple suivant.

> ⓘ **Notes**
>
> Il existe une différence majeure entre ces deux méthodes d'initialisation de la collection. Dans le cas de clés dupliquées, par exemple :
>
> C#
>
> ```
> { 111, new StudentName { FirstName="Sachin", LastName="Karnik", ID=211
> } },
> { 111, new StudentName { FirstName="Dina", LastName="Salimzianova",
> ID=317 } },
> ```
>
> La méthode **Add** lève **ArgumentException** : `'An item with the same key has already been added. Key: 111'`, tandis que la deuxième partie de l'exemple, la méthode d'indexation publique en lecture/écriture, remplace silencieusement l'entrée existante par la même clé.

Exemple

Dans l'exemple de code suivant, un Dictionary<TKey,TValue> est initialisé avec des instances de type `StudentName`. La première initialisation utilise la méthode `Add` avec deux arguments. Le compilateur génère un appel à `Add` pour chacune des paires de clés `int` et de valeurs `StudentName`. Le deuxième utilise une méthode d'indexeur publique en lecture/écriture de la classe `Dictionary` :

C#

```
public class HowToDictionaryInitializer
{
    class StudentName
    {
        public string? FirstName { get; set; }
        public string? LastName { get; set; }
        public int ID { get; set; }
    }

    public static void Main()
    {
        var students = new Dictionary<int, StudentName>()
        {
            { 111, new StudentName { FirstName="Sachin", LastName="Karnik",
ID=211 } },
            { 112, new StudentName { FirstName="Dina",
LastName="Salimzianova", ID=317 } },
            { 113, new StudentName { FirstName="Andy", LastName="Ruth",
ID=198 } }
        };

        foreach(var index in Enumerable.Range(111, 3))
        {
            Console.WriteLine($"Student {index} is
{students[index].FirstName} {students[index].LastName}");
        }
        Console.WriteLine();

        var students2 = new Dictionary<int, StudentName>()
        {
            [111] = new StudentName { FirstName="Sachin", LastName="Karnik",
ID=211 },
            [112] = new StudentName { FirstName="Dina",
LastName="Salimzianova", ID=317 } ,
            [113] = new StudentName { FirstName="Andy", LastName="Ruth",
ID=198 }
        };

        foreach (var index in Enumerable.Range(111, 3))
        {
            Console.WriteLine($"Student {index} is
{students2[index].FirstName} {students2[index].LastName}");
        }
    }
}
```

Notez les deux paires d'accolades dans chaque élément de la collection dans la première déclaration. Les accolades les plus intérieures encadrent l'initialiseur d'objet pour `StudentName`, et les accolades les plus extérieures encadrent l'initialiseur pour la paire clé/valeur ajoutée à `students`Dictionary<TKey,TValue>. Pour finir, l'initialiseur de collection entier pour le dictionnaire est placé entre accolades. Dans la deuxième

initialisation, le côté gauche de l'affectation est la clé et le côté droit est la valeur, avec un initialiseur d'objet pour `StudentName`.

Voir aussi

- Guide de programmation C#
- Initialiseurs d'objets et de collections

Types imbriqués (guide de programmation C#)

Article • 07/04/2023

Un type défini dans une classe, un struct ou une interface est appelé « type imbriqué ». Par exemple

```
C#

public class Container
{
    class Nested
    {
        Nested() { }
    }
}
```

Que le type externe soit une classe ou un struct, les types imbriqués sont par défaut privés ; ils sont accessibles uniquement à partir de leur type de conteneur. Dans l'exemple précédent, la classe `Nested` est inaccessible aux types externes.

Vous pouvez aussi spécifier un modificateur d'accès pour définir l'accessibilité d'un type imbriqué, comme ceci :

- Les types imbriqués d'une **classe** peuvent être public, protected, internal, protected internal, private ou private protected.

  Cependant, le fait de définir une classe imbriquée `protected`, `protected internal` ou `private protected` à l'intérieur d'une classe sealed a pour effet de générer l'avertissement de compilateur CS0628, « nouveau membre protected déclaré dans une classe sealed ».

  Sachez également que rendre un type imbriqué visible en externe enfreint la règle de qualité du code CA1034 « Les types imbriqués ne doivent pas être visibles ».

- Les types imbriqués d'un **struct** peuvent être public, internal ou private.

L'exemple suivant fait passer le type de la classe `Nested` à public :

```
C#

public class Container
{
    public class Nested
```

```
    {
        Nested() { }
    }
}
```

Le type imbriqué (ou interne) peut accéder au type conteneur (ou externe). Pour accéder au type conteneur, passez-le en tant qu'argument au constructeur du type imbriqué. Par exemple :

C#

```
public class Container
{
    public class Nested
    {
        private Container? parent;

        public Nested()
        {
        }
        public Nested(Container parent)
        {
            this.parent = parent;
        }
    }
}
```

Un type imbriqué a accès à tous les membres qui sont accessibles à son type conteneur. Il peut accéder aux membres privés et protégés du type conteneur, y compris à tous les membres protégés hérités.

Dans la déclaration précédente, le nom complet de classe `Nested` est `Container.Nested`. Il s'agit du nom utilisé pour créer une instance de la classe imbriquée, comme suit :

C#

```
Container.Nested nest = new Container.Nested();
```

Voir aussi

- Guide de programmation C#
- Système de types C#
- Modificateurs d'accès
- Constructeurs
- Règle CA1034

Classes et méthodes partielles (Guide de programmation C#)

Article • 07/04/2023

Il est possible de fractionner la définition d'une classe, d'un struct, d'une interface ou d'une méthode entre plusieurs fichiers sources. Chaque fichier source contient une section de la définition de méthode ou de type, et toutes les parties sont combinées au moment où l'application est compilée.

Classes partielles

Il peut être utile de fractionner une définition de classe dans les situations suivantes :

- Dans des projets volumineux, le fractionnement d'une classe entre plusieurs fichiers distincts permet à plusieurs programmeurs d'y travailler simultanément.
- Quand vous utilisez une source générée automatiquement, vous pouvez ajouter du code à la classe sans avoir à recréer le fichier source. Visual Studio suit cette approche pour créer des formulaires Windows Forms, du code wrapper de service web, etc. Vous pouvez écrire du code qui utilise ces classes sans avoir à modifier le fichier créé par Visual Studio.
- Lorsque vous utilisez des générateurs de sources pour générer des fonctionnalités supplémentaires dans une classe.

Pour fractionner une définition de classe, utilisez le modificateur de mot clé partial, comme ci-dessous :

C#

```
public partial class Employee
{
    public void DoWork()
    {
    }
}

public partial class Employee
{
    public void GoToLunch()
    {
    }
}
```

Le mot clé `partial` indique que d'autres parties de la classe, du struct ou de l'interface peuvent être définies dans l'espace de noms. Toutes les parties doivent utiliser le mot clé `partial`. Toutes les parties doivent être disponibles à la compilation pour former le type final. Toutes les parties doivent avoir la même accessibilité : `public`, `private`, etc.

Si une partie est déclarée comme abstract, l'ensemble du type est considéré comme abstract. Si une partie est déclarée comme sealed, l'ensemble du type est considéré comme sealed. Si une partie déclare un type de base, l'ensemble du type hérite de cette classe.

Toutes les parties qui spécifient une classe de base doivent être en accord, mais les parties qui omettent une classe de base héritent tout de même du type de base. Les parties peuvent spécifier des interfaces de base différentes. Le type final implémente alors toutes les interfaces indiquées dans toutes les déclarations partielles. Tous les membres de classe, de struct ou d'interface déclarés dans une définition partielle sont disponibles pour toutes les autres parties. Le type final est la combinaison de toutes les parties au moment de la compilation.

> ⓘ **Notes**
>
> Le modificateur `partial` n'est pas disponible sur les déclarations de délégué ou d'énumération.

L'exemple suivant montre que les types imbriqués peuvent être partiels, même si le type dans lequel ils sont imbriqués n'est pas partiel lui-même.

C#

```
class Container
{
    partial class Nested
    {
        void Test() { }
    }

    partial class Nested
    {
        void Test2() { }
    }
}
```

Au moment de la compilation, les attributs de définitions de type partiel sont fusionnés. Observez, par exemple, les déclarations suivantes :

C#

```
[SerializableAttribute]
partial class Moon { }

[ObsoleteAttribute]
partial class Moon { }
```

Elles sont équivalentes aux déclarations suivantes :

C#

```
[SerializableAttribute]
[ObsoleteAttribute]
class Moon { }
```

Les éléments suivants sont fusionnés à partir de toutes les définitions de type partiel :

- Commentaires XML
- interfaces
- attributs de paramètre de type générique
- attributs de classe
- membres

Observez, par exemple, les déclarations suivantes :

C#

```
partial class Earth : Planet, IRotate { }
partial class Earth : IRevolve { }
```

Elles sont équivalentes aux déclarations suivantes :

C#

```
class Earth : Planet, IRotate, IRevolve { }
```

Restrictions

Il y a plusieurs règles à respecter quand vous utilisez des définitions de classe partielle :

- Toutes les définitions de type partiel conçues comme des parties du même type doivent être modifiées avec `partial`. Par exemple, les déclarations de classe suivantes génèrent une erreur :

```C#
public partial class A { }
//public class A { }  // Error, must also be marked partial
```

- Le modificateur `partial` peut uniquement être placé juste avant les mots clés `class`, `struct` ou `interface`.
- Les types partiels imbriqués sont autorisés dans les définitions de type partiel, comme illustré dans l'exemple suivant :

```C#
partial class ClassWithNestedClass
{
    partial class NestedClass { }
}

partial class ClassWithNestedClass
{
    partial class NestedClass { }
}
```

- Toutes les définitions de type partiel conçues comme des parties du même type doivent être définies dans le même assembly et dans le même module (fichier .exe ou .dll). Les définitions partielles ne peuvent pas être fractionnées entre plusieurs modules.
- Le nom de classe et les paramètres de type générique doivent correspondre dans toutes les définitions de type partiel. Les types génériques peuvent être partiels. Chaque déclaration partielle doit utiliser les mêmes noms de paramètres, dans le même ordre.
- Les mots clés suivants sont facultatifs dans une définition de type partiel. S'ils sont utilisés dans une définition de type partiel, ils ne doivent pas être en conflit avec les mots clés spécifiés dans une autre définition partielle pour le même type :
  - public
  - private
  - protected
  - internal
  - abstract
  - sealed
  - classe de base
  - modificateur new (parties imbriquées)
  - contraintes génériques

Pour plus d'informations, consultez Contraintes sur les paramètres de type.

Exemples

Dans l'exemple suivant, les champs et le constructeur de la classe `Coords` sont déclarés dans une définition de classe partielle, et le membre `PrintCoords` est déclaré dans une autre définition de classe partielle.

C#

```
public partial class Coords
{
    private int x;
    private int y;

    public Coords(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

public partial class Coords
{
    public void PrintCoords()
    {
        Console.WriteLine("Coords: {0},{1}", x, y);
    }
}

class TestCoords
{
    static void Main()
    {
        Coords myCoords = new Coords(10, 15);
        myCoords.PrintCoords();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: Coords: 10,15
```

L'exemple suivant montre que vous pouvez également développer des interfaces et des structs partiels.

C#

```
partial interface ITest
{
    void Interface_Test();
```

```
}

partial interface ITest
{
    void Interface_Test2();
}

partial struct S1
{
    void Struct_Test() { }
}

partial struct S1
{
    void Struct_Test2() { }
}
```

Méthodes partielles

Une classe partielle ou un struct partiel peut contenir une méthode partielle. Une partie de la classe contient la signature de la méthode. Une implémentation peut être définie dans la même partie ou dans une autre partie. Si l'implémentation n'est pas fournie, la méthode et tous les appels à cette méthode sont supprimés au moment de la compilation. Une implémentation peut être nécessaire en fonction de la signature de la méthode. Une méthode partielle n'est pas tenue d'avoir une implémentation dans les cas suivants :

- Elle n'a aucun modificateur d'accessibilité (y compris le private par défaut).
- Elle retourne la valeur void.
- Elle n'a pas de paramètres out.
- Elle n'a aucun des modificateurs suivants : virtual, override, sealed, new ou extern.

Les méthodes qui ne sont pas conformes à toutes ces restrictions (par exemple, la méthode `public virtual partial void`) doivent fournir une implémentation. Cette implémentation peut être fournie par un *générateur de sources*.

Les méthodes partielles permettent à l'implémenteur d'une partie d'une classe de déclarer une méthode. L'implémenteur d'une autre partie de la classe peut définir cette méthode. Ceci peut être utile dans les deux cas suivants : pour les modèles qui génèrent du code réutilisable et pour les générateurs de sources.

- **Code du modèle** : le modèle réserve un nom et une signature de méthode afin que le code généré puisse appeler la méthode. Ces méthodes suivent les restrictions qui permettent à un développeur de décider s'il doit implémenter la méthode. Si la méthode n'est pas implémentée, le compilateur supprime la

signature de méthode et tous les appels à la méthode. Les appels à la méthode, y compris tous les résultats retournés par l'évaluation des arguments dans les appels, n'ont aucun effet au moment de l'exécution. C'est pourquoi le code dans la classe partielle peut utiliser librement une méthode partielle, même si l'implémentation n'est pas fournie. Aucune erreur de compilation ou d'exécution n'est générée si la méthode est appelée, mais qu'elle n'est pas finalement implémentée.

- **Générateurs de sources** : les générateurs de sources fournissent une implémentation pour les méthodes. Le développeur humain peut ajouter la déclaration de méthode (souvent avec des attributs lus par le générateur de sources). Le développeur peut écrire du code qui appelle ces méthodes. Le générateur de sources s'exécute au moment de la compilation et fournit l'implémentation. Dans ce scénario, les restrictions relatives aux méthodes partielles qui ne peuvent pas être implémentées ne sont souvent pas respectées.

C#

```
// Definition in file1.cs
partial void OnNameChanged();

// Implementation in file2.cs
partial void OnNameChanged()
{
  // method body
}
```

- Les déclarations de méthode partielle doivent commencer par le mot clé contextuel partial.
- Les signatures de méthode partielle figurant dans les deux parties du type partial doivent correspondre.
- Les méthodes partielles peuvent avoir des modificateurs static et unsafe.
- Les méthodes partielles peuvent être génériques. Les contraintes sont placées sur la déclaration de méthode partielle de définition et peuvent être répétées facultativement sur la déclaration d'implémentation. Les noms de paramètre et de paramètre de type ne doivent pas obligatoirement être identiques dans la déclaration d'implémentation et la déclaration de définition.
- Vous pouvez créer un délégué pour une méthode partielle qui a été définie et implémentée, mais pas pour une méthode partielle qui a uniquement été définie.

Spécification du langage C#

Pour plus d'informations, consultez Types partiels dans la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Guide de programmation C#
- Classes
- Types de structure
- Interfaces
- partial, Type

Comment retourner des sous-ensembles de propriétés d'éléments dans une requête (Guide de programmation C#)

Article • 07/04/2023

Utilisez un type anonyme dans une expression de requête lorsque les deux conditions suivantes s'appliquent :

- Vous souhaitez retourner uniquement certaines propriétés de chaque élément source.
- Vous n'avez pas besoin de stocker les résultats de requête en dehors de la portée de la méthode dans laquelle la requête est exécutée.

Si vous souhaitez uniquement retourner une propriété ou un champ de chaque élément source, vous pouvez utiliser l'opérateur point dans la clause `select`. Par exemple, pour retourner uniquement l'`ID` de chaque `student`, écrivez la clause `select` de la façon suivante :

C#

```
select student.ID;
```

Exemple

L'exemple suivant montre comment utiliser un type anonyme pour retourner uniquement un sous-ensemble des propriétés de chaque élément source qui répond à la condition spécifiée.

C#

```
private static void QueryByScore()
{
    // Create the query. var is required because
    // the query produces a sequence of anonymous types.
    var queryHighScores =
        from student in students
        where student.ExamScores[0] > 95
        select new { student.FirstName, student.LastName };

    // Execute the query.
    foreach (var obj in queryHighScores)
```

```
    {
        // The anonymous type's properties were not named. Therefore
        // they have the same names as the Student properties.
        Console.WriteLine(obj.FirstName + ", " + obj.LastName);
    }
}
/* Output:
Adams, Terry
Fakhouri, Fadi
Garcia, Cesar
Omelchenko, Svetlana
Zabokritski, Eugene
*/
```

Notez que le type anonyme utilise les noms de l'élément source pour ses propriétés si aucun nom n'est spécifié. Pour attribuer de nouveaux noms aux propriétés du type anonyme, écrivez l'instruction `select` de la façon suivante :

C#

```
select new { First = student.FirstName, Last = student.LastName };
```

Si vous effectuez cette opération dans l'exemple précédent, l'instruction `Console.WriteLine` doit également changer :

C#

```
Console.WriteLine(student.First + " " + student.Last);
```

Compilation du code

Pour exécuter ce code, copiez et collez la classe dans une application console C# avec une directive `using` pour System.Linq.

Voir aussi

- Guide de programmation C#
- Types anonymes
- LINQ en C#

Implémentation d'interface explicite (Guide de programmation C#)

Article • 07/04/2023

Si une classe implémente deux interfaces qui contiennent un membre avec la même signature, l'implémentation de ce membre sur la classe a pour conséquence que les deux interfaces utilisent ce membre comme leur implémentation. Dans l'exemple suivant, tous les appels à `Paint` appellent la même méthode. Ce premier exemple définit les types :

C#

```
public interface IControl
{
    void Paint();
}
public interface ISurface
{
    void Paint();
}
public class SampleClass : IControl, ISurface
{
    // Both ISurface.Paint and IControl.Paint call this method.
    public void Paint()
    {
        Console.WriteLine("Paint method in SampleClass");
    }
}
```

L'exemple suivant appelle les méthodes :

C#

```
SampleClass sample = new SampleClass();
IControl control = sample;
ISurface surface = sample;

// The following lines all call the same method.
sample.Paint();
control.Paint();
surface.Paint();

// Output:
// Paint method in SampleClass
// Paint method in SampleClass
// Paint method in SampleClass
```

Vous ne souhaiterez cependant peut-être pas que la même implémentation soit appelée pour les deux interfaces. Pour appeler une implémentation différente en fonction de l'interface utilisée, vous pouvez implémenter explicitement un membre d'interface. Une implémentation d'interface explicite est un membre de classe qui est uniquement appelé via l'interface spécifiée. Nommez le membre de classe en le préfixant avec le nom de l'interface et un point. Par exemple :

C#

```
public class SampleClass : IControl, ISurface
{
    void IControl.Paint()
    {
        System.Console.WriteLine("IControl.Paint");
    }
    void ISurface.Paint()
    {
        System.Console.WriteLine("ISurface.Paint");
    }
}
```

Le membre de classe `IControl.Paint` est disponible uniquement via l'interface `IControl` et `ISurface.Paint` est disponible uniquement via `ISurface`. Les deux implémentations de méthode sont distinctes et aucune d'elles n'est disponible directement sur la classe. Par exemple :

C#

```
SampleClass sample = new SampleClass();
IControl control = sample;
ISurface surface = sample;

// The following lines all call the same method.
//sample.Paint(); // Compiler error.
control.Paint();  // Calls IControl.Paint on SampleClass.
surface.Paint();  // Calls ISurface.Paint on SampleClass.

// Output:
// IControl.Paint
// ISurface.Paint
```

L'implémentation explicite est également utilisée pour résoudre les cas où deux interfaces déclarent chacune des membres différents du même nom, comme une propriété et une méthode. Pour implémenter les deux interfaces, une classe doit utiliser une implémentation explicite pour la propriété `P`, ou la méthode `P`, ou les deux, pour éviter une erreur du compilateur. Par exemple :

C#

```
interface ILeft
{
    int P { get;}
}
interface IRight
{
    int P();
}

class Middle : ILeft, IRight
{
    public int P() { return 0; }
    int ILeft.P { get { return 0; } }
}
```

Une implémentation d'interface explicite n'a pas de modificateur d'accès, car elle n'est pas accessible en tant que membre du type dans lequel elle est définie. Elle n'est accessible qu'en cas d'appel via une instance de l'interface. Si vous spécifiez un modificateur d'accès pour une implémentation d'interface explicite, vous obtenez l'erreur du compilateur CS0106. Pour plus d'informations, consultez interface (Référence C#).

Vous pouvez définir une implémentation pour les membres déclarés dans une interface. Si une classe hérite d'une implémentation de méthode à partir d'une interface, cette méthode n'est accessible qu'à l'aide d'une référence du type d'interface. Le membre hérité n'apparaît pas dans le cadre de l'interface publique. L'exemple suivant définit une implémentation par défaut pour une méthode d'interface :

C#

```
public interface IControl
{
    void Paint() => Console.WriteLine("Default Paint method");
}
public class SampleClass : IControl
{
    // Paint() is inherited from IControl.
}
```

L'exemple suivant appelle l'implémentation par défaut :

C#

```
var sample = new SampleClass();
//sample.Paint();// "Paint" isn't accessible.
```

```
var control = sample as IControl;
control.Paint();
```

Toute classe qui implémente l'interface `IControl` peut remplacer la méthode par défaut `Paint` , en tant que méthode publique ou en tant qu'implémentation d'interface explicite.

Voir aussi

- Guide de programmation C#
- Programmation orientée objet
- Interfaces
- Héritage

Comment : implémenter de manière explicite des membres d'interface (Guide de programmation C#)

Article • 07/04/2023

Cet exemple déclare une interface, `IDimensions` et une classe `Box`, qui implémente de manière explicite les membres d'interface `GetLength` et `GetWidth`. Les membres sont accessibles via l'instance d'interface `dimensions`.

Exemple

C#

```
interface IDimensions
{
    float GetLength();
    float GetWidth();
}

class Box : IDimensions
{
    float lengthInches;
    float widthInches;

    Box(float length, float width)
    {
        lengthInches = length;
        widthInches = width;
    }
    // Explicit interface member implementation:
    float IDimensions.GetLength()
    {
        return lengthInches;
    }
    // Explicit interface member implementation:
    float IDimensions.GetWidth()
    {
        return widthInches;
    }

    static void Main()
    {
        // Declare a class instance box1:
        Box box1 = new Box(30.0f, 20.0f);

        // Declare an interface instance dimensions:
```

```
        IDimensions dimensions = box1;

        // The following commented lines would produce compilation
        // errors because they try to access an explicitly implemented
        // interface member from a class instance:
        //System.Console.WriteLine("Length: {0}", box1.GetLength());
        //System.Console.WriteLine("Width: {0}", box1.GetWidth());

        // Print out the dimensions of the box by calling the methods
        // from an instance of the interface:
        System.Console.WriteLine("Length: {0}", dimensions.GetLength());
        System.Console.WriteLine("Width: {0}", dimensions.GetWidth());
    }
}
/* Output:
    Length: 30
    Width: 20
*/
```

Programmation fiable

- Notez que les lignes suivantes de la méthode `Main` sont commentées car elles produiraient des erreurs de compilation. Un membre d'interface qui est implémenté de façon explicite n'est pas accessible à partir d'une instance de classe :

  C#

  ```
  //System.Console.WriteLine("Length: {0}", box1.GetLength());
  //System.Console.WriteLine("Width: {0}", box1.GetWidth());
  ```

- Notez également que les lignes suivantes de la méthode `Main` impriment correctement les dimensions de la zone, car les méthodes sont appelées à partir d'une instance de l'interface :

  C#

  ```
  System.Console.WriteLine("Length: {0}", dimensions.GetLength());
  System.Console.WriteLine("Width: {0}", dimensions.GetWidth());
  ```

Voir aussi

- Guide de programmation C#
- Programmation orientée objet
- Interfaces

- Comment implémenter explicitement des membres de deux interfaces

Comment implémenter de manière explicite des membres de deux interfaces (Guide de programmation C#)

Article • 07/04/2023

L'implémentation explicite d'une interface permet également au programmeur d'implémenter deux interfaces qui ont les mêmes noms de membres et donnent à chaque membre d'interface une implémentation distincte. L'exemple suivant affiche les dimensions d'une zone dans les unités de mesure à la fois métriques et britanniques. La classe Box implémente deux interfaces, IEnglishDimensions et IMetricDimensions, qui représentent les différents systèmes de mesure. Les deux interfaces ont des noms de membres identiques, Length et Width.

Exemple

C#

```
// Declare the English units interface:
interface IEnglishDimensions
{
    float Length();
    float Width();
}

// Declare the metric units interface:
interface IMetricDimensions
{
    float Length();
    float Width();
}

// Declare the Box class that implements the two interfaces:
// IEnglishDimensions and IMetricDimensions:
class Box : IEnglishDimensions, IMetricDimensions
{
    float lengthInches;
    float widthInches;

    public Box(float lengthInches, float widthInches)
    {
        this.lengthInches = lengthInches;
        this.widthInches = widthInches;
    }

    // Explicitly implement the members of IEnglishDimensions:
```

```
    float IEnglishDimensions.Length() => lengthInches;

    float IEnglishDimensions.Width() => widthInches;

    // Explicitly implement the members of IMetricDimensions:
    float IMetricDimensions.Length() => lengthInches * 2.54f;

    float IMetricDimensions.Width() => widthInches * 2.54f;

    static void Main()
    {
        // Declare a class instance box1:
        Box box1 = new Box(30.0f, 20.0f);

        // Declare an instance of the English units interface:
        IEnglishDimensions eDimensions = box1;

        // Declare an instance of the metric units interface:
        IMetricDimensions mDimensions = box1;

        // Print dimensions in English units:
        System.Console.WriteLine("Length(in): {0}", eDimensions.Length());
        System.Console.WriteLine("Width (in): {0}", eDimensions.Width());

        // Print dimensions in metric units:
        System.Console.WriteLine("Length(cm): {0}", mDimensions.Length());
        System.Console.WriteLine("Width (cm): {0}", mDimensions.Width());
    }
}
/* Output:
    Length(in): 30
    Width (in): 20
    Length(cm): 76.2
    Width (cm): 50.8
*/
```

Programmation fiable

Si vous souhaitez utiliser par défaut les unités de mesure britanniques, implémentez les méthodes Length et Width normalement, puis implémentez explicitement les méthodes Length et Width à partir de l'interface IMetricDimensions :

C#

```
// Normal implementation:
public float Length() => lengthInches;
public float Width() => widthInches;

// Explicit implementation:
float IMetricDimensions.Length() => lengthInches * 2.54f;
float IMetricDimensions.Width() => widthInches * 2.54f;
```

Dans ce cas, vous pouvez accéder aux unités de mesure britanniques à partir de l'instance de classe et accéder aux unités métriques à partir de l'instance d'interface :

C#

```
public static void Test()
{
    Box box1 = new Box(30.0f, 20.0f);
    IMetricDimensions mDimensions = box1;

    System.Console.WriteLine("Length(in): {0}", box1.Length());
    System.Console.WriteLine("Width (in): {0}", box1.Width());
    System.Console.WriteLine("Length(cm): {0}", mDimensions.Length());
    System.Console.WriteLine("Width (cm): {0}", mDimensions.Width());
}
```

Voir aussi

- Guide de programmation C#
- Programmation orientée objet
- Interfaces
- Comment implémenter explicitement des membres d'interface

Délégués (Guide de programmation C#)

Article • 25/02/2023

Un délégué est un type qui représente des références aux méthodes avec une liste de paramètres et un type de retour particuliers. Lorsque vous instanciez un délégué, vous pouvez associer son instance à toute méthode ayant une signature et un type de retour compatibles. Vous pouvez appeler la méthode par le biais l'instance de délégué.

Les délégués sont utilisés pour passer des méthodes comme arguments à d'autres méthodes. Les gestionnaires d'événements sont tout simplement des méthodes appelées par le biais de délégués. Vous créez une méthode personnalisée, et une classe telle qu'un contrôle Windows peut appeler votre méthode lorsqu'un certain événement se produit. L'exemple suivant illustre une déclaration de délégué :

C#

```
public delegate int PerformCalculation(int x, int y);
```

Toute méthode de n'importe quelle classe ou structure accessible qui correspond au type de délégué, peut être assignée au délégué. La méthode peut être une méthode d'instance ou statique. Cette flexibilité signifie que vous pouvez modifier par programme les appels de méthode ou connecter du nouveau code à des classes existantes.

> ⓘ **Notes**
>
> Dans le contexte de surcharge de méthodes, la signature d'une méthode n'inclut pas la valeur de retour. Mais dans le contexte des délégués, la signature inclut la valeur de retour. En d'autres termes, une méthode doit avoir le même type de retour que le délégué.

Cette capacité à faire référence à une méthode en tant que paramètre fait des délégués les instruments idéaux pour définir des méthodes de rappel. Vous pouvez écrire une méthode qui compare deux objets dans votre application. Cette méthode peut être utilisée dans un délégué pour un algorithme de tri. Étant donné que le code de comparaison est distinct de la bibliothèque, la méthode de tri peut être plus générale.

Des pointeurs de fonction ont été ajoutés à C# 9 pour des scénarios similaires, où vous avez besoin de davantage de contrôle sur la convention d'appel. Le code associé à un

délégué est appelé à l'aide d'une méthode virtuelle ajoutée à un type délégué. À l'aide de pointeurs de fonction, vous pouvez spécifier différentes conventions.

Vue d'ensemble des délégués

Les délégués ont les propriétés suivantes :

- Les délégués sont comparables aux pointeurs de fonction C++, sauf que les délégués sont totalement orientés objet, et contrairement aux pointeurs C++ vers les fonctions membres, les délégués encapsulent une instance d'objet et une méthode.
- Les délégués permettent aux méthodes d'être transmises comme des paramètres.
- Les délégués peuvent être utilisés pour définir des méthodes de rappel.
- Les délégués peuvent être chaînés ; par exemple, plusieurs méthodes peuvent être appelées sur un seul événement.
- Les méthodes ne doivent pas nécessairement correspondre exactement au type délégué. Pour plus d'informations, consultez Utilisation de la variance dans les délégués.
- Les expressions lambda sont un moyen plus concis d'écrire des blocs de code inclus. Les expressions lambda (dans certains contextes) sont compilées en types délégués. Pour plus d'informations sur les expressions lambda, voir Expressions lambda.

Dans cette section

- Utilisation de délégués
- Quand utiliser des délégués à la place d'interfaces (Guide de programmation C#)
- Délégués avec méthodes nommées et Méthodes anonymes
- Utilisation de la variance dans les délégués
- Guide pratique pour combiner des délégués (délégués multicast)
- Comment déclarer, instancier et utiliser un délégué

Spécification du langage C#

Pour plus d'informations, consultez Délégués dans la Spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Chapitres proposés

- Delegates, Events, and Lambda Expressions (Délégués, événements et expressions lambda) dans C# 3.0 Cookbook, Third Edition: More than 250 solutions for C# 3.0 programmers
- Délégués et événements dans Apprentissage de C# 3.0 : Maîtriser les fondamentaux de C# 3.0

Voir aussi

- Delegate
- Guide de programmation C#
- Événements

Utilisation de délégués (guide de programmation C#)

Article • 31/07/2023

Un délégué est un type qui encapsule sans risque une méthode ; il est similaire à un pointeur de fonction en C et C++. Toutefois, à la différence des pointeurs de fonction C, les délégués sont orientés objet, de type sécurisé et sûrs. Le type d'un délégué est défini par le nom du délégué. L'exemple suivant déclare un délégué nommé `Callback` qui peut encapsuler une méthode acceptant une chaîne (string) comme argument et qui retourne void :

C#

```
public delegate void Callback(string message);
```

Un objet délégué est normalement construit en fournissant le nom de la méthode que le délégué enveloppera, ou avec une expression lambda. Une fois qu'un délégué est instancié, un appel de méthode fait au délégué est transmis par le délégué à cette méthode. Les paramètres passés au délégué par l'appelant sont passés à la méthode et la valeur de retour de la méthode, le cas échéant, est retournée à l'appelant par le délégué. Cette opération est connue sous le nom d'appel du délégué. Un délégué instancié peut être appelé comme s'il s'agissait de la méthode encapsulée elle-même. Par exemple :

C#

```
// Create a method for a delegate.
public static void DelegateMethod(string message)
{
    Console.WriteLine(message);
}
```

C#

```
// Instantiate the delegate.
Callback handler = DelegateMethod;

// Call the delegate.
handler("Hello World");
```

Les types délégués sont dérivés de la classe Delegate dans .NET. Les types délégués sont scellés (sealed), c'est-à-dire qu'ils ne peuvent pas être faire l'objet d'une dérivation, et il

n’est pas possible de dériver des classes personnalisées à partir de Delegate. Étant donné que le délégué instancié est un objet, il peut être passé comme argument, ou assigné à une propriété. Cela permet à une méthode d'accepter un délégué comme paramètre et d'appeler le délégué ultérieurement. Cette opération est connue sous le nom de rappel asynchrone et constitue une méthode courante pour notifier un appelant quand un long processus s'est achevé. Lorsqu'un délégué est ainsi utilisé, le code qui utilise le délégué n'a pas besoin de connaître l'implémentation de la méthode utilisée. La fonctionnalité est semblable à l'encapsulation que fournissent les interfaces.

Une autre utilisation courante des rappels consiste à définir une méthode de comparaison personnalisée et à passer ce délégué à une méthode de tri. Le code de l'appelant peut ainsi faire partie de l'algorithme de tri. L'exemple de méthode suivant utilise le type `Del` comme paramètre :

C#

```
public static void MethodWithCallback(int param1, int param2, Callback
callback)
{
    callback("The number is: " + (param1 + param2).ToString());
}
```

Vous pouvez ensuite passer le délégué créé ci-dessus à cette méthode :

C#

```
MethodWithCallback(1, 2, handler);
```

et vous recevez le résultat suivant sur la console :

Console

```
The number is: 3
```

En utilisant le délégué comme abstraction, `MethodWithCallback` n'a pas besoin d'appeler la console directement. La méthode n'a pas à être conçue en l'associant obligatoirement à une console. En effet, `MethodWithCallback` se contente de préparer une chaîne et de la passer à une autre méthode. Cela est particulièrement efficace, car une méthode déléguée peut utiliser n'importe quel nombre de paramètres.

Quand un délégué est construit pour encapsuler une méthode d'instance, le délégué fait référence à l'instance et à la méthode. Un délégué n'ayant aucune connaissance du type d'instance hormis la méthode qu'il encapsule, un délégué peut faire référence à

n'importe quel type d'objet tant qu'il existe une méthode sur l'objet correspondant à la signature du délégué. Quand un délégué est construit pour encapsuler une méthode statique, il fait uniquement référence à la méthode. Prenons les déclarations suivantes :

C#

```
public class MethodClass
{
    public void Method1(string message) { }
    public void Method2(string message) { }
}
```

Avec la méthode statique `DelegateMethod` montrée précédemment, nous avons désormais trois méthodes pouvant être encapsulées par une instance `Del`.

Un délégué peut appeler plusieurs méthodes quand il est appelé. Cette opération se nomme multidiffusion. L'ajout d'une méthode supplémentaire à la liste des méthodes du délégué – la liste d'invocation – nécessite simplement l'ajout de deux délégués à l'aide des opérateurs d'addition ou d'assignation d'addition (« + » ou « += »). Par exemple :

C#

```
var obj = new MethodClass();
Callback d1 = obj.Method1;
Callback d2 = obj.Method2;
Callback d3 = DelegateMethod;

//Both types of assignment are valid.
Callback allMethodsDelegate = d1 + d2;
allMethodsDelegate += d3;
```

À ce stade, `allMethodsDelegate` contient trois méthodes dans sa liste d'invocation : `Method1`, `Method2` et `DelegateMethod`. Les trois délégués initiaux, `d1`, `d2` et `d3`, restent inchangés. Lors de l'appel de `allMethodsDelegate`, les trois méthodes sont appelées dans l'ordre. Si le délégué utilise des paramètres de référence, la référence est passée séquentiellement à chacune des trois méthodes et toute modification apportée par une méthode est visible dans la méthode suivante. Quand l'une des méthodes lève une exception qui n'est pas interceptée dans la méthode, cette exception est passée à l'appelant du délégué et aucune des méthodes suivantes dans la liste d'invocation n'est appelée. Si le délégué a une valeur de retour et/ou des paramètres out, il retourne la valeur de retour et les paramètres de la dernière méthode appelée. Pour supprimer une méthode de la liste d'invocation, utilisez les opérateurs de soustraction ou d'assignation de soustraction (`-` ou `-=`). Par exemple :

C#

```
//remove Method1
allMethodsDelegate -= d1;

// copy AllMethodsDelegate while removing d2
Callback oneMethodDelegate = allMethodsDelegate - d2;
```

Étant donné que les types délégués sont dérivés de `System.Delegate`, les méthodes et les propriétés définies par cette classe peuvent être appelées sur le délégué. Par exemple, pour rechercher le nombre de méthodes dans la liste d'invocation d'un délégué, vous pouvez écrire :

C#

```
int invocationCount = d1.GetInvocationList().GetLength(0);
```

Les délégués ayant plusieurs méthodes dans leur liste d'invocation dérivent de MulticastDelegate, qui est une sous-classe de `System.Delegate`. Le code ci-dessus fonctionne dans les deux cas, car les deux classes prennent en charge `GetInvocationList`.

Les délégués multicast sont largement utilisés dans la gestion des événements. Les objets sources d'événements envoient des notifications d'événement aux objets destinataires qui se sont inscrits pour recevoir cet événement. Pour s'inscrire à un événement, le destinataire crée une méthode conçue pour gérer l'événement, puis crée un délégué pour cette méthode et passe le délégué à la source d'événements. La source appelle le délégué quand l'événement se produit. Le délégué appelle ensuite la méthode de gestion d'événement sur le destinataire, en fournissant les données d'événement. Le type délégué d'un événement donné est défini par la source de l'événement. Pour plus d'informations, consultez Événements.

La comparaison de délégués de deux types différents assignés au moment de la compilation provoque une erreur de compilation. Si les instances de délégué sont statiquement du type `System.Delegate`, la comparaison est alors autorisée, mais retourne la valeur false à l'exécution. Par exemple :

C#

```
delegate void Callback1();
delegate void Callback2();

static void method(Callback1 d, Callback2 e, System.Delegate f)
{
```

```
    // Compile-time error.
    //Console.WriteLine(d == e);

    // OK at compile-time. False if the run-time type of f
    // is not the same as that of d.
    Console.WriteLine(d == f);
}
```

Voir aussi

- Guide de programmation C#
- Délégués
- Utilisation de la variance dans les délégués
- Variance dans les délégués
- Utilisation de la variance pour les délégués génériques Func et Action
- Événements

Délégués avec méthodes nommées et méthodes anonymes (Guide de programmation C#)

Article • 07/04/2023

Un délégué peut être associé à une méthode nommée. Quand vous instanciez un délégué à l'aide d'une méthode nommée, la méthode est passée en tant que paramètre, par exemple :

C#

```
// Declare a delegate.
delegate void Del(int x);

// Define a named method.
void DoWork(int k) { /* ... */ }

// Instantiate the delegate using the method as a parameter.
Del d = obj.DoWork;
```

C'est ce qui s'appelle utiliser une méthode nommée. Les délégués construits avec une méthode nommée peuvent encapsuler une méthode static ou une méthode d'instance. Les méthodes nommées représentent la seule façon d'instancier un délégué dans les versions précédentes de C#. Toutefois, dans une situation où la création d'une méthode constitue une charge de travail non souhaitée, C# vous permet d'instancier un délégué et de spécifier immédiatement un bloc de code que le délégué traite quand il est appelé. Le bloc peut contenir une expression lambda ou une méthode anonyme.

La méthode que vous passez en tant que paramètre de délégué doit avoir la même signature que la déclaration du délégué. Une instance de délégué encapsule soit une méthode statique soit une méthode d'instance.

> ⓘ **Notes**
>
> Même si le délégué peut utiliser un paramètre **out**, nous ne recommandons pas son utilisation avec les délégués d'événement multicast, car vous ne pouvez pas savoir quel délégué sera appelé.

À compter de C# 10, les groupes de méthodes avec une seule surcharge possèdent un *type naturel*. Cela signifie que le compilateur peut déduire le type de retour et les types

de paramètres pour le type délégué :

```
C#
```

```
var read = Console.Read; // Just one overload; Func<int> inferred
var write = Console.Write; // ERROR: Multiple overloads, can't choose
```

Exemples

L'exemple suivant est un exemple simple de déclaration et d'utilisation d'un délégué. Notez que le délégué, `Del`, et la méthode associée, `MultiplyNumbers`, ont la même signature

```
C#
```

```
// Declare a delegate
delegate void Del(int i, double j);

class MathClass
{
    static void Main()
    {
        MathClass m = new MathClass();

        // Delegate instantiation using "MultiplyNumbers"
        Del d = m.MultiplyNumbers;

        // Invoke the delegate object.
        Console.WriteLine("Invoking the delegate using 'MultiplyNumbers':");
        for (int i = 1; i <= 5; i++)
        {
            d(i, 2);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    // Declare the associated method.
    void MultiplyNumbers(int m, double n)
    {
        Console.Write(m * n + " ");
    }
}
/* Output:
    Invoking the delegate using 'MultiplyNumbers':
    2 4 6 8 10
*/
```

Dans l'exemple suivant, un délégué est associé à une méthode statique et à une méthode d'instance, et retourne des informations spécifiques à partir de chacune.

C#

```
// Declare a delegate
delegate void Del();

class SampleClass
{
    public void InstanceMethod()
    {
        Console.WriteLine("A message from the instance method.");
    }

    static public void StaticMethod()
    {
        Console.WriteLine("A message from the static method.");
    }
}

class TestSampleClass
{
    static void Main()
    {
        var sc = new SampleClass();

        // Map the delegate to the instance method:
        Del d = sc.InstanceMethod;
        d();

        // Map to the static method:
        d = SampleClass.StaticMethod;
        d();
    }
}
/* Output:
    A message from the instance method.
    A message from the static method.
*/
```

Voir aussi

- Guide de programmation C#
- Délégués
- Guide pratique pour combiner des délégués (délégués multicast)
- Événements

Comment combiner des délégués (délégués multicast) (Guide de programmation C#)

Article • 07/04/2023

Cet exemple explique comment créer des délégués multicast. Une propriété utile des objets délégués est que plusieurs objets peuvent être assignés à une instance de délégué à l'aide de l'opérateur `+`. Le délégué multicast contient une liste des délégués assignés. Quand le délégué multicast est appelé, il appelle les délégués dans la liste, dans l'ordre. Seuls des délégués de même type peuvent être combinés.

Vous pouvez utiliser l'opérateur `-` pour supprimer un délégué de composant d'un délégué multicast.

Exemple

C#

```
using System;

// Define a custom delegate that has a string parameter and returns void.
delegate void CustomDel(string s);

class TestClass
{
    // Define two methods that have the same signature as CustomDel.
    static void Hello(string s)
    {
        Console.WriteLine($"  Hello, {s}!");
    }

    static void Goodbye(string s)
    {
        Console.WriteLine($"  Goodbye, {s}!");
    }

    static void Main()
    {
        // Declare instances of the custom delegate.
        CustomDel hiDel, byeDel, multiDel, multiMinusHiDel;

        // In this example, you can omit the custom delegate if you
        // want to and use Action<string> instead.
        //Action<string> hiDel, byeDel, multiDel, multiMinusHiDel;
```

```
        // Initialize the delegate object hiDel that references the
        // method Hello.
        hiDel = Hello;

        // Initialize the delegate object byeDel that references the
        // method Goodbye.
        byeDel = Goodbye;

        // The two delegates, hiDel and byeDel, are combined to
        // form multiDel.
        multiDel = hiDel + byeDel;

        // Remove hiDel from the multicast delegate, leaving byeDel,
        // which calls only the method Goodbye.
        multiMinusHiDel = multiDel - hiDel;

        Console.WriteLine("Invoking delegate hiDel:");
        hiDel("A");
        Console.WriteLine("Invoking delegate byeDel:");
        byeDel("B");
        Console.WriteLine("Invoking delegate multiDel:");
        multiDel("C");
        Console.WriteLine("Invoking delegate multiMinusHiDel:");
        multiMinusHiDel("D");
    }
}
/* Output:
Invoking delegate hiDel:
  Hello, A!
Invoking delegate byeDel:
  Goodbye, B!
Invoking delegate multiDel:
  Hello, C!
  Goodbye, C!
Invoking delegate multiMinusHiDel:
  Goodbye, D!
*/
```

Voir aussi

- MulticastDelegate
- Guide de programmation C#
- Événements

Guide pratique pour déclarer, instancier et utiliser un délégué (Guide de programmation C#)

Article • 07/04/2023

Vous pouvez déclarer des délégués à l'aide de l'une des méthodes suivantes :

- Déclarez un type délégué et déclarez une méthode avec une signature correspondante :

C#

```
// Declare a delegate.
delegate void Del(string str);

// Declare a method with the same signature as the delegate.
static void Notify(string name)
{
    Console.WriteLine($"Notification received for: {name}");
}
```

C#

```
// Create an instance of the delegate.
Del del1 = new Del(Notify);
```

- Affectez un groupe de méthodes à un type de délégué :

C#

```
// C# 2.0 provides a simpler way to declare an instance of Del.
Del del2 = Notify;
```

- Déclarez une méthode anonyme :

C#

```
// Instantiate Del by using an anonymous method.
Del del3 = delegate(string name)
    { Console.WriteLine($"Notification received for: {name}"); };
```

- Utilisez une expression lambda :

C#

```
// Instantiate Del by using a lambda expression.
Del del4 = name =>  { Console.WriteLine($"Notification received for:
{name}"); };
```

Pour plus d'informations, consultez Expressions lambda.

L'exemple suivant illustre la déclaration, l'instanciation et l'utilisation d'un délégué. La classe `BookDB` encapsule une base de données de librairie qui tient à jour un inventaire des livres. Elle expose une méthode, `ProcessPaperbackBooks`, qui recherche tous les livres de poche dans la base de données et appelle un délégué pour chacun d'entre eux. Le type `delegate` qui est utilisé se nomme `ProcessBookCallback`. La classe `Test` utilise cette classe pour imprimer les titres et le prix moyen des livres de poche.

L'utilisation des délégués favorise une bonne séparation des fonctionnalités entre la base de données de librairie et le code client. Le code client n'a aucune connaissance de la façon dont les livres sont stockés, ni de la façon dont le code librairie trouve les livres de poche. Le code librairie n'a aucune connaissance du traitement effectué sur les livres de poche une fois qu'il les a trouvés.

Exemple

C#

```
// A set of classes for handling a bookstore:
namespace Bookstore
{
    using System.Collections;

    // Describes a book in the book list:
    public struct Book
    {
        public string Title;        // Title of the book.
        public string Author;       // Author of the book.
        public decimal Price;       // Price of the book.
        public bool Paperback;      // Is it paperback?

        public Book(string title, string author, decimal price, bool
paperBack)
        {
            Title = title;
            Author = author;
            Price = price;
            Paperback = paperBack;
        }
    }
```

```
    // Declare a delegate type for processing a book:
    public delegate void ProcessBookCallback(Book book);

    // Maintains a book database.
    public class BookDB
    {
        // List of all books in the database:
        ArrayList list = new ArrayList();

        // Add a book to the database:
        public void AddBook(string title, string author, decimal price, bool
paperBack)
        {
            list.Add(new Book(title, author, price, paperBack));
        }

        // Call a passed-in delegate on each paperback book to process it:
        public void ProcessPaperbackBooks(ProcessBookCallback processBook)
        {
            foreach (Book b in list)
            {
                if (b.Paperback)
                    // Calling the delegate:
                    processBook(b);
            }
        }
    }
}

// Using the Bookstore classes:
namespace BookTestClient
{
    using Bookstore;

    // Class to total and average prices of books:
    class PriceTotaller
    {
        int countBooks = 0;
        decimal priceBooks = 0.0m;

        internal void AddBookToTotal(Book book)
        {
            countBooks += 1;
            priceBooks += book.Price;
        }

        internal decimal AveragePrice()
        {
            return priceBooks / countBooks;
        }
    }

    // Class to test the book database:
    class Test
```

```
    {
        // Print the title of the book.
        static void PrintTitle(Book b)
        {
            Console.WriteLine($"   {b.Title}");
        }

        // Execution starts here.
        static void Main()
        {
            BookDB bookDB = new BookDB();

            // Initialize the database with some books:
            AddBooks(bookDB);

            // Print all the titles of paperbacks:
            Console.WriteLine("Paperback Book Titles:");

            // Create a new delegate object associated with the static
            // method Test.PrintTitle:
            bookDB.ProcessPaperbackBooks(PrintTitle);

            // Get the average price of a paperback by using
            // a PriceTotaller object:
            PriceTotaller totaller = new PriceTotaller();

            // Create a new delegate object associated with the nonstatic
            // method AddBookToTotal on the object totaller:
            bookDB.ProcessPaperbackBooks(totaller.AddBookToTotal);

            Console.WriteLine("Average Paperback Book Price: ${0:#.##}",
                    totaller.AveragePrice());
        }

        // Initialize the book database with some test books:
        static void AddBooks(BookDB bookDB)
        {
            bookDB.AddBook("The C Programming Language", "Brian W. Kernighan
and Dennis M. Ritchie", 19.95m, true);
            bookDB.AddBook("The Unicode Standard 2.0", "The Unicode
Consortium", 39.95m, true);
            bookDB.AddBook("The MS-DOS Encyclopedia", "Ray Duncan", 129.95m,
false);
            bookDB.AddBook("Dogbert's Clues for the Clueless", "Scott
Adams", 12.00m, true);
        }
    }
}
/* Output:
Paperback Book Titles:
   The C Programming Language
   The Unicode Standard 2.0
   Dogbert's Clues for the Clueless
Average Paperback Book Price: $23.97
*/
```

Programmation fiable

- Déclaration d'un délégué.

  L'instruction suivante déclare un nouveau type délégué.

  ```
  C#
  ```

  ```
  public delegate void ProcessBookCallback(Book book);
  ```

  Chaque type délégué décrit le nombre et les types des arguments, ainsi que le type de la valeur de retour des méthodes qu'il peut encapsuler. Chaque fois qu'un nouvel ensemble de types d'arguments ou de types de valeur de retour est nécessaire, un nouveau type délégué doit être déclaré.

- Instanciation d'un délégué.

  Une fois que vous avez déclaré un type délégué, vous devez créer un objet délégué et l'associer à une méthode particulière. Pour cela, dans l'exemple précédent vous passez la méthode `PrintTitle` à la méthode `ProcessPaperbackBooks`, comme illustré dans l'exemple suivant :

  ```
  C#
  ```

  ```
  bookDB.ProcessPaperbackBooks(PrintTitle);
  ```

  Cela crée un nouvel objet délégué associé à la méthode statique `Test.PrintTitle`. De même, la méthode non statique `AddBookToTotal` sur l'objet `totaller` est passée comme dans l'exemple suivant :

  ```
  C#
  ```

  ```
  bookDB.ProcessPaperbackBooks(totaller.AddBookToTotal);
  ```

  Dans les deux cas, un nouvel objet délégué est passé à la méthode `ProcessPaperbackBooks`.

  Une fois un délégué créé, la méthode à laquelle il est associé ne change jamais. Les objets délégué sont immuables.

- Appel d'un délégué.

Une fois créé, un objet délégué est généralement passé à un autre code qui appellera le délégué. Pour appeler un objet délégué, vous devez utiliser le nom de l'objet délégué, suivi des arguments entre parenthèses à passer au délégué. Voici un exemple d'appel de délégué :

```
C#

processBook(b);
```

Vous pouvez appeler un délégué de manière synchrone, comme dans cet exemple, ou de manière asynchrone à l'aide des méthodes `BeginInvoke` et `EndInvoke`.

Voir aussi

- Guide de programmation C#
- Événements
- Délégués

Tableaux

Article • 04/09/2023

Vous pouvez stocker plusieurs variables du même type dans une structure de données de type tableau. Vous déclarez un tableau en spécifiant le type de ses éléments. Si vous souhaitez que le tableau stocke des éléments de n'importe quel type, vous pouvez spécifier `object` comme type. Dans le système de type unifié de C#, tous les types (les types référence et valeur, prédéfinis ou définis par l'utilisateur) héritent directement ou indirectement du type Object.

```
C#

type[] arrayName;
```

Un tableau possède les propriétés suivantes :

- Un tableau peut être unidimensionnel, multidimensionnel ou en escalier.
- Le nombre de dimensions est défini lorsqu'une variable de tableau est déclarée. La longueur de chaque dimension est établie lors de la création de l'instance de tableau. Ces valeurs ne peuvent pas être modifiées pendant la durée de vie de l'instance.
- Un tableau en escalier est un tableau de tableaux, et chaque tableau membre a la valeur `null` par défaut.
- Les tableaux sont indexés sur zéro : un tableau avec `n` éléments est indexée de `0` à `n-1`.
- Les éléments de tableau peuvent être de n'importe quel type, y compris un type tableau.
- Les types tableau sont des types référence dérivés du type de base abstrait Array. Tous les tableaux implémentent IList et IEnumerable. Vous pouvez utiliser l'instruction foreach pour itérer au sein d'un tableau. Les tableaux unidimensionnels implémentent aussi IList<T> et IEnumerable<T>.

Les éléments d'un tableau peuvent être initialisés sur des valeurs connues lors de la création du tableau. À compter de C# 12, tous les types de collection peuvent être initialisés à l'aide d'une expression de collection. Les éléments qui ne sont pas initialisés sont définis sur la valeur par défaut. La valeur par défaut est le modèle 0 bits. Tous les types de référence (y compris les types non-nullables) ont les valeurs `null`. Tous les types de valeur ont les modèles 0 bits. Cela signifie que la propriété Nullable<T>.HasValue est `false` et que la propriété Nullable<T>.Value n'est pas définie. Dans l'implémentation .NET, la propriété `Value` lève une exception.

L'exemple suivant crée des tableaux unidimensionnels, multidimensionnels et en escalier :

```
C#
```

```
// Declare a single-dimensional array of 5 integers.
int[] array1 = new int[5];

// Declare and set array element values.
int[] array2 = { 1, 2, 3, 4, 5, 6 };

// Declare a two dimensional array.
int[,] multiDimensionalArray1 = new int[2, 3];

// Declare and set array element values.
int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };

// Declare a jagged array.
int[][] jaggedArray = new int[6][];

// Set the values of the first array in the jagged array structure.
jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
```

Tableaux unidimensionnels

Un *tableau unidimensionnel* est une séquence d'éléments similaires. Vous accédez à un élément via son *index*. L'*index* est sa position ordinale dans la séquence. Le premier élément du tableau est à l'index `0`. Vous créez un tableau unidimensionnel à l'aide du nouvel opérateur en spécifiant le type d'élément du tableau et le nombre d'éléments. L'exemple suivant déclare et initialise des tableaux unidimensionnels :

```
C#
```

```
int[] array = new int[5];
string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };

Console.WriteLine(weekDays[0]);
Console.WriteLine(weekDays[1]);
Console.WriteLine(weekDays[2]);
Console.WriteLine(weekDays[3]);
Console.WriteLine(weekDays[4]);
Console.WriteLine(weekDays[5]);
Console.WriteLine(weekDays[6]);

/*Output:
Sun
Mon
Tue
Wed
```

```
Thu
Fri
Sat
*/
```

La première déclaration déclare un tableau non initialisé de cinq entiers, de `array[0]` à `array[4]`. Les éléments du tableau sont initialisés à la valeur par défaut du type d'élément, `0` pour les entiers. La deuxième déclaration déclare un tableau de chaînes et initialise les sept valeurs de ce tableau. Une instruction foreach itère les éléments du tableau `weekday` et affiche toutes les valeurs. Pour les tableaux unidimensionnels, l'instruction `foreach` traite les éléments dans l'ordre croissant des index, en commençant par l'index 0 et en terminant par l'index `Length - 1`.

Passer des tableaux unidimensionnels en tant qu'arguments

Vous pouvez passer un tableau unidimensionnel initialisé à une méthode. Dans l'exemple suivant, un tableau de chaînes est initialisé et passé en tant qu'argument à une méthode `DisplayArray` pour des chaînes. La méthode affiche les éléments du tableau. Ensuite, la méthode `ChangeArray` inverse les éléments du tableau, puis la méthode `ChangeArrayElements` modifie les trois premiers éléments du tableau. Une fois que chaque méthode est retournée, la méthode `DisplayArray` montre que le passage d'un tableau par valeur n'empêche pas les modifications des éléments du tableau.

C#

```
class ArrayExample
{
    static void DisplayArray(string[] arr) =>
Console.WriteLine(string.Join(" ", arr));

    // Change the array by reversing its elements.
    static void ChangeArray(string[] arr) => Array.Reverse(arr);

    static void ChangeArrayElements(string[] arr)
    {
        // Change the value of the first three array elements.
        arr[0] = "Mon";
        arr[1] = "Wed";
        arr[2] = "Fri";
    }

    static void Main()
    {
        // Declare and initialize an array.
        string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri",
```

```
"Sat" };
        // Display the array elements.
        DisplayArray(weekDays);
        Console.WriteLine();

        // Reverse the array.
        ChangeArray(weekDays);
        // Display the array again to verify that it stays reversed.
        Console.WriteLine("Array weekDays after the call to ChangeArray:");
        DisplayArray(weekDays);
        Console.WriteLine();

        // Assign new values to individual array elements.
        ChangeArrayElements(weekDays);
        // Display the array again to verify that it has changed.
        Console.WriteLine("Array weekDays after the call to
ChangeArrayElements:");
        DisplayArray(weekDays);
    }
}
// The example displays the following output:
//         Sun Mon Tue Wed Thu Fri Sat
//
//        Array weekDays after the call to ChangeArray:
//        Sat Fri Thu Wed Tue Mon Sun
//
//        Array weekDays after the call to ChangeArrayElements:
//        Mon Wed Fri Wed Tue Mon Sun
```

Tableaux multidimensionnels

Les tableaux peuvent avoir plusieurs dimensions. Par exemple, les déclarations suivantes créent quatre tableaux : deux ont deux dimensions, deux ont trois dimensions. Les deux premières déclarations déclarent la longueur de chaque dimension, mais n'initialisent pas les valeurs du tableau. Les deux deuxièmes déclarations utilisent un initialiseur pour définir les valeurs de chaque élément dans le tableau multidimensionnel.

C#

```
int[,] array2DDeclaration = new int[4, 2];

int[,,] array3DDeclaration = new int[4, 2, 3];

// Two-dimensional array.
int[,] array2DInitialization =  { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// Three-dimensional array.
int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,   5,  6 } },
                                { { 7, 8, 9 }, { 10, 11, 12 } } };

// Accessing array elements.
```

```
System.Console.WriteLine(array2DInitialization[0, 0]);
System.Console.WriteLine(array2DInitialization[0, 1]);
System.Console.WriteLine(array2DInitialization[1, 0]);
System.Console.WriteLine(array2DInitialization[1, 1]);

System.Console.WriteLine(array2DInitialization[3, 0]);
System.Console.WriteLine(array2DInitialization[3, 1]);
// Output:
// 1
// 2
// 3
// 4
// 7
// 8

System.Console.WriteLine(array3D[1, 0, 1]);
System.Console.WriteLine(array3D[1, 1, 2]);
// Output:
// 8
// 12

// Getting the total count of elements or the length of a given dimension.
var allLength = array3D.Length;
var total = 1;
for (int i = 0; i < array3D.Rank; i++)
{
    total *= array3D.GetLength(i);
}
System.Console.WriteLine($"{allLength} equals {total}");
// Output:
// 12 equals 12
```

Pour les tableaux multidimensionnels, les éléments sont traités de sorte que les index de la dimension la plus à droite sont incrémentés en premier, puis la dimension immédiatement à gauche, et ainsi de suite jusqu'à l'index le plus à gauche. L'exemple suivant énumère à la fois un tableau 2D et un tableau 3D :

C#

```
int[,] numbers2D = { { 9, 99 }, { 3, 33 }, { 5, 55 } };

foreach (int i in numbers2D)
{
    System.Console.Write($"{i} ");
}
// Output: 9 99 3 33 5 55

int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,   5,  6 } },
                          { { 7, 8, 9 }, { 10, 11, 12 } } };
foreach (int i in array3D)
{
    System.Console.Write($"{i} ");
```

```
}
// Output: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
```

Dans un tableau 2D, vous pouvez considérer l'index de gauche comme la *ligne* et l'index de droite comme la *colonne*.

Cependant, dans le cas de tableaux multidimensionnels, l'utilisation d'une boucle for imbriquée vous permet de mieux contrôler l'ordre dans lequel les éléments du tableau sont traités :

C#

```
int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,   5,  6 } },
                        { { 7, 8, 9 }, { 10, 11, 12 } } };

for (int i = 0; i < array3D.GetLength(0); i++)
{
    for (int j = 0; j < array3D.GetLength(1); j++)
    {
        for (int k = 0; k < array3D.GetLength(2); k++)
        {
            System.Console.Write($"{array3D[i, j, k]} ");
        }
        System.Console.WriteLine();
    }
    System.Console.WriteLine();
}
// Output (including blank lines):
// 1 2 3
// 4 5 6
//
// 7 8 9
// 10 11 12
//
```

Passer des tableaux multidimensionnels en tant qu'arguments

Vous pouvez passer un tableau multidimensionnel initialisé à une méthode de la même manière que vous passez un tableau unidimensionnel. Le code suivant illustre une déclaration partielle d'une méthode Print qui accepte un tableau à deux dimensions en tant qu'argument. Vous pouvez initialiser et passer un nouveau tableau en une seule étape, comme dans l'exemple suivant. Dans l'exemple suivant, un tableau à deux dimensions d'entiers est initialisé et passé à la méthode `Print2DArray`. La méthode affiche les éléments du tableau.

C#

```
static void Print2DArray(int[,] arr)
{
    // Display the array elements.
    for (int i = 0; i < arr.GetLength(0); i++)
    {
        for (int j = 0; j < arr.GetLength(1); j++)
        {
            System.Console.WriteLine("Element({0},{1})={2}", i, j, arr[i,
j]);
        }
    }
}
static void ExampleUsage()
{
    // Pass the array as an argument.
    Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } });
}
/* Output:
    Element(0,0)=1
    Element(0,1)=2
    Element(1,0)=3
    Element(1,1)=4
    Element(2,0)=5
    Element(2,1)=6
    Element(3,0)=7
    Element(3,1)=8
*/
```

Tableaux en escalier

Un tableau en escalier est un tableau dont les éléments sont des tableaux, éventuellement de différentes tailles. Un tableau en escalier est parfois appelé « tableau de tableaux ». Ses éléments sont des types de référence et sont initialisés sur `null`. Les exemples suivants montrent comment déclarer, initialiser et accéder aux tableaux en escalier. Le premier exemple, `jaggedArray`, est déclaré dans une instruction. Chaque tableau contenu est créé dans les instructions suivantes. Le deuxième exemple, `jaggedArray2`, est déclaré et initialisé dans une instruction. Il est possible de combiner des tableaux en escalier et des tableaux multidimensionnels. Le dernier exemple, `jaggedArray3`, déclare et initialise un tableau en escalier unidimensionnel composé de trois éléments de tableau à deux dimensions de tailles différentes.

C#

```
int[][] jaggedArray = new int[3][];
```

```
jaggedArray[0] = new int[] { 1, 3, 5, 7, 9 };
jaggedArray[1] = new int[] { 0, 2, 4, 6 };
jaggedArray[2] = new int[] { 11, 22 };

int[][] jaggedArray2 =
{
    new int[] { 1, 3, 5, 7, 9 },
    new int[] { 0, 2, 4, 6 },
    new int[] { 11, 22 }
};

// Assign 77 to the second element ([1]) of the first array ([0]):
jaggedArray2[0][1] = 77;

// Assign 88 to the second element ([1]) of the third array ([2]):
jaggedArray2[2][1] = 88;

int[][,] jaggedArray3 = new int[3][,]
{
    new int[,] { {1,3}, {5,7} },
    new int[,] { {0,2}, {4,6}, {8,10} },
    new int[,] { {11,22}, {99,88}, {0,9} }
};

Console.Write("{0}", jaggedArray3[0][1, 0]);
Console.WriteLine(jaggedArray3.Length);
```

Les éléments d'un tableau en escalier doivent être initialisés avant de pouvoir les utiliser. Chacun des éléments est lui-même un tableau. Il est également possible d'utiliser des initialiseurs pour remplir les éléments de tableau avec des valeurs. Lorsque vous utilisez des initialiseurs, vous n'avez pas besoin de la taille du tableau.

Cet exemple génère un tableau dont les éléments sont eux-mêmes des tableaux. Chacun des éléments de tableau ont une taille différente.

C#

```
// Declare the array of two elements.
int[][] arr = new int[2][];

// Initialize the elements.
arr[0] = new int[5] { 1, 3, 5, 7, 9 };
arr[1] = new int[4] { 2, 4, 6, 8 };

// Display the array elements.
for (int i = 0; i < arr.Length; i++)
{
    System.Console.Write("Element({0}): ", i);

    for (int j = 0; j < arr[i].Length; j++)
    {
        System.Console.Write("{0}{1}", arr[i][j], j == (arr[i].Length - 1) ?
```

```
"" : " ");
    }
    System.Console.WriteLine();
}
/* Output:
    Element(0): 1 3 5 7 9
    Element(1): 2 4 6 8
*/
```

Tableaux implicitement typés

Vous pouvez créer un tableau implicitement typé dans lequel le type de l'instance de tableau est déduit à partir des éléments spécifiés dans l'initialiseur de tableau. Les règles applicables à une variable implicitement typée s'appliquent aussi aux tableaux implicitement typés. Pour plus d'informations, consultez Variables locales implicitement typées.

Les exemples suivants montrent comment créer un tableau implicitement typé :

C#

```
var a = new[] { 1, 10, 100, 1000 }; // int[]

// Accessing array
Console.WriteLine("First element: " + a[0]);
Console.WriteLine("Second element: " + a[1]);
Console.WriteLine("Third element: " + a[2]);
Console.WriteLine("Fourth element: " + a[3]);
/* Outputs
First element: 1
Second element: 10
Third element: 100
Fourth element: 1000
*/

var b = new[] { "hello", null, "world" }; // string[]

// Accessing elements of an array using 'string.Join' method
Console.WriteLine(string.Join(" ", b));
/* Output
hello  world
*/

// single-dimension jagged array
var c = new[]
{
                                    new[]{1,2,3,4},
                                    new[]{5,6,7,8}
            };
// Looping through the outer array
```

```
for (int k = 0; k < c.Length; k++)
{
    // Looping through each inner array
    for (int j = 0; j < c[k].Length; j++)
    {
        // Accessing each element and printing it to the console
        Console.WriteLine($"Element at c[{k}][{j}] is: {c[k][j]}");
    }
}
/* Outputs
Element at c[0][0] is: 1
Element at c[0][1] is: 2
Element at c[0][2] is: 3
Element at c[0][3] is: 4
Element at c[1][0] is: 5
Element at c[1][1] is: 6
Element at c[1][2] is: 7
Element at c[1][3] is: 8
*/

// jagged array of strings
var d = new[]
{
                new[]{"Luca", "Mads", "Luke", "Dinesh"},
                new[]{"Karen", "Suma", "Frances"}
            };

// Looping through the outer array
int i = 0;
foreach (var subArray in d)
{
    // Looping through each inner array
    int j = 0;
    foreach (var element in subArray)
    {
        // Accessing each element and printing it to the console
        Console.WriteLine($"Element at d[{i}][{j}] is: {element}");
        j++;
    }
    i++;
}
/* Outputs
Element at d[0][0] is: Luca
Element at d[0][1] is: Mads
Element at d[0][2] is: Luke
Element at d[0][3] is: Dinesh
Element at d[1][0] is: Karen
Element at d[1][1] is: Suma
Element at d[1][2] is: Frances
*/
```

Dans l'exemple précédent, vous pouvez constater que dans le cas des tableaux implicitement typés, aucun crochet n'est utilisé à gauche de l'instruction d'initialisation.

De plus, les tableaux en escalier sont initialisés à l'aide de `new []` à l'instar des tableaux unidimensionnels.

Quand vous créez un type anonyme qui contient un tableau, le tableau doit être implicitement typé dans l'initialiseur d'objet du type. Dans l'exemple suivant, `contacts` est un tableau implicitement typé de types anonymes, dont chacun contient un tableau nommé `PhoneNumbers`. Le mot clé `var` n'est pas utilisé dans les initialiseurs d'objets.

C#

```
var contacts = new[]
{
        new {
            Name = " Eugene Zabokritski",
            PhoneNumbers = new[] { "206-555-0108", "425-555-0001" }
        },
        new {
            Name = " Hanying Feng",
            PhoneNumbers = new[] { "650-555-0199" }
        }
    };
```

Tableaux

Article • 04/09/2023

Vous pouvez stocker plusieurs variables du même type dans une structure de données de type tableau. Vous déclarez un tableau en spécifiant le type de ses éléments. Si vous souhaitez que le tableau stocke des éléments de n'importe quel type, vous pouvez spécifier `object` comme type. Dans le système de type unifié de C#, tous les types (les types référence et valeur, prédéfinis ou définis par l'utilisateur) héritent directement ou indirectement du type Object.

```
C#

type[] arrayName;
```

Un tableau possède les propriétés suivantes :

- Un tableau peut être unidimensionnel, multidimensionnel ou en escalier.
- Le nombre de dimensions est défini lorsqu'une variable de tableau est déclarée. La longueur de chaque dimension est établie lors de la création de l'instance de tableau. Ces valeurs ne peuvent pas être modifiées pendant la durée de vie de l'instance.
- Un tableau en escalier est un tableau de tableaux, et chaque tableau membre a la valeur `null` par défaut.
- Les tableaux sont indexés sur zéro : un tableau avec `n` éléments est indexée de `0` à `n-1`.
- Les éléments de tableau peuvent être de n'importe quel type, y compris un type tableau.
- Les types tableau sont des types référence dérivés du type de base abstrait Array. Tous les tableaux implémentent IList et IEnumerable. Vous pouvez utiliser l'instruction foreach pour itérer au sein d'un tableau. Les tableaux unidimensionnels implémentent aussi IList<T> et IEnumerable<T>.

Les éléments d'un tableau peuvent être initialisés sur des valeurs connues lors de la création du tableau. À compter de C# 12, tous les types de collection peuvent être initialisés à l'aide d'une expression de collection. Les éléments qui ne sont pas initialisés sont définis sur la valeur par défaut. La valeur par défaut est le modèle 0 bits. Tous les types de référence (y compris les types non-nullables) ont les valeurs `null`. Tous les types de valeur ont les modèles 0 bits. Cela signifie que la propriété Nullable<T>.HasValue est `false` et que la propriété Nullable<T>.Value n'est pas définie. Dans l'implémentation .NET, la propriété `Value` lève une exception.

L'exemple suivant crée des tableaux unidimensionnels, multidimensionnels et en escalier :

```
C#
```

```
// Declare a single-dimensional array of 5 integers.
int[] array1 = new int[5];

// Declare and set array element values.
int[] array2 = { 1, 2, 3, 4, 5, 6 };

// Declare a two dimensional array.
int[,] multiDimensionalArray1 = new int[2, 3];

// Declare and set array element values.
int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };

// Declare a jagged array.
int[][] jaggedArray = new int[6][];

// Set the values of the first array in the jagged array structure.
jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
```

Tableaux unidimensionnels

Un *tableau unidimensionnel* est une séquence d'éléments similaires. Vous accédez à un élément via son *index*. L'*index* est sa position ordinale dans la séquence. Le premier élément du tableau est à l'index `0`. Vous créez un tableau unidimensionnel à l'aide du nouvel opérateur en spécifiant le type d'élément du tableau et le nombre d'éléments. L'exemple suivant déclare et initialise des tableaux unidimensionnels :

```
C#
```

```
int[] array = new int[5];
string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };

Console.WriteLine(weekDays[0]);
Console.WriteLine(weekDays[1]);
Console.WriteLine(weekDays[2]);
Console.WriteLine(weekDays[3]);
Console.WriteLine(weekDays[4]);
Console.WriteLine(weekDays[5]);
Console.WriteLine(weekDays[6]);

/*Output:
Sun
Mon
Tue
Wed
```

```
Thu
Fri
Sat
*/
```

La première déclaration déclare un tableau non initialisé de cinq entiers, de `array[0]` à `array[4]`. Les éléments du tableau sont initialisés à la valeur par défaut du type d'élément, `0` pour les entiers. La deuxième déclaration déclare un tableau de chaînes et initialise les sept valeurs de ce tableau. Une instruction foreach itère les éléments du tableau `weekday` et affiche toutes les valeurs. Pour les tableaux unidimensionnels, l'instruction `foreach` traite les éléments dans l'ordre croissant des index, en commençant par l'index 0 et en terminant par l'index `Length - 1`.

Passer des tableaux unidimensionnels en tant qu'arguments

Vous pouvez passer un tableau unidimensionnel initialisé à une méthode. Dans l'exemple suivant, un tableau de chaînes est initialisé et passé en tant qu'argument à une méthode `DisplayArray` pour des chaînes. La méthode affiche les éléments du tableau. Ensuite, la méthode `ChangeArray` inverse les éléments du tableau, puis la méthode `ChangeArrayElements` modifie les trois premiers éléments du tableau. Une fois que chaque méthode est retournée, la méthode `DisplayArray` montre que le passage d'un tableau par valeur n'empêche pas les modifications des éléments du tableau.

C#

```
class ArrayExample
{
    static void DisplayArray(string[] arr) =>
Console.WriteLine(string.Join(" ", arr));

    // Change the array by reversing its elements.
    static void ChangeArray(string[] arr) => Array.Reverse(arr);

    static void ChangeArrayElements(string[] arr)
    {
        // Change the value of the first three array elements.
        arr[0] = "Mon";
        arr[1] = "Wed";
        arr[2] = "Fri";
    }

    static void Main()
    {
        // Declare and initialize an array.
        string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri",
```

```
"Sat" };
        // Display the array elements.
        DisplayArray(weekDays);
        Console.WriteLine();

        // Reverse the array.
        ChangeArray(weekDays);
        // Display the array again to verify that it stays reversed.
        Console.WriteLine("Array weekDays after the call to ChangeArray:");
        DisplayArray(weekDays);
        Console.WriteLine();

        // Assign new values to individual array elements.
        ChangeArrayElements(weekDays);
        // Display the array again to verify that it has changed.
        Console.WriteLine("Array weekDays after the call to
ChangeArrayElements:");
        DisplayArray(weekDays);
    }
}
// The example displays the following output:
//         Sun Mon Tue Wed Thu Fri Sat
//
//        Array weekDays after the call to ChangeArray:
//        Sat Fri Thu Wed Tue Mon Sun
//
//        Array weekDays after the call to ChangeArrayElements:
//        Mon Wed Fri Wed Tue Mon Sun
```

Tableaux multidimensionnels

Les tableaux peuvent avoir plusieurs dimensions. Par exemple, les déclarations suivantes créent quatre tableaux : deux ont deux dimensions, deux ont trois dimensions. Les deux premières déclarations déclarent la longueur de chaque dimension, mais n'initialisent pas les valeurs du tableau. Les deux deuxièmes déclarations utilisent un initialiseur pour définir les valeurs de chaque élément dans le tableau multidimensionnel.

C#

```
int[,] array2DDeclaration = new int[4, 2];

int[,,] array3DDeclaration = new int[4, 2, 3];

// Two-dimensional array.
int[,] array2DInitialization =  { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// Three-dimensional array.
int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,   5,  6 } },
                                { { 7, 8, 9 }, { 10, 11, 12 } } };

// Accessing array elements.
```

```
System.Console.WriteLine(array2DInitialization[0, 0]);
System.Console.WriteLine(array2DInitialization[0, 1]);
System.Console.WriteLine(array2DInitialization[1, 0]);
System.Console.WriteLine(array2DInitialization[1, 1]);

System.Console.WriteLine(array2DInitialization[3, 0]);
System.Console.WriteLine(array2DInitialization[3, 1]);
// Output:
// 1
// 2
// 3
// 4
// 7
// 8

System.Console.WriteLine(array3D[1, 0, 1]);
System.Console.WriteLine(array3D[1, 1, 2]);
// Output:
// 8
// 12

// Getting the total count of elements or the length of a given dimension.
var allLength = array3D.Length;
var total = 1;
for (int i = 0; i < array3D.Rank; i++)
{
    total *= array3D.GetLength(i);
}
System.Console.WriteLine($"{allLength} equals {total}");
// Output:
// 12 equals 12
```

Pour les tableaux multidimensionnels, les éléments sont traités de sorte que les index de la dimension la plus à droite sont incrémentés en premier, puis la dimension immédiatement à gauche, et ainsi de suite jusqu'à l'index le plus à gauche. L'exemple suivant énumère à la fois un tableau 2D et un tableau 3D :

C#

```
int[,] numbers2D = { { 9, 99 }, { 3, 33 }, { 5, 55 } };

foreach (int i in numbers2D)
{
    System.Console.Write($"{i} ");
}
// Output: 9 99 3 33 5 55

int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,   5,  6 } },
                          { { 7, 8, 9 }, { 10, 11, 12 } } };
foreach (int i in array3D)
{
    System.Console.Write($"{i} ");
```

```
}
// Output: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
```

Dans un tableau 2D, vous pouvez considérer l'index de gauche comme la *ligne* et l'index de droite comme la *colonne*.

Cependant, dans le cas de tableaux multidimensionnels, l'utilisation d'une boucle for imbriquée vous permet de mieux contrôler l'ordre dans lequel les éléments du tableau sont traités :

C#

```
int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,   5,  6 } },
                        { { 7, 8, 9 }, { 10, 11, 12 } } };

for (int i = 0; i < array3D.GetLength(0); i++)
{
    for (int j = 0; j < array3D.GetLength(1); j++)
    {
        for (int k = 0; k < array3D.GetLength(2); k++)
        {
            System.Console.Write($"{array3D[i, j, k]} ");
        }
        System.Console.WriteLine();
    }
    System.Console.WriteLine();
}
// Output (including blank lines):
// 1 2 3
// 4 5 6
//
// 7 8 9
// 10 11 12
//
```

Passer des tableaux multidimensionnels en tant qu'arguments

Vous pouvez passer un tableau multidimensionnel initialisé à une méthode de la même manière que vous passez un tableau unidimensionnel. Le code suivant illustre une déclaration partielle d'une méthode Print qui accepte un tableau à deux dimensions en tant qu'argument. Vous pouvez initialiser et passer un nouveau tableau en une seule étape, comme dans l'exemple suivant. Dans l'exemple suivant, un tableau à deux dimensions d'entiers est initialisé et passé à la méthode `Print2DArray`. La méthode affiche les éléments du tableau.

C#

```
static void Print2DArray(int[,] arr)
{
    // Display the array elements.
    for (int i = 0; i < arr.GetLength(0); i++)
    {
        for (int j = 0; j < arr.GetLength(1); j++)
        {
            System.Console.WriteLine("Element({0},{1})={2}", i, j, arr[i,
j]);
        }
    }
}
static void ExampleUsage()
{
    // Pass the array as an argument.
    Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } });
}
/* Output:
    Element(0,0)=1
    Element(0,1)=2
    Element(1,0)=3
    Element(1,1)=4
    Element(2,0)=5
    Element(2,1)=6
    Element(3,0)=7
    Element(3,1)=8
*/
```

Tableaux en escalier

Un tableau en escalier est un tableau dont les éléments sont des tableaux, éventuellement de différentes tailles. Un tableau en escalier est parfois appelé « tableau de tableaux ». Ses éléments sont des types de référence et sont initialisés sur `null`. Les exemples suivants montrent comment déclarer, initialiser et accéder aux tableaux en escalier. Le premier exemple, `jaggedArray`, est déclaré dans une instruction. Chaque tableau contenu est créé dans les instructions suivantes. Le deuxième exemple, `jaggedArray2`, est déclaré et initialisé dans une instruction. Il est possible de combiner des tableaux en escalier et des tableaux multidimensionnels. Le dernier exemple, `jaggedArray3`, déclare et initialise un tableau en escalier unidimensionnel composé de trois éléments de tableau à deux dimensions de tailles différentes.

C#

```
int[][] jaggedArray = new int[3][];
```

```
jaggedArray[0] = new int[] { 1, 3, 5, 7, 9 };
jaggedArray[1] = new int[] { 0, 2, 4, 6 };
jaggedArray[2] = new int[] { 11, 22 };

int[][] jaggedArray2 =
{
    new int[] { 1, 3, 5, 7, 9 },
    new int[] { 0, 2, 4, 6 },
    new int[] { 11, 22 }
};

// Assign 77 to the second element ([1]) of the first array ([0]):
jaggedArray2[0][1] = 77;

// Assign 88 to the second element ([1]) of the third array ([2]):
jaggedArray2[2][1] = 88;

int[][,] jaggedArray3 = new int[3][,]
{
    new int[,] { {1,3}, {5,7} },
    new int[,] { {0,2}, {4,6}, {8,10} },
    new int[,] { {11,22}, {99,88}, {0,9} }
};

Console.Write("{0}", jaggedArray3[0][1, 0]);
Console.WriteLine(jaggedArray3.Length);
```

Les éléments d'un tableau en escalier doivent être initialisés avant de pouvoir les utiliser. Chacun des éléments est lui-même un tableau. Il est également possible d'utiliser des initialiseurs pour remplir les éléments de tableau avec des valeurs. Lorsque vous utilisez des initialiseurs, vous n'avez pas besoin de la taille du tableau.

Cet exemple génère un tableau dont les éléments sont eux-mêmes des tableaux. Chacun des éléments de tableau ont une taille différente.

C#

```
// Declare the array of two elements.
int[][] arr = new int[2][];

// Initialize the elements.
arr[0] = new int[5] { 1, 3, 5, 7, 9 };
arr[1] = new int[4] { 2, 4, 6, 8 };

// Display the array elements.
for (int i = 0; i < arr.Length; i++)
{
    System.Console.Write("Element({0}): ", i);

    for (int j = 0; j < arr[i].Length; j++)
    {
        System.Console.Write("{0}{1}", arr[i][j], j == (arr[i].Length - 1) ?
```

```
"" : " ");
    }
    System.Console.WriteLine();
}
/* Output:
    Element(0): 1 3 5 7 9
    Element(1): 2 4 6 8
*/
```

Tableaux implicitement typés

Vous pouvez créer un tableau implicitement typé dans lequel le type de l'instance de tableau est déduit à partir des éléments spécifiés dans l'initialiseur de tableau. Les règles applicables à une variable implicitement typée s'appliquent aussi aux tableaux implicitement typés. Pour plus d'informations, consultez Variables locales implicitement typées.

Les exemples suivants montrent comment créer un tableau implicitement typé :

C#

```
var a = new[] { 1, 10, 100, 1000 }; // int[]

// Accessing array
Console.WriteLine("First element: " + a[0]);
Console.WriteLine("Second element: " + a[1]);
Console.WriteLine("Third element: " + a[2]);
Console.WriteLine("Fourth element: " + a[3]);
/* Outputs
First element: 1
Second element: 10
Third element: 100
Fourth element: 1000
*/

var b = new[] { "hello", null, "world" }; // string[]

// Accessing elements of an array using 'string.Join' method
Console.WriteLine(string.Join(" ", b));
/* Output
hello  world
*/

// single-dimension jagged array
var c = new[]
{
                                    new[]{1,2,3,4},
                                    new[]{5,6,7,8}
            };
// Looping through the outer array
```

```
for (int k = 0; k < c.Length; k++)
{
    // Looping through each inner array
    for (int j = 0; j < c[k].Length; j++)
    {
        // Accessing each element and printing it to the console
        Console.WriteLine($"Element at c[{k}][{j}] is: {c[k][j]}");
    }
}
/* Outputs
Element at c[0][0] is: 1
Element at c[0][1] is: 2
Element at c[0][2] is: 3
Element at c[0][3] is: 4
Element at c[1][0] is: 5
Element at c[1][1] is: 6
Element at c[1][2] is: 7
Element at c[1][3] is: 8
*/

// jagged array of strings
var d = new[]
{
                new[]{"Luca", "Mads", "Luke", "Dinesh"},
                new[]{"Karen", "Suma", "Frances"}
            };

// Looping through the outer array
int i = 0;
foreach (var subArray in d)
{
    // Looping through each inner array
    int j = 0;
    foreach (var element in subArray)
    {
        // Accessing each element and printing it to the console
        Console.WriteLine($"Element at d[{i}][{j}] is: {element}");
        j++;
    }
    i++;
}
/* Outputs
Element at d[0][0] is: Luca
Element at d[0][1] is: Mads
Element at d[0][2] is: Luke
Element at d[0][3] is: Dinesh
Element at d[1][0] is: Karen
Element at d[1][1] is: Suma
Element at d[1][2] is: Frances
*/
```

Dans l'exemple précédent, vous pouvez constater que dans le cas des tableaux implicitement typés, aucun crochet n'est utilisé à gauche de l'instruction d'initialisation.

De plus, les tableaux en escalier sont initialisés à l'aide de `new []` à l'instar des tableaux unidimensionnels.

Quand vous créez un type anonyme qui contient un tableau, le tableau doit être implicitement typé dans l'initialiseur d'objet du type. Dans l'exemple suivant, `contacts` est un tableau implicitement typé de types anonymes, dont chacun contient un tableau nommé `PhoneNumbers`. Le mot clé `var` n'est pas utilisé dans les initialiseurs d'objets.

C#

```
var contacts = new[]
{
        new {
            Name = " Eugene Zabokritski",
            PhoneNumbers = new[] { "206-555-0108", "425-555-0001" }
        },
        new {
            Name = " Hanying Feng",
            PhoneNumbers = new[] { "650-555-0199" }
        }
    };
```

Tableaux

Article • 04/09/2023

Vous pouvez stocker plusieurs variables du même type dans une structure de données de type tableau. Vous déclarez un tableau en spécifiant le type de ses éléments. Si vous souhaitez que le tableau stocke des éléments de n'importe quel type, vous pouvez spécifier `object` comme type. Dans le système de type unifié de C#, tous les types (les types référence et valeur, prédéfinis ou définis par l'utilisateur) héritent directement ou indirectement du type Object.

```
C#

type[] arrayName;
```

Un tableau possède les propriétés suivantes :

- Un tableau peut être unidimensionnel, multidimensionnel ou en escalier.
- Le nombre de dimensions est défini lorsqu'une variable de tableau est déclarée. La longueur de chaque dimension est établie lors de la création de l'instance de tableau. Ces valeurs ne peuvent pas être modifiées pendant la durée de vie de l'instance.
- Un tableau en escalier est un tableau de tableaux, et chaque tableau membre a la valeur `null` par défaut.
- Les tableaux sont indexés sur zéro : un tableau avec `n` éléments est indexée de `0` à `n-1`.
- Les éléments de tableau peuvent être de n'importe quel type, y compris un type tableau.
- Les types tableau sont des types référence dérivés du type de base abstrait Array. Tous les tableaux implémentent IList et IEnumerable. Vous pouvez utiliser l'instruction foreach pour itérer au sein d'un tableau. Les tableaux unidimensionnels implémentent aussi IList<T> et IEnumerable<T>.

Les éléments d'un tableau peuvent être initialisés sur des valeurs connues lors de la création du tableau. À compter de C# 12, tous les types de collection peuvent être initialisés à l'aide d'une expression de collection. Les éléments qui ne sont pas initialisés sont définis sur la valeur par défaut. La valeur par défaut est le modèle 0 bits. Tous les types de référence (y compris les types non-nullables) ont les valeurs `null`. Tous les types de valeur ont les modèles 0 bits. Cela signifie que la propriété Nullable<T>.HasValue est `false` et que la propriété Nullable<T>.Value n'est pas définie. Dans l'implémentation .NET, la propriété `Value` lève une exception.

L'exemple suivant crée des tableaux unidimensionnels, multidimensionnels et en escalier :

```
C#
```

```
// Declare a single-dimensional array of 5 integers.
int[] array1 = new int[5];

// Declare and set array element values.
int[] array2 = { 1, 2, 3, 4, 5, 6 };

// Declare a two dimensional array.
int[,] multiDimensionalArray1 = new int[2, 3];

// Declare and set array element values.
int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };

// Declare a jagged array.
int[][] jaggedArray = new int[6][];

// Set the values of the first array in the jagged array structure.
jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
```

Tableaux unidimensionnels

Un *tableau unidimensionnel* est une séquence d'éléments similaires. Vous accédez à un élément via son *index*. L'*index* est sa position ordinale dans la séquence. Le premier élément du tableau est à l'index `0`. Vous créez un tableau unidimensionnel à l'aide du nouvel opérateur en spécifiant le type d'élément du tableau et le nombre d'éléments. L'exemple suivant déclare et initialise des tableaux unidimensionnels :

```
C#
```

```
int[] array = new int[5];
string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };

Console.WriteLine(weekDays[0]);
Console.WriteLine(weekDays[1]);
Console.WriteLine(weekDays[2]);
Console.WriteLine(weekDays[3]);
Console.WriteLine(weekDays[4]);
Console.WriteLine(weekDays[5]);
Console.WriteLine(weekDays[6]);

/*Output:
Sun
Mon
Tue
Wed
```

```
Thu
Fri
Sat
*/
```

La première déclaration déclare un tableau non initialisé de cinq entiers, de `array[0]` à `array[4]`. Les éléments du tableau sont initialisés à la valeur par défaut du type d'élément, `0` pour les entiers. La deuxième déclaration déclare un tableau de chaînes et initialise les sept valeurs de ce tableau. Une instruction foreach itère les éléments du tableau `weekday` et affiche toutes les valeurs. Pour les tableaux unidimensionnels, l'instruction `foreach` traite les éléments dans l'ordre croissant des index, en commençant par l'index 0 et en terminant par l'index `Length - 1`.

Passer des tableaux unidimensionnels en tant qu'arguments

Vous pouvez passer un tableau unidimensionnel initialisé à une méthode. Dans l'exemple suivant, un tableau de chaînes est initialisé et passé en tant qu'argument à une méthode `DisplayArray` pour des chaînes. La méthode affiche les éléments du tableau. Ensuite, la méthode `ChangeArray` inverse les éléments du tableau, puis la méthode `ChangeArrayElements` modifie les trois premiers éléments du tableau. Une fois que chaque méthode est retournée, la méthode `DisplayArray` montre que le passage d'un tableau par valeur n'empêche pas les modifications des éléments du tableau.

C#

```
class ArrayExample
{
    static void DisplayArray(string[] arr) =>
Console.WriteLine(string.Join(" ", arr));

    // Change the array by reversing its elements.
    static void ChangeArray(string[] arr) => Array.Reverse(arr);

    static void ChangeArrayElements(string[] arr)
    {
        // Change the value of the first three array elements.
        arr[0] = "Mon";
        arr[1] = "Wed";
        arr[2] = "Fri";
    }

    static void Main()
    {
        // Declare and initialize an array.
        string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri",
```

```
"Sat" };
        // Display the array elements.
        DisplayArray(weekDays);
        Console.WriteLine();

        // Reverse the array.
        ChangeArray(weekDays);
        // Display the array again to verify that it stays reversed.
        Console.WriteLine("Array weekDays after the call to ChangeArray:");
        DisplayArray(weekDays);
        Console.WriteLine();

        // Assign new values to individual array elements.
        ChangeArrayElements(weekDays);
        // Display the array again to verify that it has changed.
        Console.WriteLine("Array weekDays after the call to
ChangeArrayElements:");
        DisplayArray(weekDays);
    }
}
// The example displays the following output:
//         Sun Mon Tue Wed Thu Fri Sat
//
//        Array weekDays after the call to ChangeArray:
//        Sat Fri Thu Wed Tue Mon Sun
//
//        Array weekDays after the call to ChangeArrayElements:
//        Mon Wed Fri Wed Tue Mon Sun
```

Tableaux multidimensionnels

Les tableaux peuvent avoir plusieurs dimensions. Par exemple, les déclarations suivantes créent quatre tableaux : deux ont deux dimensions, deux ont trois dimensions. Les deux premières déclarations déclarent la longueur de chaque dimension, mais n'initialisent pas les valeurs du tableau. Les deux deuxièmes déclarations utilisent un initialiseur pour définir les valeurs de chaque élément dans le tableau multidimensionnel.

C#

```
int[,] array2DDeclaration = new int[4, 2];

int[,,] array3DDeclaration = new int[4, 2, 3];

// Two-dimensional array.
int[,] array2DInitialization =  { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// Three-dimensional array.
int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,   5,  6 } },
                                { { 7, 8, 9 }, { 10, 11, 12 } } };

// Accessing array elements.
```

```
System.Console.WriteLine(array2DInitialization[0, 0]);
System.Console.WriteLine(array2DInitialization[0, 1]);
System.Console.WriteLine(array2DInitialization[1, 0]);
System.Console.WriteLine(array2DInitialization[1, 1]);

System.Console.WriteLine(array2DInitialization[3, 0]);
System.Console.WriteLine(array2DInitialization[3, 1]);
// Output:
// 1
// 2
// 3
// 4
// 7
// 8

System.Console.WriteLine(array3D[1, 0, 1]);
System.Console.WriteLine(array3D[1, 1, 2]);
// Output:
// 8
// 12

// Getting the total count of elements or the length of a given dimension.
var allLength = array3D.Length;
var total = 1;
for (int i = 0; i < array3D.Rank; i++)
{
    total *= array3D.GetLength(i);
}
System.Console.WriteLine($"{allLength} equals {total}");
// Output:
// 12 equals 12
```

Pour les tableaux multidimensionnels, les éléments sont traités de sorte que les index de la dimension la plus à droite sont incrémentés en premier, puis la dimension immédiatement à gauche, et ainsi de suite jusqu'à l'index le plus à gauche. L'exemple suivant énumère à la fois un tableau 2D et un tableau 3D :

C#

```
int[,] numbers2D = { { 9, 99 }, { 3, 33 }, { 5, 55 } };

foreach (int i in numbers2D)
{
    System.Console.Write($"{i} ");
}
// Output: 9 99 3 33 5 55

int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,   5,  6 } },
                          { { 7, 8, 9 }, { 10, 11, 12 } } };
foreach (int i in array3D)
{
    System.Console.Write($"{i} ");
```

```
}
// Output: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
```

Dans un tableau 2D, vous pouvez considérer l'index de gauche comme la *ligne* et l'index de droite comme la *colonne*.

Cependant, dans le cas de tableaux multidimensionnels, l'utilisation d'une boucle for imbriquée vous permet de mieux contrôler l'ordre dans lequel les éléments du tableau sont traités :

C#

```
int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,   5,  6 } },
                        { { 7, 8, 9 }, { 10, 11, 12 } } };

for (int i = 0; i < array3D.GetLength(0); i++)
{
    for (int j = 0; j < array3D.GetLength(1); j++)
    {
        for (int k = 0; k < array3D.GetLength(2); k++)
        {
            System.Console.Write($"{array3D[i, j, k]} ");
        }
        System.Console.WriteLine();
    }
    System.Console.WriteLine();
}
// Output (including blank lines):
// 1 2 3
// 4 5 6
//
// 7 8 9
// 10 11 12
//
```

Passer des tableaux multidimensionnels en tant qu'arguments

Vous pouvez passer un tableau multidimensionnel initialisé à une méthode de la même manière que vous passez un tableau unidimensionnel. Le code suivant illustre une déclaration partielle d'une méthode Print qui accepte un tableau à deux dimensions en tant qu'argument. Vous pouvez initialiser et passer un nouveau tableau en une seule étape, comme dans l'exemple suivant. Dans l'exemple suivant, un tableau à deux dimensions d'entiers est initialisé et passé à la méthode `Print2DArray`. La méthode affiche les éléments du tableau.

C#

```
static void Print2DArray(int[,] arr)
{
    // Display the array elements.
    for (int i = 0; i < arr.GetLength(0); i++)
    {
        for (int j = 0; j < arr.GetLength(1); j++)
        {
            System.Console.WriteLine("Element({0},{1})={2}", i, j, arr[i,
j]);
        }
    }
}
static void ExampleUsage()
{
    // Pass the array as an argument.
    Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } });
}
/* Output:
    Element(0,0)=1
    Element(0,1)=2
    Element(1,0)=3
    Element(1,1)=4
    Element(2,0)=5
    Element(2,1)=6
    Element(3,0)=7
    Element(3,1)=8
*/
```

Tableaux en escalier

Un tableau en escalier est un tableau dont les éléments sont des tableaux, éventuellement de différentes tailles. Un tableau en escalier est parfois appelé « tableau de tableaux ». Ses éléments sont des types de référence et sont initialisés sur `null`. Les exemples suivants montrent comment déclarer, initialiser et accéder aux tableaux en escalier. Le premier exemple, `jaggedArray`, est déclaré dans une instruction. Chaque tableau contenu est créé dans les instructions suivantes. Le deuxième exemple, `jaggedArray2`, est déclaré et initialisé dans une instruction. Il est possible de combiner des tableaux en escalier et des tableaux multidimensionnels. Le dernier exemple, `jaggedArray3`, déclare et initialise un tableau en escalier unidimensionnel composé de trois éléments de tableau à deux dimensions de tailles différentes.

C#

```
int[][] jaggedArray = new int[3][];
```

```
jaggedArray[0] = new int[] { 1, 3, 5, 7, 9 };
jaggedArray[1] = new int[] { 0, 2, 4, 6 };
jaggedArray[2] = new int[] { 11, 22 };

int[][] jaggedArray2 =
{
    new int[] { 1, 3, 5, 7, 9 },
    new int[] { 0, 2, 4, 6 },
    new int[] { 11, 22 }
};

// Assign 77 to the second element ([1]) of the first array ([0]):
jaggedArray2[0][1] = 77;

// Assign 88 to the second element ([1]) of the third array ([2]):
jaggedArray2[2][1] = 88;

int[][,] jaggedArray3 = new int[3][,]
{
    new int[,] { {1,3}, {5,7} },
    new int[,] { {0,2}, {4,6}, {8,10} },
    new int[,] { {11,22}, {99,88}, {0,9} }
};

Console.Write("{0}", jaggedArray3[0][1, 0]);
Console.WriteLine(jaggedArray3.Length);
```

Les éléments d'un tableau en escalier doivent être initialisés avant de pouvoir les utiliser. Chacun des éléments est lui-même un tableau. Il est également possible d'utiliser des initialiseurs pour remplir les éléments de tableau avec des valeurs. Lorsque vous utilisez des initialiseurs, vous n'avez pas besoin de la taille du tableau.

Cet exemple génère un tableau dont les éléments sont eux-mêmes des tableaux. Chacun des éléments de tableau ont une taille différente.

C#

```
// Declare the array of two elements.
int[][] arr = new int[2][];

// Initialize the elements.
arr[0] = new int[5] { 1, 3, 5, 7, 9 };
arr[1] = new int[4] { 2, 4, 6, 8 };

// Display the array elements.
for (int i = 0; i < arr.Length; i++)
{
    System.Console.Write("Element({0}): ", i);

    for (int j = 0; j < arr[i].Length; j++)
    {
        System.Console.Write("{0}{1}", arr[i][j], j == (arr[i].Length - 1) ?
```

```
"" : " ");
    }
    System.Console.WriteLine();
}
/* Output:
    Element(0): 1 3 5 7 9
    Element(1): 2 4 6 8
*/
```

Tableaux implicitement typés

Vous pouvez créer un tableau implicitement typé dans lequel le type de l'instance de tableau est déduit à partir des éléments spécifiés dans l'initialiseur de tableau. Les règles applicables à une variable implicitement typée s'appliquent aussi aux tableaux implicitement typés. Pour plus d'informations, consultez Variables locales implicitement typées.

Les exemples suivants montrent comment créer un tableau implicitement typé :

C#

```
var a = new[] { 1, 10, 100, 1000 }; // int[]

// Accessing array
Console.WriteLine("First element: " + a[0]);
Console.WriteLine("Second element: " + a[1]);
Console.WriteLine("Third element: " + a[2]);
Console.WriteLine("Fourth element: " + a[3]);
/* Outputs
First element: 1
Second element: 10
Third element: 100
Fourth element: 1000
*/

var b = new[] { "hello", null, "world" }; // string[]

// Accessing elements of an array using 'string.Join' method
Console.WriteLine(string.Join(" ", b));
/* Output
hello  world
*/

// single-dimension jagged array
var c = new[]
{
                                    new[]{1,2,3,4},
                                    new[]{5,6,7,8}
            };
// Looping through the outer array
```

```
for (int k = 0; k < c.Length; k++)
{
    // Looping through each inner array
    for (int j = 0; j < c[k].Length; j++)
    {
        // Accessing each element and printing it to the console
        Console.WriteLine($"Element at c[{k}][{j}] is: {c[k][j]}");
    }
}
/* Outputs
Element at c[0][0] is: 1
Element at c[0][1] is: 2
Element at c[0][2] is: 3
Element at c[0][3] is: 4
Element at c[1][0] is: 5
Element at c[1][1] is: 6
Element at c[1][2] is: 7
Element at c[1][3] is: 8
*/

// jagged array of strings
var d = new[]
{
                new[]{"Luca", "Mads", "Luke", "Dinesh"},
                new[]{"Karen", "Suma", "Frances"}
            };

// Looping through the outer array
int i = 0;
foreach (var subArray in d)
{
    // Looping through each inner array
    int j = 0;
    foreach (var element in subArray)
    {
        // Accessing each element and printing it to the console
        Console.WriteLine($"Element at d[{i}][{j}] is: {element}");
        j++;
    }
    i++;
}
/* Outputs
Element at d[0][0] is: Luca
Element at d[0][1] is: Mads
Element at d[0][2] is: Luke
Element at d[0][3] is: Dinesh
Element at d[1][0] is: Karen
Element at d[1][1] is: Suma
Element at d[1][2] is: Frances
*/
```

Dans l'exemple précédent, vous pouvez constater que dans le cas des tableaux implicitement typés, aucun crochet n'est utilisé à gauche de l'instruction d'initialisation.

De plus, les tableaux en escalier sont initialisés à l'aide de `new []` à l'instar des tableaux unidimensionnels.

Quand vous créez un type anonyme qui contient un tableau, le tableau doit être implicitement typé dans l'initialiseur d'objet du type. Dans l'exemple suivant, `contacts` est un tableau implicitement typé de types anonymes, dont chacun contient un tableau nommé `PhoneNumbers`. Le mot clé `var` n'est pas utilisé dans les initialiseurs d'objets.

C#

```
var contacts = new[]
{
        new {
            Name = " Eugene Zabokritski",
            PhoneNumbers = new[] { "206-555-0108", "425-555-0001" }
        },
        new {
            Name = " Hanying Feng",
            PhoneNumbers = new[] { "650-555-0199" }
        }
    };
```

Tableaux

Article • 04/09/2023

Vous pouvez stocker plusieurs variables du même type dans une structure de données de type tableau. Vous déclarez un tableau en spécifiant le type de ses éléments. Si vous souhaitez que le tableau stocke des éléments de n'importe quel type, vous pouvez spécifier `object` comme type. Dans le système de type unifié de C#, tous les types (les types référence et valeur, prédéfinis ou définis par l'utilisateur) héritent directement ou indirectement du type Object.

```
C#

type[] arrayName;
```

Un tableau possède les propriétés suivantes :

- Un tableau peut être unidimensionnel, multidimensionnel ou en escalier.
- Le nombre de dimensions est défini lorsqu'une variable de tableau est déclarée. La longueur de chaque dimension est établie lors de la création de l'instance de tableau. Ces valeurs ne peuvent pas être modifiées pendant la durée de vie de l'instance.
- Un tableau en escalier est un tableau de tableaux, et chaque tableau membre a la valeur `null` par défaut.
- Les tableaux sont indexés sur zéro : un tableau avec `n` éléments est indexée de `0` à `n-1`.
- Les éléments de tableau peuvent être de n'importe quel type, y compris un type tableau.
- Les types tableau sont des types référence dérivés du type de base abstrait Array. Tous les tableaux implémentent IList et IEnumerable. Vous pouvez utiliser l'instruction foreach pour itérer au sein d'un tableau. Les tableaux unidimensionnels implémentent aussi IList<T> et IEnumerable<T>.

Les éléments d'un tableau peuvent être initialisés sur des valeurs connues lors de la création du tableau. À compter de C# 12, tous les types de collection peuvent être initialisés à l'aide d'une expression de collection. Les éléments qui ne sont pas initialisés sont définis sur la valeur par défaut. La valeur par défaut est le modèle 0 bits. Tous les types de référence (y compris les types non-nullables) ont les valeurs `null`. Tous les types de valeur ont les modèles 0 bits. Cela signifie que la propriété Nullable<T>.HasValue est `false` et que la propriété Nullable<T>.Value n'est pas définie. Dans l'implémentation .NET, la propriété `Value` lève une exception.

L'exemple suivant crée des tableaux unidimensionnels, multidimensionnels et en escalier :

```
C#
```

```
// Declare a single-dimensional array of 5 integers.
int[] array1 = new int[5];

// Declare and set array element values.
int[] array2 = { 1, 2, 3, 4, 5, 6 };

// Declare a two dimensional array.
int[,] multiDimensionalArray1 = new int[2, 3];

// Declare and set array element values.
int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };

// Declare a jagged array.
int[][] jaggedArray = new int[6][];

// Set the values of the first array in the jagged array structure.
jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
```

Tableaux unidimensionnels

Un *tableau unidimensionnel* est une séquence d'éléments similaires. Vous accédez à un élément via son *index*. L'*index* est sa position ordinale dans la séquence. Le premier élément du tableau est à l'index `0`. Vous créez un tableau unidimensionnel à l'aide du nouvel opérateur en spécifiant le type d'élément du tableau et le nombre d'éléments. L'exemple suivant déclare et initialise des tableaux unidimensionnels :

```
C#
```

```
int[] array = new int[5];
string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };

Console.WriteLine(weekDays[0]);
Console.WriteLine(weekDays[1]);
Console.WriteLine(weekDays[2]);
Console.WriteLine(weekDays[3]);
Console.WriteLine(weekDays[4]);
Console.WriteLine(weekDays[5]);
Console.WriteLine(weekDays[6]);

/*Output:
Sun
Mon
Tue
Wed
```

```
Thu
Fri
Sat
*/
```

La première déclaration déclare un tableau non initialisé de cinq entiers, de `array[0]` à `array[4]`. Les éléments du tableau sont initialisés à la valeur par défaut du type d'élément, `0` pour les entiers. La deuxième déclaration déclare un tableau de chaînes et initialise les sept valeurs de ce tableau. Une instruction foreach itère les éléments du tableau `weekday` et affiche toutes les valeurs. Pour les tableaux unidimensionnels, l'instruction `foreach` traite les éléments dans l'ordre croissant des index, en commençant par l'index 0 et en terminant par l'index `Length - 1`.

Passer des tableaux unidimensionnels en tant qu'arguments

Vous pouvez passer un tableau unidimensionnel initialisé à une méthode. Dans l'exemple suivant, un tableau de chaînes est initialisé et passé en tant qu'argument à une méthode `DisplayArray` pour des chaînes. La méthode affiche les éléments du tableau. Ensuite, la méthode `ChangeArray` inverse les éléments du tableau, puis la méthode `ChangeArrayElements` modifie les trois premiers éléments du tableau. Une fois que chaque méthode est retournée, la méthode `DisplayArray` montre que le passage d'un tableau par valeur n'empêche pas les modifications des éléments du tableau.

C#

```
class ArrayExample
{
    static void DisplayArray(string[] arr) =>
Console.WriteLine(string.Join(" ", arr));

    // Change the array by reversing its elements.
    static void ChangeArray(string[] arr) => Array.Reverse(arr);

    static void ChangeArrayElements(string[] arr)
    {
        // Change the value of the first three array elements.
        arr[0] = "Mon";
        arr[1] = "Wed";
        arr[2] = "Fri";
    }

    static void Main()
    {
        // Declare and initialize an array.
        string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri",
```

```
"Sat" };
        // Display the array elements.
        DisplayArray(weekDays);
        Console.WriteLine();

        // Reverse the array.
        ChangeArray(weekDays);
        // Display the array again to verify that it stays reversed.
        Console.WriteLine("Array weekDays after the call to ChangeArray:");
        DisplayArray(weekDays);
        Console.WriteLine();

        // Assign new values to individual array elements.
        ChangeArrayElements(weekDays);
        // Display the array again to verify that it has changed.
        Console.WriteLine("Array weekDays after the call to
ChangeArrayElements:");
        DisplayArray(weekDays);
    }
}
// The example displays the following output:
//         Sun Mon Tue Wed Thu Fri Sat
//
//        Array weekDays after the call to ChangeArray:
//        Sat Fri Thu Wed Tue Mon Sun
//
//        Array weekDays after the call to ChangeArrayElements:
//        Mon Wed Fri Wed Tue Mon Sun
```

Tableaux multidimensionnels

Les tableaux peuvent avoir plusieurs dimensions. Par exemple, les déclarations suivantes créent quatre tableaux : deux ont deux dimensions, deux ont trois dimensions. Les deux premières déclarations déclarent la longueur de chaque dimension, mais n'initialisent pas les valeurs du tableau. Les deux deuxièmes déclarations utilisent un initialiseur pour définir les valeurs de chaque élément dans le tableau multidimensionnel.

C#

```
int[,] array2DDeclaration = new int[4, 2];

int[,,] array3DDeclaration = new int[4, 2, 3];

// Two-dimensional array.
int[,] array2DInitialization =  { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// Three-dimensional array.
int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,   5,  6 } },
                                { { 7, 8, 9 }, { 10, 11, 12 } } };

// Accessing array elements.
```

```
System.Console.WriteLine(array2DInitialization[0, 0]);
System.Console.WriteLine(array2DInitialization[0, 1]);
System.Console.WriteLine(array2DInitialization[1, 0]);
System.Console.WriteLine(array2DInitialization[1, 1]);

System.Console.WriteLine(array2DInitialization[3, 0]);
System.Console.WriteLine(array2DInitialization[3, 1]);
// Output:
// 1
// 2
// 3
// 4
// 7
// 8

System.Console.WriteLine(array3D[1, 0, 1]);
System.Console.WriteLine(array3D[1, 1, 2]);
// Output:
// 8
// 12

// Getting the total count of elements or the length of a given dimension.
var allLength = array3D.Length;
var total = 1;
for (int i = 0; i < array3D.Rank; i++)
{
    total *= array3D.GetLength(i);
}
System.Console.WriteLine($"{allLength} equals {total}");
// Output:
// 12 equals 12
```

Pour les tableaux multidimensionnels, les éléments sont traités de sorte que les index de la dimension la plus à droite sont incrémentés en premier, puis la dimension immédiatement à gauche, et ainsi de suite jusqu'à l'index le plus à gauche. L'exemple suivant énumère à la fois un tableau 2D et un tableau 3D :

C#

```
int[,] numbers2D = { { 9, 99 }, { 3, 33 }, { 5, 55 } };

foreach (int i in numbers2D)
{
    System.Console.Write($"{i} ");
}
// Output: 9 99 3 33 5 55

int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,   5,  6 } },
                          { { 7, 8, 9 }, { 10, 11, 12 } } };
foreach (int i in array3D)
{
    System.Console.Write($"{i} ");
```

```
}
// Output: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
```

Dans un tableau 2D, vous pouvez considérer l'index de gauche comme la *ligne* et l'index de droite comme la *colonne*.

Cependant, dans le cas de tableaux multidimensionnels, l'utilisation d'une boucle for imbriquée vous permet de mieux contrôler l'ordre dans lequel les éléments du tableau sont traités :

C#

```
int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,   5,  6 } },
                        { { 7, 8, 9 }, { 10, 11, 12 } } };

for (int i = 0; i < array3D.GetLength(0); i++)
{
    for (int j = 0; j < array3D.GetLength(1); j++)
    {
        for (int k = 0; k < array3D.GetLength(2); k++)
        {
            System.Console.Write($"{array3D[i, j, k]} ");
        }
        System.Console.WriteLine();
    }
    System.Console.WriteLine();
}
// Output (including blank lines):
// 1 2 3
// 4 5 6
//
// 7 8 9
// 10 11 12
//
```

Passer des tableaux multidimensionnels en tant qu'arguments

Vous pouvez passer un tableau multidimensionnel initialisé à une méthode de la même manière que vous passez un tableau unidimensionnel. Le code suivant illustre une déclaration partielle d'une méthode Print qui accepte un tableau à deux dimensions en tant qu'argument. Vous pouvez initialiser et passer un nouveau tableau en une seule étape, comme dans l'exemple suivant. Dans l'exemple suivant, un tableau à deux dimensions d'entiers est initialisé et passé à la méthode `Print2DArray`. La méthode affiche les éléments du tableau.

C#

```
static void Print2DArray(int[,] arr)
{
    // Display the array elements.
    for (int i = 0; i < arr.GetLength(0); i++)
    {
        for (int j = 0; j < arr.GetLength(1); j++)
        {
            System.Console.WriteLine("Element({0},{1})={2}", i, j, arr[i,
j]);
        }
    }
}
static void ExampleUsage()
{
    // Pass the array as an argument.
    Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } });
}
/* Output:
    Element(0,0)=1
    Element(0,1)=2
    Element(1,0)=3
    Element(1,1)=4
    Element(2,0)=5
    Element(2,1)=6
    Element(3,0)=7
    Element(3,1)=8
*/
```

Tableaux en escalier

Un tableau en escalier est un tableau dont les éléments sont des tableaux, éventuellement de différentes tailles. Un tableau en escalier est parfois appelé « tableau de tableaux ». Ses éléments sont des types de référence et sont initialisés sur `null`. Les exemples suivants montrent comment déclarer, initialiser et accéder aux tableaux en escalier. Le premier exemple, `jaggedArray`, est déclaré dans une instruction. Chaque tableau contenu est créé dans les instructions suivantes. Le deuxième exemple, `jaggedArray2`, est déclaré et initialisé dans une instruction. Il est possible de combiner des tableaux en escalier et des tableaux multidimensionnels. Le dernier exemple, `jaggedArray3`, déclare et initialise un tableau en escalier unidimensionnel composé de trois éléments de tableau à deux dimensions de tailles différentes.

C#

```
int[][] jaggedArray = new int[3][];
```

```
jaggedArray[0] = new int[] { 1, 3, 5, 7, 9 };
jaggedArray[1] = new int[] { 0, 2, 4, 6 };
jaggedArray[2] = new int[] { 11, 22 };

int[][] jaggedArray2 =
{
    new int[] { 1, 3, 5, 7, 9 },
    new int[] { 0, 2, 4, 6 },
    new int[] { 11, 22 }
};

// Assign 77 to the second element ([1]) of the first array ([0]):
jaggedArray2[0][1] = 77;

// Assign 88 to the second element ([1]) of the third array ([2]):
jaggedArray2[2][1] = 88;

int[][,] jaggedArray3 = new int[3][,]
{
    new int[,] { {1,3}, {5,7} },
    new int[,] { {0,2}, {4,6}, {8,10} },
    new int[,] { {11,22}, {99,88}, {0,9} }
};

Console.Write("{0}", jaggedArray3[0][1, 0]);
Console.WriteLine(jaggedArray3.Length);
```

Les éléments d'un tableau en escalier doivent être initialisés avant de pouvoir les utiliser. Chacun des éléments est lui-même un tableau. Il est également possible d'utiliser des initialiseurs pour remplir les éléments de tableau avec des valeurs. Lorsque vous utilisez des initialiseurs, vous n'avez pas besoin de la taille du tableau.

Cet exemple génère un tableau dont les éléments sont eux-mêmes des tableaux. Chacun des éléments de tableau ont une taille différente.

C#

```
// Declare the array of two elements.
int[][] arr = new int[2][];

// Initialize the elements.
arr[0] = new int[5] { 1, 3, 5, 7, 9 };
arr[1] = new int[4] { 2, 4, 6, 8 };

// Display the array elements.
for (int i = 0; i < arr.Length; i++)
{
    System.Console.Write("Element({0}): ", i);

    for (int j = 0; j < arr[i].Length; j++)
    {
        System.Console.Write("{0}{1}", arr[i][j], j == (arr[i].Length - 1) ?
```

```
"" : " ");
    }
    System.Console.WriteLine();
}
/* Output:
    Element(0): 1 3 5 7 9
    Element(1): 2 4 6 8
*/
```

Tableaux implicitement typés

Vous pouvez créer un tableau implicitement typé dans lequel le type de l'instance de tableau est déduit à partir des éléments spécifiés dans l'initialiseur de tableau. Les règles applicables à une variable implicitement typée s'appliquent aussi aux tableaux implicitement typés. Pour plus d'informations, consultez Variables locales implicitement typées.

Les exemples suivants montrent comment créer un tableau implicitement typé :

C#

```
var a = new[] { 1, 10, 100, 1000 }; // int[]

// Accessing array
Console.WriteLine("First element: " + a[0]);
Console.WriteLine("Second element: " + a[1]);
Console.WriteLine("Third element: " + a[2]);
Console.WriteLine("Fourth element: " + a[3]);
/* Outputs
First element: 1
Second element: 10
Third element: 100
Fourth element: 1000
*/

var b = new[] { "hello", null, "world" }; // string[]

// Accessing elements of an array using 'string.Join' method
Console.WriteLine(string.Join(" ", b));
/* Output
hello  world
*/

// single-dimension jagged array
var c = new[]
{
                                    new[]{1,2,3,4},
                                    new[]{5,6,7,8}
            };
// Looping through the outer array
```

```
for (int k = 0; k < c.Length; k++)
{
    // Looping through each inner array
    for (int j = 0; j < c[k].Length; j++)
    {
        // Accessing each element and printing it to the console
        Console.WriteLine($"Element at c[{k}][{j}] is: {c[k][j]}");
    }
}
/* Outputs
Element at c[0][0] is: 1
Element at c[0][1] is: 2
Element at c[0][2] is: 3
Element at c[0][3] is: 4
Element at c[1][0] is: 5
Element at c[1][1] is: 6
Element at c[1][2] is: 7
Element at c[1][3] is: 8
*/

// jagged array of strings
var d = new[]
{
                new[]{"Luca", "Mads", "Luke", "Dinesh"},
                new[]{"Karen", "Suma", "Frances"}
            };

// Looping through the outer array
int i = 0;
foreach (var subArray in d)
{
    // Looping through each inner array
    int j = 0;
    foreach (var element in subArray)
    {
        // Accessing each element and printing it to the console
        Console.WriteLine($"Element at d[{i}][{j}] is: {element}");
        j++;
    }
    i++;
}
/* Outputs
Element at d[0][0] is: Luca
Element at d[0][1] is: Mads
Element at d[0][2] is: Luke
Element at d[0][3] is: Dinesh
Element at d[1][0] is: Karen
Element at d[1][1] is: Suma
Element at d[1][2] is: Frances
*/
```

Dans l'exemple précédent, vous pouvez constater que dans le cas des tableaux implicitement typés, aucun crochet n'est utilisé à gauche de l'instruction d'initialisation.

De plus, les tableaux en escalier sont initialisés à l'aide de `new []` à l'instar des tableaux unidimensionnels.

Quand vous créez un type anonyme qui contient un tableau, le tableau doit être implicitement typé dans l'initialiseur d'objet du type. Dans l'exemple suivant, `contacts` est un tableau implicitement typé de types anonymes, dont chacun contient un tableau nommé `PhoneNumbers`. Le mot clé `var` n'est pas utilisé dans les initialiseurs d'objets.

C#

```
var contacts = new[]
{
        new {
            Name = " Eugene Zabokritski",
            PhoneNumbers = new[] { "206-555-0108", "425-555-0001" }
        },
        new {
            Name = " Hanying Feng",
            PhoneNumbers = new[] { "650-555-0199" }
        }
    };
```

Tableaux

Article • 04/09/2023

Vous pouvez stocker plusieurs variables du même type dans une structure de données de type tableau. Vous déclarez un tableau en spécifiant le type de ses éléments. Si vous souhaitez que le tableau stocke des éléments de n'importe quel type, vous pouvez spécifier `object` comme type. Dans le système de type unifié de C#, tous les types (les types référence et valeur, prédéfinis ou définis par l'utilisateur) héritent directement ou indirectement du type Object.

```
C#

type[] arrayName;
```

Un tableau possède les propriétés suivantes :

- Un tableau peut être unidimensionnel, multidimensionnel ou en escalier.
- Le nombre de dimensions est défini lorsqu'une variable de tableau est déclarée. La longueur de chaque dimension est établie lors de la création de l'instance de tableau. Ces valeurs ne peuvent pas être modifiées pendant la durée de vie de l'instance.
- Un tableau en escalier est un tableau de tableaux, et chaque tableau membre a la valeur `null` par défaut.
- Les tableaux sont indexés sur zéro : un tableau avec `n` éléments est indexée de `0` à `n-1`.
- Les éléments de tableau peuvent être de n'importe quel type, y compris un type tableau.
- Les types tableau sont des types référence dérivés du type de base abstrait Array. Tous les tableaux implémentent IList et IEnumerable. Vous pouvez utiliser l'instruction foreach pour itérer au sein d'un tableau. Les tableaux unidimensionnels implémentent aussi IList<T> et IEnumerable<T>.

Les éléments d'un tableau peuvent être initialisés sur des valeurs connues lors de la création du tableau. À compter de C# 12, tous les types de collection peuvent être initialisés à l'aide d'une expression de collection. Les éléments qui ne sont pas initialisés sont définis sur la valeur par défaut. La valeur par défaut est le modèle 0 bits. Tous les types de référence (y compris les types non-nullables) ont les valeurs `null`. Tous les types de valeur ont les modèles 0 bits. Cela signifie que la propriété Nullable<T>.HasValue est `false` et que la propriété Nullable<T>.Value n'est pas définie. Dans l'implémentation .NET, la propriété `Value` lève une exception.

L'exemple suivant crée des tableaux unidimensionnels, multidimensionnels et en escalier :

```
C#
```

```
// Declare a single-dimensional array of 5 integers.
int[] array1 = new int[5];

// Declare and set array element values.
int[] array2 = { 1, 2, 3, 4, 5, 6 };

// Declare a two dimensional array.
int[,] multiDimensionalArray1 = new int[2, 3];

// Declare and set array element values.
int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };

// Declare a jagged array.
int[][] jaggedArray = new int[6][];

// Set the values of the first array in the jagged array structure.
jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
```

Tableaux unidimensionnels

Un *tableau unidimensionnel* est une séquence d'éléments similaires. Vous accédez à un élément via son *index*. L'*index* est sa position ordinale dans la séquence. Le premier élément du tableau est à l'index `0`. Vous créez un tableau unidimensionnel à l'aide du nouvel opérateur en spécifiant le type d'élément du tableau et le nombre d'éléments. L'exemple suivant déclare et initialise des tableaux unidimensionnels :

```
C#
```

```
int[] array = new int[5];
string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };

Console.WriteLine(weekDays[0]);
Console.WriteLine(weekDays[1]);
Console.WriteLine(weekDays[2]);
Console.WriteLine(weekDays[3]);
Console.WriteLine(weekDays[4]);
Console.WriteLine(weekDays[5]);
Console.WriteLine(weekDays[6]);

/*Output:
Sun
Mon
Tue
Wed
```

```
Thu
Fri
Sat
*/
```

La première déclaration déclare un tableau non initialisé de cinq entiers, de `array[0]` à `array[4]`. Les éléments du tableau sont initialisés à la valeur par défaut du type d'élément, `0` pour les entiers. La deuxième déclaration déclare un tableau de chaînes et initialise les sept valeurs de ce tableau. Une instruction foreach itère les éléments du tableau `weekday` et affiche toutes les valeurs. Pour les tableaux unidimensionnels, l'instruction `foreach` traite les éléments dans l'ordre croissant des index, en commençant par l'index 0 et en terminant par l'index `Length - 1`.

Passer des tableaux unidimensionnels en tant qu'arguments

Vous pouvez passer un tableau unidimensionnel initialisé à une méthode. Dans l'exemple suivant, un tableau de chaînes est initialisé et passé en tant qu'argument à une méthode `DisplayArray` pour des chaînes. La méthode affiche les éléments du tableau. Ensuite, la méthode `ChangeArray` inverse les éléments du tableau, puis la méthode `ChangeArrayElements` modifie les trois premiers éléments du tableau. Une fois que chaque méthode est retournée, la méthode `DisplayArray` montre que le passage d'un tableau par valeur n'empêche pas les modifications des éléments du tableau.

C#

```
class ArrayExample
{
    static void DisplayArray(string[] arr) =>
Console.WriteLine(string.Join(" ", arr));

    // Change the array by reversing its elements.
    static void ChangeArray(string[] arr) => Array.Reverse(arr);

    static void ChangeArrayElements(string[] arr)
    {
        // Change the value of the first three array elements.
        arr[0] = "Mon";
        arr[1] = "Wed";
        arr[2] = "Fri";
    }

    static void Main()
    {
        // Declare and initialize an array.
        string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri",
```

```
"Sat" };
        // Display the array elements.
        DisplayArray(weekDays);
        Console.WriteLine();

        // Reverse the array.
        ChangeArray(weekDays);
        // Display the array again to verify that it stays reversed.
        Console.WriteLine("Array weekDays after the call to ChangeArray:");
        DisplayArray(weekDays);
        Console.WriteLine();

        // Assign new values to individual array elements.
        ChangeArrayElements(weekDays);
        // Display the array again to verify that it has changed.
        Console.WriteLine("Array weekDays after the call to
ChangeArrayElements:");
        DisplayArray(weekDays);
    }
}
// The example displays the following output:
//         Sun Mon Tue Wed Thu Fri Sat
//
//        Array weekDays after the call to ChangeArray:
//        Sat Fri Thu Wed Tue Mon Sun
//
//        Array weekDays after the call to ChangeArrayElements:
//        Mon Wed Fri Wed Tue Mon Sun
```

Tableaux multidimensionnels

Les tableaux peuvent avoir plusieurs dimensions. Par exemple, les déclarations suivantes créent quatre tableaux : deux ont deux dimensions, deux ont trois dimensions. Les deux premières déclarations déclarent la longueur de chaque dimension, mais n'initialisent pas les valeurs du tableau. Les deux deuxièmes déclarations utilisent un initialiseur pour définir les valeurs de chaque élément dans le tableau multidimensionnel.

C#

```
int[,] array2DDeclaration = new int[4, 2];

int[,,] array3DDeclaration = new int[4, 2, 3];

// Two-dimensional array.
int[,] array2DInitialization =  { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// Three-dimensional array.
int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,   5,  6 } },
                                { { 7, 8, 9 }, { 10, 11, 12 } } };

// Accessing array elements.
```

```
System.Console.WriteLine(array2DInitialization[0, 0]);
System.Console.WriteLine(array2DInitialization[0, 1]);
System.Console.WriteLine(array2DInitialization[1, 0]);
System.Console.WriteLine(array2DInitialization[1, 1]);

System.Console.WriteLine(array2DInitialization[3, 0]);
System.Console.WriteLine(array2DInitialization[3, 1]);
// Output:
// 1
// 2
// 3
// 4
// 7
// 8

System.Console.WriteLine(array3D[1, 0, 1]);
System.Console.WriteLine(array3D[1, 1, 2]);
// Output:
// 8
// 12

// Getting the total count of elements or the length of a given dimension.
var allLength = array3D.Length;
var total = 1;
for (int i = 0; i < array3D.Rank; i++)
{
    total *= array3D.GetLength(i);
}
System.Console.WriteLine($"{allLength} equals {total}");
// Output:
// 12 equals 12
```

Pour les tableaux multidimensionnels, les éléments sont traités de sorte que les index de la dimension la plus à droite sont incrémentés en premier, puis la dimension immédiatement à gauche, et ainsi de suite jusqu'à l'index le plus à gauche. L'exemple suivant énumère à la fois un tableau 2D et un tableau 3D :

C#

```
int[,] numbers2D = { { 9, 99 }, { 3, 33 }, { 5, 55 } };

foreach (int i in numbers2D)
{
    System.Console.Write($"{i} ");
}
// Output: 9 99 3 33 5 55

int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,   5,  6 } },
                          { { 7, 8, 9 }, { 10, 11, 12 } } };
foreach (int i in array3D)
{
    System.Console.Write($"{i} ");
```

```
}
// Output: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
```

Dans un tableau 2D, vous pouvez considérer l'index de gauche comme la *ligne* et l'index de droite comme la *colonne*.

Cependant, dans le cas de tableaux multidimensionnels, l'utilisation d'une boucle for imbriquée vous permet de mieux contrôler l'ordre dans lequel les éléments du tableau sont traités :

C#

```
int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,   5,  6 } },
                        { { 7, 8, 9 }, { 10, 11, 12 } } };

for (int i = 0; i < array3D.GetLength(0); i++)
{
    for (int j = 0; j < array3D.GetLength(1); j++)
    {
        for (int k = 0; k < array3D.GetLength(2); k++)
        {
            System.Console.Write($"{array3D[i, j, k]} ");
        }
        System.Console.WriteLine();
    }
    System.Console.WriteLine();
}
// Output (including blank lines):
// 1 2 3
// 4 5 6
//
// 7 8 9
// 10 11 12
//
```

Passer des tableaux multidimensionnels en tant qu'arguments

Vous pouvez passer un tableau multidimensionnel initialisé à une méthode de la même manière que vous passez un tableau unidimensionnel. Le code suivant illustre une déclaration partielle d'une méthode Print qui accepte un tableau à deux dimensions en tant qu'argument. Vous pouvez initialiser et passer un nouveau tableau en une seule étape, comme dans l'exemple suivant. Dans l'exemple suivant, un tableau à deux dimensions d'entiers est initialisé et passé à la méthode `Print2DArray`. La méthode affiche les éléments du tableau.

C#

```
static void Print2DArray(int[,] arr)
{
    // Display the array elements.
    for (int i = 0; i < arr.GetLength(0); i++)
    {
        for (int j = 0; j < arr.GetLength(1); j++)
        {
            System.Console.WriteLine("Element({0},{1})={2}", i, j, arr[i,
j]);
        }
    }
}
static void ExampleUsage()
{
    // Pass the array as an argument.
    Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } });
}
/* Output:
    Element(0,0)=1
    Element(0,1)=2
    Element(1,0)=3
    Element(1,1)=4
    Element(2,0)=5
    Element(2,1)=6
    Element(3,0)=7
    Element(3,1)=8
*/
```

Tableaux en escalier

Un tableau en escalier est un tableau dont les éléments sont des tableaux, éventuellement de différentes tailles. Un tableau en escalier est parfois appelé « tableau de tableaux ». Ses éléments sont des types de référence et sont initialisés sur `null`. Les exemples suivants montrent comment déclarer, initialiser et accéder aux tableaux en escalier. Le premier exemple, `jaggedArray`, est déclaré dans une instruction. Chaque tableau contenu est créé dans les instructions suivantes. Le deuxième exemple, `jaggedArray2`, est déclaré et initialisé dans une instruction. Il est possible de combiner des tableaux en escalier et des tableaux multidimensionnels. Le dernier exemple, `jaggedArray3`, déclare et initialise un tableau en escalier unidimensionnel composé de trois éléments de tableau à deux dimensions de tailles différentes.

C#

```
int[][] jaggedArray = new int[3][];
```

```
jaggedArray[0] = new int[] { 1, 3, 5, 7, 9 };
jaggedArray[1] = new int[] { 0, 2, 4, 6 };
jaggedArray[2] = new int[] { 11, 22 };

int[][] jaggedArray2 =
{
    new int[] { 1, 3, 5, 7, 9 },
    new int[] { 0, 2, 4, 6 },
    new int[] { 11, 22 }
};

// Assign 77 to the second element ([1]) of the first array ([0]):
jaggedArray2[0][1] = 77;

// Assign 88 to the second element ([1]) of the third array ([2]):
jaggedArray2[2][1] = 88;

int[][,] jaggedArray3 = new int[3][,]
{
    new int[,] { {1,3}, {5,7} },
    new int[,] { {0,2}, {4,6}, {8,10} },
    new int[,] { {11,22}, {99,88}, {0,9} }
};

Console.Write("{0}", jaggedArray3[0][1, 0]);
Console.WriteLine(jaggedArray3.Length);
```

Les éléments d'un tableau en escalier doivent être initialisés avant de pouvoir les utiliser. Chacun des éléments est lui-même un tableau. Il est également possible d'utiliser des initialiseurs pour remplir les éléments de tableau avec des valeurs. Lorsque vous utilisez des initialiseurs, vous n'avez pas besoin de la taille du tableau.

Cet exemple génère un tableau dont les éléments sont eux-mêmes des tableaux. Chacun des éléments de tableau ont une taille différente.

C#

```
// Declare the array of two elements.
int[][] arr = new int[2][];

// Initialize the elements.
arr[0] = new int[5] { 1, 3, 5, 7, 9 };
arr[1] = new int[4] { 2, 4, 6, 8 };

// Display the array elements.
for (int i = 0; i < arr.Length; i++)
{
    System.Console.Write("Element({0}): ", i);

    for (int j = 0; j < arr[i].Length; j++)
    {
        System.Console.Write("{0}{1}", arr[i][j], j == (arr[i].Length - 1) ?
```

```
"" : " ");
    }
    System.Console.WriteLine();
}
/* Output:
    Element(0): 1 3 5 7 9
    Element(1): 2 4 6 8
*/
```

Tableaux implicitement typés

Vous pouvez créer un tableau implicitement typé dans lequel le type de l'instance de tableau est déduit à partir des éléments spécifiés dans l'initialiseur de tableau. Les règles applicables à une variable implicitement typée s'appliquent aussi aux tableaux implicitement typés. Pour plus d'informations, consultez Variables locales implicitement typées.

Les exemples suivants montrent comment créer un tableau implicitement typé :

C#

```
var a = new[] { 1, 10, 100, 1000 }; // int[]

// Accessing array
Console.WriteLine("First element: " + a[0]);
Console.WriteLine("Second element: " + a[1]);
Console.WriteLine("Third element: " + a[2]);
Console.WriteLine("Fourth element: " + a[3]);
/* Outputs
First element: 1
Second element: 10
Third element: 100
Fourth element: 1000
*/

var b = new[] { "hello", null, "world" }; // string[]

// Accessing elements of an array using 'string.Join' method
Console.WriteLine(string.Join(" ", b));
/* Output
hello  world
*/

// single-dimension jagged array
var c = new[]
{
                                    new[]{1,2,3,4},
                                    new[]{5,6,7,8}
            };
// Looping through the outer array
```

```
for (int k = 0; k < c.Length; k++)
{
    // Looping through each inner array
    for (int j = 0; j < c[k].Length; j++)
    {
        // Accessing each element and printing it to the console
        Console.WriteLine($"Element at c[{k}][{j}] is: {c[k][j]}");
    }
}
/* Outputs
Element at c[0][0] is: 1
Element at c[0][1] is: 2
Element at c[0][2] is: 3
Element at c[0][3] is: 4
Element at c[1][0] is: 5
Element at c[1][1] is: 6
Element at c[1][2] is: 7
Element at c[1][3] is: 8
*/

// jagged array of strings
var d = new[]
{
                new[]{"Luca", "Mads", "Luke", "Dinesh"},
                new[]{"Karen", "Suma", "Frances"}
            };

// Looping through the outer array
int i = 0;
foreach (var subArray in d)
{
    // Looping through each inner array
    int j = 0;
    foreach (var element in subArray)
    {
        // Accessing each element and printing it to the console
        Console.WriteLine($"Element at d[{i}][{j}] is: {element}");
        j++;
    }
    i++;
}
/* Outputs
Element at d[0][0] is: Luca
Element at d[0][1] is: Mads
Element at d[0][2] is: Luke
Element at d[0][3] is: Dinesh
Element at d[1][0] is: Karen
Element at d[1][1] is: Suma
Element at d[1][2] is: Frances
*/
```

Dans l'exemple précédent, vous pouvez constater que dans le cas des tableaux implicitement typés, aucun crochet n'est utilisé à gauche de l'instruction d'initialisation.

De plus, les tableaux en escalier sont initialisés à l'aide de `new []` à l'instar des tableaux unidimensionnels.

Quand vous créez un type anonyme qui contient un tableau, le tableau doit être implicitement typé dans l'initialiseur d'objet du type. Dans l'exemple suivant, `contacts` est un tableau implicitement typé de types anonymes, dont chacun contient un tableau nommé `PhoneNumbers`. Le mot clé `var` n'est pas utilisé dans les initialiseurs d'objets.

C#

```
var contacts = new[]
{
        new {
            Name = " Eugene Zabokritski",
            PhoneNumbers = new[] { "206-555-0108", "425-555-0001" }
        },
        new {
            Name = " Hanying Feng",
            PhoneNumbers = new[] { "650-555-0199" }
        }
    };
```

Tableaux

Article • 04/09/2023

Vous pouvez stocker plusieurs variables du même type dans une structure de données de type tableau. Vous déclarez un tableau en spécifiant le type de ses éléments. Si vous souhaitez que le tableau stocke des éléments de n'importe quel type, vous pouvez spécifier `object` comme type. Dans le système de type unifié de C#, tous les types (les types référence et valeur, prédéfinis ou définis par l'utilisateur) héritent directement ou indirectement du type Object.

```
C#

type[] arrayName;
```

Un tableau possède les propriétés suivantes :

- Un tableau peut être unidimensionnel, multidimensionnel ou en escalier.
- Le nombre de dimensions est défini lorsqu'une variable de tableau est déclarée. La longueur de chaque dimension est établie lors de la création de l'instance de tableau. Ces valeurs ne peuvent pas être modifiées pendant la durée de vie de l'instance.
- Un tableau en escalier est un tableau de tableaux, et chaque tableau membre a la valeur `null` par défaut.
- Les tableaux sont indexés sur zéro : un tableau avec `n` éléments est indexée de `0` à `n-1`.
- Les éléments de tableau peuvent être de n'importe quel type, y compris un type tableau.
- Les types tableau sont des types référence dérivés du type de base abstrait Array. Tous les tableaux implémentent IList et IEnumerable. Vous pouvez utiliser l'instruction foreach pour itérer au sein d'un tableau. Les tableaux unidimensionnels implémentent aussi IList<T> et IEnumerable<T>.

Les éléments d'un tableau peuvent être initialisés sur des valeurs connues lors de la création du tableau. À compter de C# 12, tous les types de collection peuvent être initialisés à l'aide d'une expression de collection. Les éléments qui ne sont pas initialisés sont définis sur la valeur par défaut. La valeur par défaut est le modèle 0 bits. Tous les types de référence (y compris les types non-nullables) ont les valeurs `null`. Tous les types de valeur ont les modèles 0 bits. Cela signifie que la propriété Nullable<T>.HasValue est `false` et que la propriété Nullable<T>.Value n'est pas définie. Dans l'implémentation .NET, la propriété `Value` lève une exception.

L'exemple suivant crée des tableaux unidimensionnels, multidimensionnels et en escalier :

```
C#
```

```
// Declare a single-dimensional array of 5 integers.
int[] array1 = new int[5];

// Declare and set array element values.
int[] array2 = { 1, 2, 3, 4, 5, 6 };

// Declare a two dimensional array.
int[,] multiDimensionalArray1 = new int[2, 3];

// Declare and set array element values.
int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };

// Declare a jagged array.
int[][] jaggedArray = new int[6][];

// Set the values of the first array in the jagged array structure.
jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
```

Tableaux unidimensionnels

Un *tableau unidimensionnel* est une séquence d'éléments similaires. Vous accédez à un élément via son *index*. L'*index* est sa position ordinale dans la séquence. Le premier élément du tableau est à l'index `0`. Vous créez un tableau unidimensionnel à l'aide du nouvel opérateur en spécifiant le type d'élément du tableau et le nombre d'éléments. L'exemple suivant déclare et initialise des tableaux unidimensionnels :

```
C#
```

```
int[] array = new int[5];
string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };

Console.WriteLine(weekDays[0]);
Console.WriteLine(weekDays[1]);
Console.WriteLine(weekDays[2]);
Console.WriteLine(weekDays[3]);
Console.WriteLine(weekDays[4]);
Console.WriteLine(weekDays[5]);
Console.WriteLine(weekDays[6]);

/*Output:
Sun
Mon
Tue
Wed
```

```
Thu
Fri
Sat
*/
```

La première déclaration déclare un tableau non initialisé de cinq entiers, de `array[0]` à `array[4]`. Les éléments du tableau sont initialisés à la valeur par défaut du type d'élément, `0` pour les entiers. La deuxième déclaration déclare un tableau de chaînes et initialise les sept valeurs de ce tableau. Une instruction foreach itère les éléments du tableau `weekday` et affiche toutes les valeurs. Pour les tableaux unidimensionnels, l'instruction `foreach` traite les éléments dans l'ordre croissant des index, en commençant par l'index 0 et en terminant par l'index `Length - 1`.

Passer des tableaux unidimensionnels en tant qu'arguments

Vous pouvez passer un tableau unidimensionnel initialisé à une méthode. Dans l'exemple suivant, un tableau de chaînes est initialisé et passé en tant qu'argument à une méthode `DisplayArray` pour des chaînes. La méthode affiche les éléments du tableau. Ensuite, la méthode `ChangeArray` inverse les éléments du tableau, puis la méthode `ChangeArrayElements` modifie les trois premiers éléments du tableau. Une fois que chaque méthode est retournée, la méthode `DisplayArray` montre que le passage d'un tableau par valeur n'empêche pas les modifications des éléments du tableau.

C#

```
class ArrayExample
{
    static void DisplayArray(string[] arr) =>
Console.WriteLine(string.Join(" ", arr));

    // Change the array by reversing its elements.
    static void ChangeArray(string[] arr) => Array.Reverse(arr);

    static void ChangeArrayElements(string[] arr)
    {
        // Change the value of the first three array elements.
        arr[0] = "Mon";
        arr[1] = "Wed";
        arr[2] = "Fri";
    }

    static void Main()
    {
        // Declare and initialize an array.
        string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri",
```

```
"Sat" };
        // Display the array elements.
        DisplayArray(weekDays);
        Console.WriteLine();

        // Reverse the array.
        ChangeArray(weekDays);
        // Display the array again to verify that it stays reversed.
        Console.WriteLine("Array weekDays after the call to ChangeArray:");
        DisplayArray(weekDays);
        Console.WriteLine();

        // Assign new values to individual array elements.
        ChangeArrayElements(weekDays);
        // Display the array again to verify that it has changed.
        Console.WriteLine("Array weekDays after the call to
ChangeArrayElements:");
        DisplayArray(weekDays);
    }
}
// The example displays the following output:
//         Sun Mon Tue Wed Thu Fri Sat
//
//        Array weekDays after the call to ChangeArray:
//        Sat Fri Thu Wed Tue Mon Sun
//
//        Array weekDays after the call to ChangeArrayElements:
//        Mon Wed Fri Wed Tue Mon Sun
```

Tableaux multidimensionnels

Les tableaux peuvent avoir plusieurs dimensions. Par exemple, les déclarations suivantes créent quatre tableaux : deux ont deux dimensions, deux ont trois dimensions. Les deux premières déclarations déclarent la longueur de chaque dimension, mais n'initialisent pas les valeurs du tableau. Les deux deuxièmes déclarations utilisent un initialiseur pour définir les valeurs de chaque élément dans le tableau multidimensionnel.

C#

```
int[,] array2DDeclaration = new int[4, 2];

int[,,] array3DDeclaration = new int[4, 2, 3];

// Two-dimensional array.
int[,] array2DInitialization =  { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// Three-dimensional array.
int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,   5,  6 } },
                                { { 7, 8, 9 }, { 10, 11, 12 } } };

// Accessing array elements.
```

```
System.Console.WriteLine(array2DInitialization[0, 0]);
System.Console.WriteLine(array2DInitialization[0, 1]);
System.Console.WriteLine(array2DInitialization[1, 0]);
System.Console.WriteLine(array2DInitialization[1, 1]);

System.Console.WriteLine(array2DInitialization[3, 0]);
System.Console.WriteLine(array2DInitialization[3, 1]);
// Output:
// 1
// 2
// 3
// 4
// 7
// 8

System.Console.WriteLine(array3D[1, 0, 1]);
System.Console.WriteLine(array3D[1, 1, 2]);
// Output:
// 8
// 12

// Getting the total count of elements or the length of a given dimension.
var allLength = array3D.Length;
var total = 1;
for (int i = 0; i < array3D.Rank; i++)
{
    total *= array3D.GetLength(i);
}
System.Console.WriteLine($"{allLength} equals {total}");
// Output:
// 12 equals 12
```

Pour les tableaux multidimensionnels, les éléments sont traités de sorte que les index de la dimension la plus à droite sont incrémentés en premier, puis la dimension immédiatement à gauche, et ainsi de suite jusqu'à l'index le plus à gauche. L'exemple suivant énumère à la fois un tableau 2D et un tableau 3D :

C#

```
int[,] numbers2D = { { 9, 99 }, { 3, 33 }, { 5, 55 } };

foreach (int i in numbers2D)
{
    System.Console.Write($"{i} ");
}
// Output: 9 99 3 33 5 55

int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,   5,  6 } },
                          { { 7, 8, 9 }, { 10, 11, 12 } } };
foreach (int i in array3D)
{
    System.Console.Write($"{i} ");
```

```
}
// Output: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
```

Dans un tableau 2D, vous pouvez considérer l'index de gauche comme la *ligne* et l'index de droite comme la *colonne*.

Cependant, dans le cas de tableaux multidimensionnels, l'utilisation d'une boucle for imbriquée vous permet de mieux contrôler l'ordre dans lequel les éléments du tableau sont traités :

C#

```
int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,   5,  6 } },
                        { { 7, 8, 9 }, { 10, 11, 12 } } };

for (int i = 0; i < array3D.GetLength(0); i++)
{
    for (int j = 0; j < array3D.GetLength(1); j++)
    {
        for (int k = 0; k < array3D.GetLength(2); k++)
        {
            System.Console.Write($"{array3D[i, j, k]} ");
        }
        System.Console.WriteLine();
    }
    System.Console.WriteLine();
}
// Output (including blank lines):
// 1 2 3
// 4 5 6
//
// 7 8 9
// 10 11 12
//
```

Passer des tableaux multidimensionnels en tant qu'arguments

Vous pouvez passer un tableau multidimensionnel initialisé à une méthode de la même manière que vous passez un tableau unidimensionnel. Le code suivant illustre une déclaration partielle d'une méthode Print qui accepte un tableau à deux dimensions en tant qu'argument. Vous pouvez initialiser et passer un nouveau tableau en une seule étape, comme dans l'exemple suivant. Dans l'exemple suivant, un tableau à deux dimensions d'entiers est initialisé et passé à la méthode `Print2DArray`. La méthode affiche les éléments du tableau.

C#

```
static void Print2DArray(int[,] arr)
{
    // Display the array elements.
    for (int i = 0; i < arr.GetLength(0); i++)
    {
        for (int j = 0; j < arr.GetLength(1); j++)
        {
            System.Console.WriteLine("Element({0},{1})={2}", i, j, arr[i,
j]);
        }
    }
}
static void ExampleUsage()
{
    // Pass the array as an argument.
    Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } });
}
/* Output:
    Element(0,0)=1
    Element(0,1)=2
    Element(1,0)=3
    Element(1,1)=4
    Element(2,0)=5
    Element(2,1)=6
    Element(3,0)=7
    Element(3,1)=8
*/
```

Tableaux en escalier

Un tableau en escalier est un tableau dont les éléments sont des tableaux, éventuellement de différentes tailles. Un tableau en escalier est parfois appelé « tableau de tableaux ». Ses éléments sont des types de référence et sont initialisés sur `null`. Les exemples suivants montrent comment déclarer, initialiser et accéder aux tableaux en escalier. Le premier exemple, `jaggedArray`, est déclaré dans une instruction. Chaque tableau contenu est créé dans les instructions suivantes. Le deuxième exemple, `jaggedArray2`, est déclaré et initialisé dans une instruction. Il est possible de combiner des tableaux en escalier et des tableaux multidimensionnels. Le dernier exemple, `jaggedArray3`, déclare et initialise un tableau en escalier unidimensionnel composé de trois éléments de tableau à deux dimensions de tailles différentes.

C#

```
int[][] jaggedArray = new int[3][];
```

```
jaggedArray[0] = new int[] { 1, 3, 5, 7, 9 };
jaggedArray[1] = new int[] { 0, 2, 4, 6 };
jaggedArray[2] = new int[] { 11, 22 };

int[][] jaggedArray2 =
{
    new int[] { 1, 3, 5, 7, 9 },
    new int[] { 0, 2, 4, 6 },
    new int[] { 11, 22 }
};

// Assign 77 to the second element ([1]) of the first array ([0]):
jaggedArray2[0][1] = 77;

// Assign 88 to the second element ([1]) of the third array ([2]):
jaggedArray2[2][1] = 88;

int[][,] jaggedArray3 = new int[3][,]
{
    new int[,] { {1,3}, {5,7} },
    new int[,] { {0,2}, {4,6}, {8,10} },
    new int[,] { {11,22}, {99,88}, {0,9} }
};

Console.Write("{0}", jaggedArray3[0][1, 0]);
Console.WriteLine(jaggedArray3.Length);
```

Les éléments d'un tableau en escalier doivent être initialisés avant de pouvoir les utiliser. Chacun des éléments est lui-même un tableau. Il est également possible d'utiliser des initialiseurs pour remplir les éléments de tableau avec des valeurs. Lorsque vous utilisez des initialiseurs, vous n'avez pas besoin de la taille du tableau.

Cet exemple génère un tableau dont les éléments sont eux-mêmes des tableaux. Chacun des éléments de tableau ont une taille différente.

C#

```
// Declare the array of two elements.
int[][] arr = new int[2][];

// Initialize the elements.
arr[0] = new int[5] { 1, 3, 5, 7, 9 };
arr[1] = new int[4] { 2, 4, 6, 8 };

// Display the array elements.
for (int i = 0; i < arr.Length; i++)
{
    System.Console.Write("Element({0}): ", i);

    for (int j = 0; j < arr[i].Length; j++)
    {
        System.Console.Write("{0}{1}", arr[i][j], j == (arr[i].Length - 1) ?
```

```
"" : " ");
    }
    System.Console.WriteLine();
}
/* Output:
    Element(0): 1 3 5 7 9
    Element(1): 2 4 6 8
*/
```

Tableaux implicitement typés

Vous pouvez créer un tableau implicitement typé dans lequel le type de l'instance de tableau est déduit à partir des éléments spécifiés dans l'initialiseur de tableau. Les règles applicables à une variable implicitement typée s'appliquent aussi aux tableaux implicitement typés. Pour plus d'informations, consultez Variables locales implicitement typées.

Les exemples suivants montrent comment créer un tableau implicitement typé :

C#

```
var a = new[] { 1, 10, 100, 1000 }; // int[]

// Accessing array
Console.WriteLine("First element: " + a[0]);
Console.WriteLine("Second element: " + a[1]);
Console.WriteLine("Third element: " + a[2]);
Console.WriteLine("Fourth element: " + a[3]);
/* Outputs
First element: 1
Second element: 10
Third element: 100
Fourth element: 1000
*/

var b = new[] { "hello", null, "world" }; // string[]

// Accessing elements of an array using 'string.Join' method
Console.WriteLine(string.Join(" ", b));
/* Output
hello  world
*/

// single-dimension jagged array
var c = new[]
{
                                    new[]{1,2,3,4},
                                    new[]{5,6,7,8}
            };
// Looping through the outer array
```

```
for (int k = 0; k < c.Length; k++)
{
    // Looping through each inner array
    for (int j = 0; j < c[k].Length; j++)
    {
        // Accessing each element and printing it to the console
        Console.WriteLine($"Element at c[{k}][{j}] is: {c[k][j]}");
    }
}
/* Outputs
Element at c[0][0] is: 1
Element at c[0][1] is: 2
Element at c[0][2] is: 3
Element at c[0][3] is: 4
Element at c[1][0] is: 5
Element at c[1][1] is: 6
Element at c[1][2] is: 7
Element at c[1][3] is: 8
*/

// jagged array of strings
var d = new[]
{
                new[]{"Luca", "Mads", "Luke", "Dinesh"},
                new[]{"Karen", "Suma", "Frances"}
            };

// Looping through the outer array
int i = 0;
foreach (var subArray in d)
{
    // Looping through each inner array
    int j = 0;
    foreach (var element in subArray)
    {
        // Accessing each element and printing it to the console
        Console.WriteLine($"Element at d[{i}][{j}] is: {element}");
        j++;
    }
    i++;
}
/* Outputs
Element at d[0][0] is: Luca
Element at d[0][1] is: Mads
Element at d[0][2] is: Luke
Element at d[0][3] is: Dinesh
Element at d[1][0] is: Karen
Element at d[1][1] is: Suma
Element at d[1][2] is: Frances
*/
```

Dans l'exemple précédent, vous pouvez constater que dans le cas des tableaux implicitement typés, aucun crochet n'est utilisé à gauche de l'instruction d'initialisation.

De plus, les tableaux en escalier sont initialisés à l'aide de `new []` à l'instar des tableaux unidimensionnels.

Quand vous créez un type anonyme qui contient un tableau, le tableau doit être implicitement typé dans l'initialiseur d'objet du type. Dans l'exemple suivant, `contacts` est un tableau implicitement typé de types anonymes, dont chacun contient un tableau nommé `PhoneNumbers`. Le mot clé `var` n'est pas utilisé dans les initialiseurs d'objets.

C#

```
var contacts = new[]
{
        new {
            Name = " Eugene Zabokritski",
            PhoneNumbers = new[] { "206-555-0108", "425-555-0001" }
        },
        new {
            Name = " Hanying Feng",
            PhoneNumbers = new[] { "650-555-0199" }
        }
    };
```

Tableaux

Article • 04/09/2023

Vous pouvez stocker plusieurs variables du même type dans une structure de données de type tableau. Vous déclarez un tableau en spécifiant le type de ses éléments. Si vous souhaitez que le tableau stocke des éléments de n'importe quel type, vous pouvez spécifier `object` comme type. Dans le système de type unifié de C#, tous les types (les types référence et valeur, prédéfinis ou définis par l'utilisateur) héritent directement ou indirectement du type Object.

```
C#

type[] arrayName;
```

Un tableau possède les propriétés suivantes :

- Un tableau peut être unidimensionnel, multidimensionnel ou en escalier.
- Le nombre de dimensions est défini lorsqu'une variable de tableau est déclarée. La longueur de chaque dimension est établie lors de la création de l'instance de tableau. Ces valeurs ne peuvent pas être modifiées pendant la durée de vie de l'instance.
- Un tableau en escalier est un tableau de tableaux, et chaque tableau membre a la valeur `null` par défaut.
- Les tableaux sont indexés sur zéro : un tableau avec `n` éléments est indexée de `0` à `n-1`.
- Les éléments de tableau peuvent être de n'importe quel type, y compris un type tableau.
- Les types tableau sont des types référence dérivés du type de base abstrait Array. Tous les tableaux implémentent IList et IEnumerable. Vous pouvez utiliser l'instruction foreach pour itérer au sein d'un tableau. Les tableaux unidimensionnels implémentent aussi IList<T> et IEnumerable<T>.

Les éléments d'un tableau peuvent être initialisés sur des valeurs connues lors de la création du tableau. À compter de C# 12, tous les types de collection peuvent être initialisés à l'aide d'une expression de collection. Les éléments qui ne sont pas initialisés sont définis sur la valeur par défaut. La valeur par défaut est le modèle 0 bits. Tous les types de référence (y compris les types non-nullables) ont les valeurs `null`. Tous les types de valeur ont les modèles 0 bits. Cela signifie que la propriété Nullable<T>.HasValue est `false` et que la propriété Nullable<T>.Value n'est pas définie. Dans l'implémentation .NET, la propriété `Value` lève une exception.

L'exemple suivant crée des tableaux unidimensionnels, multidimensionnels et en escalier :

```
C#
```

```
// Declare a single-dimensional array of 5 integers.
int[] array1 = new int[5];

// Declare and set array element values.
int[] array2 = { 1, 2, 3, 4, 5, 6 };

// Declare a two dimensional array.
int[,] multiDimensionalArray1 = new int[2, 3];

// Declare and set array element values.
int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };

// Declare a jagged array.
int[][] jaggedArray = new int[6][];

// Set the values of the first array in the jagged array structure.
jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
```

Tableaux unidimensionnels

Un *tableau unidimensionnel* est une séquence d'éléments similaires. Vous accédez à un élément via son *index*. L'*index* est sa position ordinale dans la séquence. Le premier élément du tableau est à l'index `0`. Vous créez un tableau unidimensionnel à l'aide du nouvel opérateur en spécifiant le type d'élément du tableau et le nombre d'éléments. L'exemple suivant déclare et initialise des tableaux unidimensionnels :

```
C#
```

```
int[] array = new int[5];
string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };

Console.WriteLine(weekDays[0]);
Console.WriteLine(weekDays[1]);
Console.WriteLine(weekDays[2]);
Console.WriteLine(weekDays[3]);
Console.WriteLine(weekDays[4]);
Console.WriteLine(weekDays[5]);
Console.WriteLine(weekDays[6]);

/*Output:
Sun
Mon
Tue
Wed
```

```
Thu
Fri
Sat
*/
```

La première déclaration déclare un tableau non initialisé de cinq entiers, de `array[0]` à `array[4]`. Les éléments du tableau sont initialisés à la valeur par défaut du type d'élément, `0` pour les entiers. La deuxième déclaration déclare un tableau de chaînes et initialise les sept valeurs de ce tableau. Une instruction foreach itère les éléments du tableau `weekday` et affiche toutes les valeurs. Pour les tableaux unidimensionnels, l'instruction `foreach` traite les éléments dans l'ordre croissant des index, en commençant par l'index 0 et en terminant par l'index `Length - 1`.

Passer des tableaux unidimensionnels en tant qu'arguments

Vous pouvez passer un tableau unidimensionnel initialisé à une méthode. Dans l'exemple suivant, un tableau de chaînes est initialisé et passé en tant qu'argument à une méthode `DisplayArray` pour des chaînes. La méthode affiche les éléments du tableau. Ensuite, la méthode `ChangeArray` inverse les éléments du tableau, puis la méthode `ChangeArrayElements` modifie les trois premiers éléments du tableau. Une fois que chaque méthode est retournée, la méthode `DisplayArray` montre que le passage d'un tableau par valeur n'empêche pas les modifications des éléments du tableau.

C#

```
class ArrayExample
{
    static void DisplayArray(string[] arr) =>
Console.WriteLine(string.Join(" ", arr));

    // Change the array by reversing its elements.
    static void ChangeArray(string[] arr) => Array.Reverse(arr);

    static void ChangeArrayElements(string[] arr)
    {
        // Change the value of the first three array elements.
        arr[0] = "Mon";
        arr[1] = "Wed";
        arr[2] = "Fri";
    }

    static void Main()
    {
        // Declare and initialize an array.
        string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri",
```

```
"Sat" };
        // Display the array elements.
        DisplayArray(weekDays);
        Console.WriteLine();

        // Reverse the array.
        ChangeArray(weekDays);
        // Display the array again to verify that it stays reversed.
        Console.WriteLine("Array weekDays after the call to ChangeArray:");
        DisplayArray(weekDays);
        Console.WriteLine();

        // Assign new values to individual array elements.
        ChangeArrayElements(weekDays);
        // Display the array again to verify that it has changed.
        Console.WriteLine("Array weekDays after the call to
ChangeArrayElements:");
        DisplayArray(weekDays);
    }
}
// The example displays the following output:
//         Sun Mon Tue Wed Thu Fri Sat
//
//        Array weekDays after the call to ChangeArray:
//        Sat Fri Thu Wed Tue Mon Sun
//
//        Array weekDays after the call to ChangeArrayElements:
//        Mon Wed Fri Wed Tue Mon Sun
```

Tableaux multidimensionnels

Les tableaux peuvent avoir plusieurs dimensions. Par exemple, les déclarations suivantes créent quatre tableaux : deux ont deux dimensions, deux ont trois dimensions. Les deux premières déclarations déclarent la longueur de chaque dimension, mais n'initialisent pas les valeurs du tableau. Les deux deuxièmes déclarations utilisent un initialiseur pour définir les valeurs de chaque élément dans le tableau multidimensionnel.

C#

```
int[,] array2DDeclaration = new int[4, 2];

int[,,] array3DDeclaration = new int[4, 2, 3];

// Two-dimensional array.
int[,] array2DInitialization =  { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// Three-dimensional array.
int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,   5,  6 } },
                                { { 7, 8, 9 }, { 10, 11, 12 } } };

// Accessing array elements.
```

```
System.Console.WriteLine(array2DInitialization[0, 0]);
System.Console.WriteLine(array2DInitialization[0, 1]);
System.Console.WriteLine(array2DInitialization[1, 0]);
System.Console.WriteLine(array2DInitialization[1, 1]);

System.Console.WriteLine(array2DInitialization[3, 0]);
System.Console.WriteLine(array2DInitialization[3, 1]);
// Output:
// 1
// 2
// 3
// 4
// 7
// 8

System.Console.WriteLine(array3D[1, 0, 1]);
System.Console.WriteLine(array3D[1, 1, 2]);
// Output:
// 8
// 12

// Getting the total count of elements or the length of a given dimension.
var allLength = array3D.Length;
var total = 1;
for (int i = 0; i < array3D.Rank; i++)
{
    total *= array3D.GetLength(i);
}
System.Console.WriteLine($"{allLength} equals {total}");
// Output:
// 12 equals 12
```

Pour les tableaux multidimensionnels, les éléments sont traités de sorte que les index de la dimension la plus à droite sont incrémentés en premier, puis la dimension immédiatement à gauche, et ainsi de suite jusqu'à l'index le plus à gauche. L'exemple suivant énumère à la fois un tableau 2D et un tableau 3D :

C#

```
int[,] numbers2D = { { 9, 99 }, { 3, 33 }, { 5, 55 } };

foreach (int i in numbers2D)
{
    System.Console.Write($"{i} ");
}
// Output: 9 99 3 33 5 55

int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,   5,  6 } },
                          { { 7, 8, 9 }, { 10, 11, 12 } } };
foreach (int i in array3D)
{
    System.Console.Write($"{i} ");
```

```
}
// Output: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
```

Dans un tableau 2D, vous pouvez considérer l'index de gauche comme la *ligne* et l'index de droite comme la *colonne*.

Cependant, dans le cas de tableaux multidimensionnels, l'utilisation d'une boucle for imbriquée vous permet de mieux contrôler l'ordre dans lequel les éléments du tableau sont traités :

C#

```
int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,   5,  6 } },
                        { { 7, 8, 9 }, { 10, 11, 12 } } };

for (int i = 0; i < array3D.GetLength(0); i++)
{
    for (int j = 0; j < array3D.GetLength(1); j++)
    {
        for (int k = 0; k < array3D.GetLength(2); k++)
        {
            System.Console.Write($"{array3D[i, j, k]} ");
        }
        System.Console.WriteLine();
    }
    System.Console.WriteLine();
}
// Output (including blank lines):
// 1 2 3
// 4 5 6
//
// 7 8 9
// 10 11 12
//
```

Passer des tableaux multidimensionnels en tant qu'arguments

Vous pouvez passer un tableau multidimensionnel initialisé à une méthode de la même manière que vous passez un tableau unidimensionnel. Le code suivant illustre une déclaration partielle d'une méthode Print qui accepte un tableau à deux dimensions en tant qu'argument. Vous pouvez initialiser et passer un nouveau tableau en une seule étape, comme dans l'exemple suivant. Dans l'exemple suivant, un tableau à deux dimensions d'entiers est initialisé et passé à la méthode `Print2DArray`. La méthode affiche les éléments du tableau.

C#

```
static void Print2DArray(int[,] arr)
{
    // Display the array elements.
    for (int i = 0; i < arr.GetLength(0); i++)
    {
        for (int j = 0; j < arr.GetLength(1); j++)
        {
            System.Console.WriteLine("Element({0},{1})={2}", i, j, arr[i,
j]);
        }
    }
}
static void ExampleUsage()
{
    // Pass the array as an argument.
    Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } });
}
/* Output:
    Element(0,0)=1
    Element(0,1)=2
    Element(1,0)=3
    Element(1,1)=4
    Element(2,0)=5
    Element(2,1)=6
    Element(3,0)=7
    Element(3,1)=8
*/
```

Tableaux en escalier

Un tableau en escalier est un tableau dont les éléments sont des tableaux, éventuellement de différentes tailles. Un tableau en escalier est parfois appelé « tableau de tableaux ». Ses éléments sont des types de référence et sont initialisés sur `null`. Les exemples suivants montrent comment déclarer, initialiser et accéder aux tableaux en escalier. Le premier exemple, `jaggedArray`, est déclaré dans une instruction. Chaque tableau contenu est créé dans les instructions suivantes. Le deuxième exemple, `jaggedArray2`, est déclaré et initialisé dans une instruction. Il est possible de combiner des tableaux en escalier et des tableaux multidimensionnels. Le dernier exemple, `jaggedArray3`, déclare et initialise un tableau en escalier unidimensionnel composé de trois éléments de tableau à deux dimensions de tailles différentes.

C#

```
int[][] jaggedArray = new int[3][];
```

```
jaggedArray[0] = new int[] { 1, 3, 5, 7, 9 };
jaggedArray[1] = new int[] { 0, 2, 4, 6 };
jaggedArray[2] = new int[] { 11, 22 };

int[][] jaggedArray2 =
{
    new int[] { 1, 3, 5, 7, 9 },
    new int[] { 0, 2, 4, 6 },
    new int[] { 11, 22 }
};

// Assign 77 to the second element ([1]) of the first array ([0]):
jaggedArray2[0][1] = 77;

// Assign 88 to the second element ([1]) of the third array ([2]):
jaggedArray2[2][1] = 88;

int[][,] jaggedArray3 = new int[3][,]
{
    new int[,] { {1,3}, {5,7} },
    new int[,] { {0,2}, {4,6}, {8,10} },
    new int[,] { {11,22}, {99,88}, {0,9} }
};

Console.Write("{0}", jaggedArray3[0][1, 0]);
Console.WriteLine(jaggedArray3.Length);
```

Les éléments d'un tableau en escalier doivent être initialisés avant de pouvoir les utiliser. Chacun des éléments est lui-même un tableau. Il est également possible d'utiliser des initialiseurs pour remplir les éléments de tableau avec des valeurs. Lorsque vous utilisez des initialiseurs, vous n'avez pas besoin de la taille du tableau.

Cet exemple génère un tableau dont les éléments sont eux-mêmes des tableaux. Chacun des éléments de tableau ont une taille différente.

C#

```
// Declare the array of two elements.
int[][] arr = new int[2][];

// Initialize the elements.
arr[0] = new int[5] { 1, 3, 5, 7, 9 };
arr[1] = new int[4] { 2, 4, 6, 8 };

// Display the array elements.
for (int i = 0; i < arr.Length; i++)
{
    System.Console.Write("Element({0}): ", i);

    for (int j = 0; j < arr[i].Length; j++)
    {
        System.Console.Write("{0}{1}", arr[i][j], j == (arr[i].Length - 1) ?
```

```
"" : " ");
    }
    System.Console.WriteLine();
}
/* Output:
    Element(0): 1 3 5 7 9
    Element(1): 2 4 6 8
*/
```

Tableaux implicitement typés

Vous pouvez créer un tableau implicitement typé dans lequel le type de l'instance de tableau est déduit à partir des éléments spécifiés dans l'initialiseur de tableau. Les règles applicables à une variable implicitement typée s'appliquent aussi aux tableaux implicitement typés. Pour plus d'informations, consultez Variables locales implicitement typées.

Les exemples suivants montrent comment créer un tableau implicitement typé :

C#

```
var a = new[] { 1, 10, 100, 1000 }; // int[]

// Accessing array
Console.WriteLine("First element: " + a[0]);
Console.WriteLine("Second element: " + a[1]);
Console.WriteLine("Third element: " + a[2]);
Console.WriteLine("Fourth element: " + a[3]);
/* Outputs
First element: 1
Second element: 10
Third element: 100
Fourth element: 1000
*/

var b = new[] { "hello", null, "world" }; // string[]

// Accessing elements of an array using 'string.Join' method
Console.WriteLine(string.Join(" ", b));
/* Output
hello  world
*/

// single-dimension jagged array
var c = new[]
{
                                    new[]{1,2,3,4},
                                    new[]{5,6,7,8}
            };
// Looping through the outer array
```

```
for (int k = 0; k < c.Length; k++)
{
    // Looping through each inner array
    for (int j = 0; j < c[k].Length; j++)
    {
        // Accessing each element and printing it to the console
        Console.WriteLine($"Element at c[{k}][{j}] is: {c[k][j]}");
    }
}
/* Outputs
Element at c[0][0] is: 1
Element at c[0][1] is: 2
Element at c[0][2] is: 3
Element at c[0][3] is: 4
Element at c[1][0] is: 5
Element at c[1][1] is: 6
Element at c[1][2] is: 7
Element at c[1][3] is: 8
*/

// jagged array of strings
var d = new[]
{
                new[]{"Luca", "Mads", "Luke", "Dinesh"},
                new[]{"Karen", "Suma", "Frances"}
            };

// Looping through the outer array
int i = 0;
foreach (var subArray in d)
{
    // Looping through each inner array
    int j = 0;
    foreach (var element in subArray)
    {
        // Accessing each element and printing it to the console
        Console.WriteLine($"Element at d[{i}][{j}] is: {element}");
        j++;
    }
    i++;
}
/* Outputs
Element at d[0][0] is: Luca
Element at d[0][1] is: Mads
Element at d[0][2] is: Luke
Element at d[0][3] is: Dinesh
Element at d[1][0] is: Karen
Element at d[1][1] is: Suma
Element at d[1][2] is: Frances
*/
```

Dans l'exemple précédent, vous pouvez constater que dans le cas des tableaux implicitement typés, aucun crochet n'est utilisé à gauche de l'instruction d'initialisation.

De plus, les tableaux en escalier sont initialisés à l'aide de `new []` à l'instar des tableaux unidimensionnels.

Quand vous créez un type anonyme qui contient un tableau, le tableau doit être implicitement typé dans l'initialiseur d'objet du type. Dans l'exemple suivant, `contacts` est un tableau implicitement typé de types anonymes, dont chacun contient un tableau nommé `PhoneNumbers`. Le mot clé `var` n'est pas utilisé dans les initialiseurs d'objets.

C#

```
var contacts = new[]
{
        new {
            Name = " Eugene Zabokritski",
            PhoneNumbers = new[] { "206-555-0108", "425-555-0001" }
        },
        new {
            Name = " Hanying Feng",
            PhoneNumbers = new[] { "650-555-0199" }
        }
    };
```

Chaînes et littéraux de chaîne

Article • 25/02/2023

Une chaîne est un objet de type String dont la valeur est du texte. En interne, le texte est stocké sous la forme d'une collection séquentielle en lecture seule d'objets Char. Il n'existe aucun caractère Null à la fin d'une chaîne C# ; par conséquent, une chaîne C# peut contenir n'importe quel nombre de caractères Null incorporés ('\0'). La propriété Length d'une chaîne représente le nombre d'objets `Char` qu'elle contient, et non pas le nombre de caractères Unicode. Pour accéder à des points de code Unicode individuels dans une chaîne, utilisez l'objet StringInfo.

chaîne vs. System.String

En C#, le mot clé `string` est un alias pour String. Par conséquent, `String` et `string` sont équivalents, même s'il est recommandé d'utiliser l'alias `string` fourni car il fonctionne même sans `using System;`. La classe `String` fournit de nombreuses méthodes pour créer, manipuler et comparer des chaînes en toute sécurité. En outre, le langage C# surcharge certains opérateurs pour simplifier les opérations de chaînes courantes. Pour plus d'informations sur le mot clé, voir chaîne. Pour plus d'informations sur le type et ses méthodes, consultez String.

Déclaration et initialisation de chaînes

Vous pouvez déclarer et initialiser des chaînes de différentes manières, comme illustré dans l'exemple suivant :

C#

```
// Declare without initializing.
string message1;

// Initialize to null.
string message2 = null;

// Initialize as an empty string.
// Use the Empty constant instead of the literal "".
string message3 = System.String.Empty;

// Initialize with a regular string literal.
string oldPath = "c:\\Program Files\\Microsoft Visual Studio 8.0";

// Initialize with a verbatim string literal.
string newPath = @"c:\Program Files\Microsoft Visual Studio 9.0";
```

```
// Use System.String if you prefer.
System.String greeting = "Hello World!";

// In local variables (i.e. within a method body)
// you can use implicit typing.
var temp = "I'm still a strongly-typed System.String!";

// Use a const string to prevent 'message4' from
// being used to store another string value.
const string message4 = "You can't get rid of me!";

// Use the String constructor only when creating
// a string from a char*, char[], or sbyte*. See
// System.String documentation for details.
char[] letters = { 'A', 'B', 'C' };
string alphabet = new string(letters);
```

Vous n'utilisez pas l'opérateur new pour créer un objet de chaîne, excepté lors de l'initialisation de la chaîne avec un tableau de chars.

Initialisez une chaîne avec la valeur constante Empty pour créer un objet String dont la chaîne est de longueur nulle. La représentation de littéral de chaîne d'une chaîne de longueur nulle est "". En initialisant les chaînes avec la valeur Empty au lieu de null, vous pouvez réduire les risques de levée de l'exception NullReferenceException. Utilisez la méthode statique IsNullOrEmpty(String) pour vérifier la valeur d'une chaîne avant de tenter d'y accéder.

Immuabilité des chaînes

Les objets de chaîne sont *immuables* : une fois créés, ils ne peuvent pas être modifiés. Toutes les méthodes String et tous les opérateurs C# qui semblent modifier une chaîne retournent en fait les résultats dans un nouvel objet string. Dans l'exemple suivant, lorsque les contenus de `s1` et `s2` sont concaténés pour former une chaîne unique, les deux chaînes d'origine restent inchangées. L'opérateur `+=` crée une nouvelle chaîne qui contient le contenu combiné. Ce nouvel objet est assigné à la variable `s1` et l'objet d'origine qui a été assigné à `s1` est libéré pour la garbage collection, car aucune autre variable ne le référence.

C#

```
string s1 = "A string is more ";
string s2 = "than the sum of its chars.";

// Concatenate s1 and s2. This actually creates a new
// string object and stores it in s1, releasing the
```

```
// reference to the original object.
s1 += s2;

System.Console.WriteLine(s1);
// Output: A string is more than the sum of its chars.
```

Étant donné qu'une « modification » de chaîne correspond en fait à la création d'une nouvelle chaîne, vous devez être prudent lorsque vous créez des références aux chaînes. Si vous créez une référence à une chaîne, puis « modifiez » la chaîne d'origine, la référence continue à pointer vers l'objet d'origine au lieu du nouvel objet créé lors de la modification de la chaîne. Le code suivant illustre ce comportement :

C#

```
string str1 = "Hello ";
string str2 = str1;
str1 += "World";

System.Console.WriteLine(str2);
//Output: Hello
```

Pour plus d'informations sur la création de nouvelles chaînes basées sur des modifications telles que les opérations rechercher-remplacer sur la chaîne d'origine, voir Comment modifier du contenu de chaîne.

Littéraux de chaîne entre guillemets

Les littéraux de chaîne entre guillemets commencent et se terminent par un simple guillemet double (`"`) sur la même ligne. Les littéraux de chaîne entre guillemets conviennent mieux aux chaînes qui s'ajustent sur une seule ligne et n'incluent aucune séquence d'échappement. Un littéral de chaîne entre guillemets doit incorporer des caractères d'échappement, comme illustré dans l'exemple suivant :

C#

```
string columns = "Column 1\tColumn 2\tColumn 3";
//Output: Column 1        Column 2        Column 3

string rows = "Row 1\r\nRow 2\r\nRow 3";
/* Output:
    Row 1
    Row 2
    Row 3
*/
```

```
string title = "\"The \u00C6olean Harp\", by Samuel Taylor Coleridge";
//Output: "The Æolean Harp", by Samuel Taylor Coleridge
```

Littéraux de chaîne textuelle

Les littéraux de chaîne verbatim sont plus pratiques pour les chaînes multilignes, les chaînes qui contiennent des caractères de barre oblique inverse ou des guillemets doubles incorporés. Les chaînes textuelles conservent les nouveaux caractères de ligne dans le texte de chaîne. Utilisez des guillemets doubles pour incorporer des guillemets dans une chaîne textuelle. L'exemple suivant montre certaines utilisations courantes des chaînes textuelles :

C#

```
string filePath = @"C:\Users\scoleridge\Documents\";
//Output: C:\Users\scoleridge\Documents\

string text = @"My pensive SARA ! thy soft cheek reclined
    Thus on mine arm, most soothing sweet it is
    To sit beside our Cot,...";
/* Output:
My pensive SARA ! thy soft cheek reclined
    Thus on mine arm, most soothing sweet it is
    To sit beside our Cot,...
*/

string quote = @"Her name was ""Sara.""";
//Output: Her name was "Sara."
```

Littéraux de chaîne bruts

À partir de C# 11, vous pouvez utiliser *des littéraux de chaîne brute* pour créer plus facilement des chaînes multilignes ou utiliser des caractères nécessitant des séquences d'échappement. *Les littéraux de chaîne brutes* suppriment la nécessité d'utiliser des séquences d'échappement. Vous pouvez écrire la chaîne, y compris la mise en forme des espaces blancs, de la façon dont vous souhaitez qu'elle apparaisse dans la sortie. *Littéral de chaîne brute* :

- Commence et se termine par une séquence d'au moins trois guillemets doubles (`"""`). Vous pouvez commencer et terminer la séquence avec plus de trois caractères consécutifs afin de prendre en charge les littéraux de chaîne qui contiennent trois (ou plus) guillemets répétés.

- Les littéraux de chaîne brute à une seule ligne nécessitent les guillemets d'ouverture et de fermeture sur la même ligne.
- Les littéraux de chaîne brute multilignes nécessitent des guillemets ouvrants et fermants sur leur propre ligne.
- Dans les littéraux de chaîne brute à plusieurs lignes, tout espace blanc à gauche des guillemets fermants est supprimé.

Les exemples suivants illustrent ces règles :

C#

```
string singleLine = """Friends say "hello" as they pass by.""";
string multiLine = """
    "Hello World!" is typically the first program someone writes.
    """;
string embeddedXML = """
       <element attr = "content">
           <body style="normal">
               Here is the main text
           </body>
           <footer>
               Excerpts from "An amazing story"
           </footer>
       </element >
       """;
// The line "<element attr = "content">" starts in the first column.
// All whitespace left of that column is removed from the string.

string rawStringLiteralDelimiter = """"
    Raw string literals are delimited
    by a string of at least three double quotes,
    like this: """
    """";
```

Les exemples suivants illustrent les erreurs du compilateur signalées en fonction de ces règles :

C#

```
// CS8997: Unterminated raw string literal.
var multiLineStart = """This
    is the beginning of a string
    """;

// CS9000: Raw string literal delimiter must be on its own line.
var multiLineEnd = """
    This is the beginning of a string """;

// CS8999: Line does not start with the same whitespace as the closing line
// of the raw string literal
```

```
var noOutdenting = """
    A line of text.
Trying to outdent the second line.
    """;
```

Les deux premiers exemples ne sont pas valides, car les littéraux de chaîne brute multiligne nécessitent la séquence de guillemets d'ouverture et de fermeture sur sa propre ligne. Le troisième exemple n'est pas valide, car le texte est sorti de la séquence de guillemets fermants.

Vous devez prendre en compte les littéraux de chaîne brute lorsque vous générez du texte qui comprend des caractères qui nécessitent des séquences d'échappement lors de l'utilisation de littéraux de chaîne entre guillemets ou de littéraux de chaînes verbatim. Les littéraux de chaîne brute seront plus faciles à lire pour vous et d'autres personnes, car ils ressembleront plus étroitement au texte de sortie. Par exemple, considérez le code suivant qui inclut une chaîne de JSON mise en forme :

C#

```
string jsonString = """
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "Hot",
  "DatesAvailable": [
    "2019-08-01T00:00:00-07:00",
    "2019-08-02T00:00:00-07:00"
  ],
  "TemperatureRanges": {
    "Cold": {
      "High": 20,
      "Low": -10
    },
    "Hot": {
      "High": 60,
      "Low": 20
    }
            },
  "SummaryWords": [
    "Cool",
    "Windy",
    "Humid"
  ]
}
""";
```

Comparez ce texte avec le texte équivalent de notre exemple sur la sérialisation JSON, qui n'utilise pas cette nouvelle fonctionnalité.

Séquences d’échappement de chaîne

Séquence d'échappement	Nom du caractère	Codage Unicode
\'	Guillemet simple	0x0027
\"	Guillemet double	0x0022
\\	Barre oblique inverse	0x005C
\0	Null	0x0000
\a	Alerte	0x0007
\b	Retour arrière	0x0008
\f	Saut de page	0x000C
\n	Nouvelle ligne	0x000A
\r	Retour chariot	0x000D
\t	Tabulation horizontale	0x0009
\v	Tabulation verticale	0x000B
\u	Séquence d’échappement Unicode (UTF-16)	`\uHHHH` (plage : 0000-FFFF ; exemple : `\u00E7` = « ç »)
\U	Séquence d’échappement Unicode (UTF-32)	`\U00HHHHHH` (plage : 000000 - 10FFFF ; exemple : `\U0001F47D` = « 👽 »)
\x	Séquence d’échappement Unicode similaire à "\u", mais avec une longueur variable	`\xH[H][H][H]` (plage : 0-FFFF ; exemple : `\x00E7` ou `\x0E7` ou `\xE7` = « ç »)

> ⚠ **Avertissement**
>
> Quand vous utilisez la séquence d’échappement `\x` et spécifiez moins de 4 chiffres hexadécimaux, si les caractères qui suivent immédiatement la séquence d’échappement sont des chiffres hexadécimaux valides (par ex. 0-9, A-F et a-f), ils sont interprétés comme faisant partie de la séquence d’échappement. Par exemple, `\xA1` produit « ¡ », qui est le point de code U+00A1. Toutefois, si le caractère suivant est « A » ou « a », la séquence d’échappement est interprétée comme étant `\xA1A` et génère « ਹ », qui correspond au point de code U+0A1A. Dans ce cas, la

spécification des 4 chiffres hexadécimaux (par ex. `\x00A1`) empêche toute mauvaise interprétation possible.

> ⓘ **Notes**
>
> Au moment de la compilation, les chaînes textuelles sont converties en chaînes normales avec les mêmes séquences d'échappement. Par conséquent, si vous affichez une chaîne textuelle dans la fenêtre Espion du débogueur, vous verrez les caractères d'échappement qui ont été ajoutés par le compilateur et non la version textuelle de votre code source. Par exemple, la chaîne textuelle `@"C:\files.txt"` s'affiche dans la fenêtre Espion en tant que « C:\\files.txt ».

Chaînes de format

Une chaîne de format est une chaîne dont le contenu est déterminé de manière dynamique au moment du runtime. Les chaînes de format sont créées en incorporant des *expressions interpolées* ou des espaces réservés à l'intérieur d'accolades dans une chaîne. Tout ce qui se trouve à l'intérieur des accolades (`{...}`) sera résolu en une valeur et sortie sous forme de chaîne mise en forme au moment du runtime. Il existe deux méthodes pour créer des chaînes de format : l'interpolation de chaîne et la mise en forme composite.

Interpolation de chaîne

Disponible dans C# 6.0 et versions ultérieures, les *chaînes interpolées* sont identifiées par le caractère spéciale `$` et incluent des expressions interpolées entre accolades. Si vous ne connaissez pas l'interpolation de chaîne, consultez le tutoriel interactif Interpolation de chaînes en C# pour obtenir une vue d'ensemble.

Utilisez l'interpolation de chaîne pour améliorer la lisibilité et la maintenance de votre code. L'interpolation de chaîne permet d'obtenir les mêmes résultats que la méthode `String.Format`, mais avec plus de facilité d'utilisation et de clarté.

C#

```
var jh = (firstName: "Jupiter", lastName: "Hammon", born: 1711, published:
1761);
Console.WriteLine($"{jh.firstName} {jh.lastName} was an African American
poet born in {jh.born}.");
Console.WriteLine($"He was first published in {jh.published} at the age of
{jh.published - jh.born}.");
```

```
Console.WriteLine($"He'd be over {Math.Round((2018d - jh.born) / 100d) *
100d} years old today.");

// Output:
// Jupiter Hammon was an African American poet born in 1711.
// He was first published in 1761 at the age of 50.
// He'd be over 300 years old today.
```

À partir de C# 10, vous pouvez utiliser l'interpolation de chaîne pour initialiser une chaîne constante lorsque toutes les expressions utilisées pour les espaces réservés sont également des chaînes constantes.

À partir de C# 11, vous pouvez combiner *des littéraux de chaîne brute avec des* interpolations de chaîne. Vous commencez et terminez la chaîne de format par au moins trois guillemets doubles successifs. Si votre chaîne de sortie doit contenir le caractère `{` ou `}`, vous pouvez utiliser des caractères supplémentaires `$` pour spécifier le nombre de caractères `{` et `}` de début de fin d'une interpolation. Toute séquence de moins de caractères `{` ou `}` de caractères est incluse dans la sortie. L'exemple suivant montre comment utiliser cette fonctionnalité pour afficher la distance d'un point par rapport à l'origine et placer le point à l'intérieur des accolades :

C#

```
int X = 2;
int Y = 3;

var pointMessage = $$"""The point {{{X}}, {{Y}}} is {{Math.Sqrt(X * X + Y *
Y)}} from the origin.""";

Console.WriteLine(pointMessage);
// Output:
// The point {2, 3} is 3.605551275463989 from the origin.
```

Mise en forme composite

String.Format utilise des espaces réservés entre accolades pour créer une chaîne de format. Cet exemple renvoie une sortie similaire à la méthode d'interpolation de chaîne utilisée ci-dessus.

C#

```
var pw = (firstName: "Phillis", lastName: "Wheatley", born: 1753, published:
1773);
Console.WriteLine("{0} {1} was an African American poet born in {2}.",
pw.firstName, pw.lastName, pw.born);
Console.WriteLine("She was first published in {0} at the age of {1}.",
```

```
pw.published, pw.published - pw.born);
Console.WriteLine("She'd be over {0} years old today.", Math.Round((2018d -
pw.born) / 100d) * 100d);

// Output:
// Phillis Wheatley was an African American poet born in 1753.
// She was first published in 1773 at the age of 20.
// She'd be over 300 years old today.
```

Pour plus d'informations sur la mise en forme des types .NET, consultez Mise en forme des types dans .NET.

Sous-chaînes

Une sous-chaîne est une séquence de caractères qui est contenue dans une chaîne. Utilisez la méthode Substring pour créer une chaîne à partir d'une partie de la chaîne d'origine. Vous pouvez rechercher une ou plusieurs occurrences d'une sous-chaîne en utilisant la méthode IndexOf. Utilisez la méthode Replace pour remplacer toutes les occurrences d'une sous-chaîne spécifiée par une nouvelle chaîne. Comme la méthode Substring, Replace retourne une nouvelle chaîne et ne modifie pas la chaîne d'origine. Pour plus d'informations, consultez Comment rechercher des chaînes et Comment modifier du contenu de chaîne.

C#

```
string s3 = "Visual C# Express";
System.Console.WriteLine(s3.Substring(7, 2));
// Output: "C#"

System.Console.WriteLine(s3.Replace("C#", "Basic"));
// Output: "Visual Basic Express"

// Index values are zero-based
int index = s3.IndexOf("C");
// index = 7
```

Accès aux caractères individuels

Vous pouvez utiliser la notation de tableau avec une valeur d'index pour obtenir un accès en lecture seule aux caractères individuels, comme dans l'exemple suivant :

C#

```
string s5 = "Printing backwards";
```

```
for (int i = 0; i < s5.Length; i++)
{
    System.Console.Write(s5[s5.Length - i - 1]);
}
// Output: "sdrawkcab gnitnirP"
```

Si les méthodes String ne fournissent pas la fonctionnalité nécessaire pour modifier les caractères individuels d'une chaîne, vous pouvez utiliser un objet StringBuilder pour modifier les caractères individuels « sur place », puis créer une chaîne pour stocker les résultats en utilisant les méthodes StringBuilder. Dans l'exemple suivant, supposons que vous devez modifier la chaîne d'origine d'une façon particulière, puis stocker les résultats pour une utilisation ultérieure :

C#

```
string question = "hOW DOES mICROSOFT wORD DEAL WITH THE cAPS lOCK KEY?";
System.Text.StringBuilder sb = new System.Text.StringBuilder(question);

for (int j = 0; j < sb.Length; j++)
{
    if (System.Char.IsLower(sb[j]) == true)
        sb[j] = System.Char.ToUpper(sb[j]);
    else if (System.Char.IsUpper(sb[j]) == true)
        sb[j] = System.Char.ToLower(sb[j]);
}
// Store the new string.
string corrected = sb.ToString();
System.Console.WriteLine(corrected);
// Output: How does Microsoft Word deal with the Caps Lock key?
```

Chaînes Null et chaînes vides

Une chaîne vide est une instance d'un objet System.String qui contient zéro caractère. Les chaînes vides sont souvent utilisées dans divers scénarios de programmation pour représenter un champ de texte vide. Vous pouvez appeler des méthodes sur des chaînes vides, car ce sont des objets System.String. Les chaînes vides sont initialisées comme suit :

C#

```
string s = String.Empty;
```

En revanche, une chaîne Null ne fait pas référence à une instance d'un objet System.String, et toute tentative pour appeler une méthode sur une chaîne Null lève une exception NullReferenceException. Toutefois, vous pouvez utiliser des chaînes Null dans

les opérations de comparaison et de concaténation avec d'autres chaînes. Les exemples suivants illustrent certains cas dans lesquels une référence à une chaîne Null provoque ou non la levée d'une exception :

C#

```
string str = "hello";
string nullStr = null;
string emptyStr = String.Empty;

string tempStr = str + nullStr;
// Output of the following line: hello
Console.WriteLine(tempStr);

bool b = (emptyStr == nullStr);
// Output of the following line: False
Console.WriteLine(b);

// The following line creates a new empty string.
string newStr = emptyStr + nullStr;

// Null strings and empty strings behave differently. The following
// two lines display 0.
Console.WriteLine(emptyStr.Length);
Console.WriteLine(newStr.Length);
// The following line raises a NullReferenceException.
//Console.WriteLine(nullStr.Length);

// The null character can be displayed and counted, like other chars.
string s1 = "\x0" + "abc";
string s2 = "abc" + "\x0";
// Output of the following line: * abc*
Console.WriteLine("*" + s1 + "*");
// Output of the following line: *abc *
Console.WriteLine("*" + s2 + "*");
// Output of the following line: 4
Console.WriteLine(s2.Length);
```

Utilisation de StringBuilder pour créer rapidement une chaîne

Les opérations Strings dans .NET sont hautement optimisées et, dans la plupart des cas, n'ont pas d'incidence significative sur les performances. Toutefois, dans certains scénarios, notamment dans le cas de boucles serrées qui s'exécutent plusieurs centaines voire milliers de fois, les opérations String peuvent affecter les performances. La classe StringBuilder crée un tampon de chaîne qui offre de meilleures performances si votre programme exécute de nombreuses manipulations de chaînes. La chaîne StringBuilder

vous permet également de réassigner des caractères individuels, une fonctionnalité non prise en charge par les types de données string intégrés. Ce code, par exemple, modifie le contenu d'une chaîne sans créer de nouvelle chaîne :

```
C#

System.Text.StringBuilder sb = new System.Text.StringBuilder("Rat: the ideal
pet");
sb[0] = 'C';
System.Console.WriteLine(sb.ToString());
//Outputs Cat: the ideal pet
```

Dans cet exemple, un objet StringBuilder est utilisé pour créer une chaîne à partir d'un ensemble de types numériques :

```
C#

var sb = new StringBuilder();

// Create a string composed of numbers 0 - 9
for (int i = 0; i < 10; i++)
{
    sb.Append(i.ToString());
}
Console.WriteLine(sb);  // displays 0123456789

// Copy one character of the string (not possible with a System.String)
sb[0] = sb[9];

Console.WriteLine(sb);  // displays 9123456789
```

Chaînes, méthodes d'extension et LINQ

Étant donné que le type String implémente IEnumerable<T>, vous pouvez utiliser les méthodes d'extension définies dans la classe Enumerable sur des chaînes. Pour éviter une surcharge visuelle, ces méthodes sont exclues d'IntelliSense pour le type String, mais elles restent néanmoins disponibles. Vous pouvez également utiliser des expressions de requête LINQ sur des chaînes. Pour plus d'informations, consultez LINQ et Strings.

Articles connexes

- Comment modifier du contenu de chaîne : Illustre les techniques permettant de transformer des chaînes et de modifier le contenu de chaînes.

- Comment comparer des chaînes : Montre comment effectuer des comparaisons de chaînes ordinales et spécifiques à la culture.
- Comment : concaténer plusieurs chaînes : Montre différentes façons de joindre plusieurs chaînes en une seule.
- Comment : analyser des chaînes à l'aide de String.Split : contient des exemples de code qui illustrent comment utiliser la String.Split méthode pour analyser les chaînes.
- Comment rechercher des chaînes : Explique comment rechercher du texte ou des modèles spécifiques dans des chaînes.
- Comment déterminer si une chaîne représente une valeur numérique : montre comment analyser une chaîne en toute sécurité pour voir si elle a une valeur numérique valide.
- Interpolation de chaîne : Décrit la fonctionnalité d'interpolation de chaîne qui fournit une syntaxe pratique pour les chaînes de format.
- Opérations de chaînes de base : Fournit des liens vers des rubriques utilisant les méthodes System.String et System.Text.StringBuilder pour effectuer des opérations de chaînes de base.
- Analyse de chaînes : Décrit comment convertir des représentations sous forme de chaîne de types de base .NET en instances de types correspondants.
- Analyse des chaînes de date et d'heure dans .NET : montre comment convertir une chaîne telle que « 24/01/2008 » en objet System.DateTime.
- Comparer des chaînes : Inclut des informations sur la façon de comparer des chaînes et fournit des exemples en C# et Visual Basic.
- Utilisation de la classe StringBuilder : Décrit la manière de créer et modifier des objets de chaîne dynamiques avec la classe StringBuilder.
- LINQ et chaînes : Fournit des informations sur l'exécution de différentes opérations sur des chaînes à l'aide de requêtes LINQ.

Guide pratique pour déterminer si une chaîne représente une valeur numérique (Guide de programmation C#)

Article • 10/05/2023

Pour déterminer si une chaîne est une représentation valide d'un type numérique spécifié, utilisez la méthode statique `TryParse` implémentée par tous les types numériques primitifs et par les types tels que DateTime et IPAddress. L'exemple suivant montre comment déterminer si « 108 » est une chaîne int valide.

C#

```
int i = 0;
string s = "108";
bool result = int.TryParse(s, out i); //i now = 108
```

Si la chaîne contient des caractères non numériques ou si la valeur numérique est trop grande ou trop petite pour le type particulier que vous avez spécifié, `TryParse` retourne la valeur false et affecte la valeur zéro comme paramètre de sortie. Sinon, elle retourne la valeur true et affecte la valeur numérique de la chaîne comme paramètre de sortie.

> ⓘ **Notes**
>
> Une chaîne peut contenir uniquement des caractères numériques et ne pas être valide pour le type dont vous utilisez la méthode `TryParse`. Par exemple, « 256 » n'est pas une valeur valide pour `byte`, mais elle est valide pour `int`. « 98,6 » n'est pas une valeur valide pour `int`, mais elle est valide pour `decimal`.

Exemple

Les exemples suivants montrent comment utiliser `TryParse` avec des représentations sous forme de chaîne de valeurs `long`, `byte` et `decimal`.

C#

```
string numString = "1287543"; //"1287543.0" will return false for a long
long number1 = 0;
bool canConvert = long.TryParse(numString, out number1);
```

```
if (canConvert == true)
Console.WriteLine("number1 now = {0}", number1);
else
Console.WriteLine("numString is not a valid long");

byte number2 = 0;
numString = "255"; // A value of 256 will return false
canConvert = byte.TryParse(numString, out number2);
if (canConvert == true)
Console.WriteLine("number2 now = {0}", number2);
else
Console.WriteLine("numString is not a valid byte");

decimal number3 = 0;
numString = "27.3"; //"27" is also a valid decimal
canConvert = decimal.TryParse(numString, out number3);
if (canConvert == true)
Console.WriteLine("number3 now = {0}", number3);
else
Console.WriteLine("number3 is not a valid decimal");
```

Programmation fiable

Les types numériques primitifs implémentent également la méthode statique `Parse`, qui lève une exception si la chaîne n'est pas un nombre valide. `TryParse` est généralement plus efficace, car elle retourne simplement false si le nombre n'est pas valide.

Sécurité .NET

Utilisez toujours la méthode `TryParse` ou `Parse` pour valider l'entrée utilisateur à partir de contrôles tels que des zones de texte et des zones de liste déroulante.

Voir aussi

- Comment convertir un tableau d'octets en entier
- Comment convertir une chaîne en nombre
- Comment effectuer une conversion entre des chaînes hexadécimales et des types numériques
- Analyse de chaînes numériques
- Mise en forme des types

Indexeurs (Guide de programmation C#)

Article • 25/02/2023

Les indexeurs permettent aux instances d'une classe ou d'un struct d'être indexés comme des tableaux. La valeur indexée peut être définie ou récupérée sans spécifier explicitement un membre de type ou d'instance. Les indexeurs s'apparentent aux propriétés à l'exception près que leurs accesseurs acceptent des paramètres.

L'exemple suivant définit une classe générique avec des méthodes d'accesseur get et set simples pour attribuer et récupérer des valeurs. La classe `Program` classe crée une instance de cette classe pour le stockage des chaînes.

C#

```
using System;

class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];

   // Define the indexer to allow client code to use [] notation.
   public T this[int i]
   {
      get { return arr[i]; }
      set { arr[i] = value; }
   }
}

class Program
{
   static void Main()
   {
      var stringCollection = new SampleCollection<string>();
      stringCollection[0] = "Hello, World";
      Console.WriteLine(stringCollection[0]);
   }
}
// The example displays the following output:
//       Hello, World.
```

> ⓘ Notes
>
> Pour plus d'exemples, consultez **Rubriques connexes**.

Définitions de corps d'expression

Il est courant pour l'accesseur get ou set d'un indexeur d'être constitué d'une instruction unique qui retourne ou définit une valeur. Les membres expression-bodied fournissent une syntaxe simplifiée pour prendre en charge ce scénario. À partir de C# 6, un indexeur en lecture seule peut être implémenté comme un membre expression-bodied, comme le montre l'exemple suivant.

C#

```
using System;

class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];
   int nextIndex = 0;

   // Define the indexer to allow client code to use [] notation.
   public T this[int i] => arr[i];

   public void Add(T value)
   {
      if (nextIndex >= arr.Length)
         throw new IndexOutOfRangeException($"The collection can hold only
{arr.Length} elements.");
      arr[nextIndex++] = value;
   }
}

class Program
{
   static void Main()
   {
      var stringCollection = new SampleCollection<string>();
      stringCollection.Add("Hello, World");
      System.Console.WriteLine(stringCollection[0]);
   }
}
// The example displays the following output:
//       Hello, World.
```

Notez que `=>` introduit le corps de l'expression et que le mot clé `get` n'est pas utilisé.

À partir de C# 7.0, l'accesseur get et l'accesseur set peuvent être implémentés en tant que membres expression-bodied. Dans ce cas, les deux mots clés `get` et `set` doivent être utilisés. Par exemple :

C#

```
using System;

class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];

   // Define the indexer to allow client code to use [] notation.
   public T this[int i]
   {
      get => arr[i];
      set => arr[i] = value;
   }
}

class Program
{
   static void Main()
   {
      var stringCollection = new SampleCollection<string>();
      stringCollection[0] = "Hello, World.";
      Console.WriteLine(stringCollection[0]);
   }
}
// The example displays the following output:
//       Hello, World.
```

Vue d'ensemble des indexeurs

- Les indexeurs permettent aux objets d'être indexés d'une manière similaire aux tableaux.
- Un accesseur `get` retourne une valeur. Un accesseur `set` affecte une valeur.
- Le mot clé this est utilisé pour définir l'indexeur.
- Le mot clé value est utilisé pour définir la valeur assignée par l'accesseur `set`.
- Les indexeurs n'ont pas besoin d'être indexés par une valeur entière. Il vous appartient de choisir comment définir le mécanisme de recherche spécifique.
- Les indexeurs peuvent être surchargés.
- Les indexeurs peuvent avoir plusieurs paramètres formels, par exemple, quand vous accédez à un tableau à deux dimensions.

Sections connexes

- Utilisation d'indexeurs
- Indexeurs dans les interfaces
- Comparaison entre propriétés et indexeurs
- Restriction d'accessibilité de l'accesseur

Spécification du langage C#

Pour plus d'informations, consultez Indexeurs dans la Spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Guide de programmation C#
- Propriétés

Utiliser des indexeurs (Guide de programmation C#)

Article • 07/04/2023

Les indexeurs simplifient, d'un point de vue syntaxique, la création d'une classe, d'un struct ou d'une interface auxquels les applications clientes peuvent accéder comme à un tableau. Le compilateur génère une propriété `Item` (ou une propriété nommée autrement si IndexerNameAttribute est présent) et les méthodes d'accesseur appropriées. Le plus souvent, les indexeurs sont implémentés dans les types dont l'objectif premier est d'encapsuler une collection ou un tableau interne. Prenons l'exemple d'une classe `TempRecord` qui représente la température, en Fahrenheit, enregistrée à 10 moments différents sur une période de 24 heures. La classe contient un tableau `temps` de type `float[]` pour stocker les valeurs de température. En implémentant un indexeur dans cette classe, les clients peuvent accéder aux températures dans une instance `TempRecord` sous la forme `float temp = tempRecord[4]` et non sous la forme `float temp = tempRecord.temps[4]`. La notation d'indexeur non seulement simplifie la syntaxe pour les applications clientes, mais elle permet aussi aux autres développeurs de comprendre de façon plus intuitive la classe et son objectif.

Pour déclarer un indexeur sur une classe ou un struct, utilisez le mot clé this, comme dans l'exemple suivant :

C#

```
// Indexer declaration
public int this[int index]
{
    // get and set accessors
}
```

> **ⓘ Important**
>
> La déclaration d'un indexeur génère automatiquement une propriété nommée `Item` sur l'objet. La propriété `Item` n'est pas directement accessible depuis l'**expression d'accès au membre** de l'instance. De plus, si vous ajoutez votre propre propriété `Item` à un objet avec un indexeur, vous obtenez une **erreur du compilateur CS0102**. Pour éviter cette erreur, utilisez **IndexerNameAttribute** et renommez l'indexeur comme indiqué ci-dessous.

Remarques

Le type d'un indexeur et le type de ses paramètres doivent être au moins aussi accessibles que l'indexeur lui-même. Pour plus d'informations sur les niveaux d'accessibilité, consultez Modificateurs d'accès.

Pour plus d'informations sur l'utilisation d'indexeurs avec une interface, consultez Indexeurs d'interface.

La signature d'un indexeur est composée du nombre et des types de ses paramètres formels. Elle ne comporte ni le type de l'indexeur ni le nom des paramètres formels. Si vous déclarez plusieurs indexeurs dans la même classe, ils doivent avoir des signatures différentes.

Un indexeur n'est pas classé en tant que variable, c'est pourquoi une valeur d'indexeur ne peut pas être passée par référence (en tant que paramètre ref ou out), sauf si sa valeur est une référence (c'est-à-dire qu'elle est retournée par référence).

Pour affecter à l'indexeur un nom exploitable dans d'autres langages, utilisez System.Runtime.CompilerServices.IndexerNameAttribute, comme dans l'exemple suivant :

C#

```
// Indexer declaration
[System.Runtime.CompilerServices.IndexerName("TheItem")]
public int this[int index]
{
    // get and set accessors
}
```

Cet indexeur a le nom `TheItem`, car il est remplacé par l'attribut de nom d'indexeur. Par défaut, le nom de l'indexeur est `Item`.

Exemple 1

L'exemple suivant montre comment déclarer un champ de tableau privé `temps`, et un indexeur. L'indexeur permet d'accéder directement à l'instance `tempRecord[i]`. Comme alternative à l'utilisation de l'indexeur, vous pouvez déclarer le tableau comme membre public et accéder directement à ses membres, `tempRecord.temps[i]`.

C#

```
public class TempRecord
{
    // Array of temperature values
    float[] temps = new float[10]
    {
        56.2F, 56.7F, 56.5F, 56.9F, 58.8F,
        61.3F, 65.9F, 62.1F, 59.2F, 57.5F
    };

    // To enable client code to validate input
    // when accessing your indexer.
    public int Length => temps.Length;

    // Indexer declaration.
    // If index is out of range, the temps array will throw the exception.
    public float this[int index]
    {
        get => temps[index];
        set => temps[index] = value;
    }
}
```

Notez que quand l'accès à un indexeur est évalué, par exemple dans une instruction `Console.Write`, l'accesseur get est appelé. C'est pourquoi une erreur de compilation se produit s'il n'existe aucun accesseur `get`.

C#

```
class Program
{
    static void Main()
    {
        var tempRecord = new TempRecord();

        // Use the indexer's set accessor
        tempRecord[3] = 58.3F;
        tempRecord[5] = 60.1F;

        // Use the indexer's get accessor
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine($"Element #{i} = {tempRecord[i]}");
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
    /* Output:
        Element #0 = 56.2
        Element #1 = 56.7
```

```
        Element #2 = 56.5
        Element #3 = 58.3
        Element #4 = 58.8
        Element #5 = 60.1
        Element #6 = 65.9
        Element #7 = 62.1
        Element #8 = 59.2
        Element #9 = 57.5
    */
}
```

Indexation avec d'autres valeurs

C# ne limite pas le type de paramètre d'indexeur au type entier. Par exemple, il peut être utile d'utiliser une chaîne avec un indexeur. Il est possible d'implémenter un tel indexeur en recherchant la chaîne dans la collection et en retournant la valeur appropriée. Comme les accesseurs peuvent être surchargés, les versions string et integer peuvent coexister.

Exemple 2

L'exemple suivant déclare une classe qui stocke les jours de la semaine. Un accesseur `get` prend une chaîne, le nom d'un jour, et retourne l'entier correspondant. Par exemple, « Sunday » retourne 0, « Monday » retourne 1 et ainsi de suite.

C#

```
// Using a string as an indexer value
class DayCollection
{
    string[] days = { "Sun", "Mon", "Tues", "Wed", "Thurs", "Fri", "Sat" };

    // Indexer with only a get accessor with the expression-bodied
definition:
    public int this[string day] => FindDayIndex(day);

    private int FindDayIndex(string day)
    {
        for (int j = 0; j < days.Length; j++)
        {
            if (days[j] == day)
            {
                return j;
            }
        }

        throw new ArgumentOutOfRangeException(
```

```
            nameof(day),
            $"Day {day} is not supported.\nDay input must be in the form
\"Sun\", \"Mon\", etc");
    }
}
```

Exemple de consommation 2

C#

```
class Program
{
    static void Main(string[] args)
    {
        var week = new DayCollection();
        Console.WriteLine(week["Fri"]);

        try
        {
            Console.WriteLine(week["Made-up day"]);
        }
        catch (ArgumentOutOfRangeException e)
        {
            Console.WriteLine($"Not supported input: {e.Message}");
        }
    }
    // Output:
    // 5
    // Not supported input: Day Made-up day is not supported.
    // Day input must be in the form "Sun", "Mon", etc (Parameter 'day')
}
```

Exemple 3

L'exemple suivant déclare une classe qui stocke les jours de la semaine à l'aide de l'enum System.DayOfWeek. Un accesseur `get` prend un `DayOfWeek`, la valeur d'un jour, et retourne l'entier correspondant. Par exemple, `DayOfWeek.Sunday` retourne 0, `DayOfWeek.Monday` retourne 1, et ainsi de suite.

C#

```
using Day = System.DayOfWeek;

class DayOfWeekCollection
{
    Day[] days =
    {
```

```
        Day.Sunday, Day.Monday, Day.Tuesday, Day.Wednesday,
        Day.Thursday, Day.Friday, Day.Saturday
    };

    // Indexer with only a get accessor with the expression-bodied
definition:
    public int this[Day day] => FindDayIndex(day);

    private int FindDayIndex(Day day)
    {
        for (int j = 0; j < days.Length; j++)
        {
            if (days[j] == day)
            {
                return j;
            }
        }
        throw new ArgumentOutOfRangeException(
            nameof(day),
            $"Day {day} is not supported.\nDay input must be a defined
System.DayOfWeek value.");
    }
}
```

Exemple de consommation 3

C#

```
class Program
{
    static void Main()
    {
        var week = new DayOfWeekCollection();
        Console.WriteLine(week[DayOfWeek.Friday]);

        try
        {
            Console.WriteLine(week[(DayOfWeek)43]);
        }
        catch (ArgumentOutOfRangeException e)
        {
            Console.WriteLine($"Not supported input: {e.Message}");
        }
    }
    // Output:
    // 5
    // Not supported input: Day 43 is not supported.
    // Day input must be a defined System.DayOfWeek value. (Parameter 'day')
}
```

Programmation fiable

La sécurité et la fiabilité des indexeurs peuvent être améliorées de deux manières principales :

- N'oubliez pas d'incorporer une stratégie de gestion des erreurs au cas où le code client passerait une valeur d'index non valide. Dans le premier exemple décrit plus haut dans cette rubrique, la classe TempRecord fournit une propriété Length qui permet au code client de vérifier l'entrée avant de la passer à l'indexeur. Vous pouvez également placer le code de gestion des erreurs à l'intérieur de l'indexeur lui-même. N'oubliez pas d'indiquer aux utilisateurs toutes les exceptions que vous levez dans un accesseur d'indexeur.

- Définissez pour les accesseurs get et set une accessibilité aussi restrictive que possible. Cela est particulièrement important dans le cas de l'accesseur `set`. Pour plus d'informations, consultez Restriction d'accessibilité de l'accesseur.

Voir aussi

- Guide de programmation C#
- Indexeurs
- Propriétés

Indexeurs dans les interfaces (Guide de programmation C#)

Article • 07/04/2023

Des indexeurs peuvent être déclarés dans une interface. Les accesseurs d'indexeurs d'interface se distinguent sur plusieurs plans des accesseurs d'indexeurs de classe, à savoir :

- Les accesseurs d'interface n'utilisent pas de modificateurs.
- Un accesseur d'interface n'a généralement pas de corps.

Un accesseur vise à indiquer si l'indexeur est en lecture-écriture, en lecture seule ou en écriture seule. Vous pouvez fournir une implémentation pour un indexeur défini dans une interface, mais cela est rare. Les indexeurs définissent généralement une API pour accéder aux champs de données et les champs de données ne peuvent pas être définis dans une interface.

L'exemple ci-dessous porte sur un accesseur d'indexeur d'interface :

C#

```
public interface ISomeInterface
{
    //...

    // Indexer declaration:
    string this[int index]
    {
        get;
        set;
    }
}
```

La signature d'un indexeur doit se distinguer de tous les autres indexeurs déclarés dans la même interface.

Exemple

L'exemple suivant montre comment implémenter des indexeurs d'interface.

C#

```
// Indexer on an interface:
public interface IIndexInterface
```

```
{
    // Indexer declaration:
    int this[int index]
    {
        get;
        set;
    }
}

// Implementing the interface.
class IndexerClass : IIndexInterface
{
    private int[] arr = new int[100];
    public int this[int index]   // indexer declaration
    {
        // The arr object will throw IndexOutOfRange exception.
        get => arr[index];
        set => arr[index] = value;
    }
}
```

C#

```
IndexerClass test = new IndexerClass();
System.Random rand = System.Random.Shared;
// Call the indexer to initialize its elements.
for (int i = 0; i < 10; i++)
{
    test[i] = rand.Next();
}
for (int i = 0; i < 10; i++)
{
    System.Console.WriteLine($"Element #{i} = {test[i]}");
}

/* Sample output:
    Element #0 = 360877544
    Element #1 = 327058047
    Element #2 = 1913480832
    Element #3 = 1519039937
    Element #4 = 601472233
    Element #5 = 323352310
    Element #6 = 1422639981
    Element #7 = 1797892494
    Element #8 = 875761049
    Element #9 = 393083859
*/
```

Dans l'exemple précédent, vous pouvez utiliser l'implémentation de membre d'interface explicite en utilisant le nom qualifié complet du membre d'interface. Par exemple

C#

```
string IIndexInterface.this[int index]
{
}
```

Cependant, le nom qualifié complet est seulement nécessaire pour éviter toute ambiguïté quand la classe implémente plusieurs interfaces avec la même signature d'indexeur. Par exemple, si une classe `Employee` implémente deux interfaces, `ICitizen` et `IEmployee`, et que les deux interfaces ont la même signature d'indexeur, l'implémentation de membre d'interface explicite est nécessaire. Autrement dit, la déclaration d'indexeur suivante :

C#

```
string IEmployee.this[int index]
{
}
```

implémente l'indexeur dans l'interface `IEmployee`, alors que la déclaration suivante :

C#

```
string ICitizen.this[int index]
{
}
```

implémente l'interface dans l'interface `ICitizen`.

Voir aussi

- Guide de programmation C#
- Indexeurs
- Propriétés
- Interfaces

Comparaison entre propriétés et indexeurs (Guide de programmation C#)

Article • 07/04/2023

Les indexeurs sont semblables aux propriétés. À l'exception des différences répertoriées dans le tableau suivant, toutes les règles définies pour les accesseurs des propriétés s'appliquent également aux accesseurs des indexeurs.

Propriété	**Indexation**
Permet aux méthodes d'être appelées comme si elles étaient des membres de données publics.	Permet aux éléments d'une collection interne d'un objet d'être accessibles à l'aide de la notation de tableau sur l'objet lui-même.
Accessible par le biais d'un nom simple.	Accessible par le biais d'un index.
Peut être un membre statique ou un membre d'instance.	Doit être un membre d'instance.
Un accesseur get d'une propriété n'a aucun paramètre.	Un accesseur `get` d'un indexeur possède la même liste de paramètres formels que l'indexeur.
Un accesseur set d'une propriété contient le paramètre `value` implicite.	Un accesseur `set` d'un indexeur possède la même liste de paramètres formels que l'indexeur, outre le paramètre value.
Prend en charge la syntaxe abrégée avec les propriétés implémentées automatiquement.	Prend en charge les membres expression-bodied pour les indexeurs get-only.

Voir aussi

- Guide de programmation C#
- Indexeurs
- Propriétés

Événements (Guide de programmation C#)

Article • 25/02/2023

Les événements permettent à une classe ou à un objet de notifier d'autres classes ou objets quand quelque chose de significatif se produit. La classe qui envoie (ou *déclenche*) l'événement est appelée *publieur* et les classes qui reçoivent (ou *gèrent*) l'événement sont appelées *abonnés*.

Dans une application C# Windows Forms ou web classique, vous vous abonnez à des événements déclenchés par des contrôles, comme des boutons et des zones de liste. Vous pouvez utiliser l'IDE Visual C# pour parcourir les événements publiés par un contrôle et sélectionner ceux que vous voulez gérer. L'IDE permet d'ajouter automatiquement une méthode de gestionnaire d'événements vide et le code pour vous abonner à l'événement. Pour plus d'informations, consultez Guide pratique pour s'abonner à des événements et s'en désabonner.

Vue d'ensemble des événements

Les événements ont les propriétés suivantes :

- Le publieur détermine quand un événement est déclenché ; les abonnés déterminent l'action entreprise en réponse à l'événement.
- Un événement peut avoir plusieurs abonnés. Un abonné peut gérer plusieurs événements provenant de plusieurs publieurs.
- Les événements qui n'ont aucun abonné ne sont jamais déclenchés.
- Les événements sont généralement utilisés pour signaler des actions de l'utilisateur, comme les clics de bouton ou les sélections de menu dans les interfaces utilisateur graphiques.
- Quand un événement a plusieurs abonnés, les gestionnaires d'événements sont appelées de façon synchrone quand un événement est déclenché. Pour appeler des événements de façon asynchrone, consultez Calling Synchronous Methods Asynchronously.
- Dans la bibliothèque de classes .NET, les événements sont basés sur le délégué EventHandler et la classe de base EventArgs.

Sections connexes

Pour plus d'informations, consultez les pages suivantes :

- Comment s'abonner à des événements et s'en désabonner
- Comment publier des événements conformes aux indications de .NET
- Comment déclencher des événements de la classe de base dans les classes dérivées
- Comment implémenter des événements d'interface
- Comment implémenter des accesseurs d'événement personnalisés

Spécification du langage C#

Pour plus d'informations, consultez Événements dans la Spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Chapitres proposés

Delegates, Events, and Lambda Expressions (Délégués, événements et expressions lambda) dans C# 3.0 Cookbook, Third Edition: More than 250 solutions for C# 3.0 programmers

Délégués et événements dans Apprentissage de C# 3.0 : Maîtriser les fondamentaux de C# 3.0

Voir aussi

- EventHandler
- Guide de programmation C#
- Délégués
- Création de gestionnaires d'événements dans les Windows Forms

Comment s'abonner à des événements et s'en désabonner (Guide de programmation C#)

Article • 10/03/2023

Vous vous abonnez à un événement publié par une autre classe lorsque vous voulez écrire du code personnalisé qui doit être appelé quand cet événement est déclenché. Par exemple, vous pouvez vous abonner à l'événement `click` d'un bouton pour permettre à votre application de réagir lorsque l'utilisateur clique sur le bouton.

Pour s'abonner aux événements à l'aide de l'IDE de Visual Studio

1. Si vous ne voyez pas la fenêtre **Propriétés**, en mode **Création**, cliquez sur le formulaire ou le contrôle pour lequel vous voulez créer un gestionnaire d'événements, puis sélectionnez **Propriétés**.

2. En haut de la fenêtre **Propriétés**, cliquez sur l'icône **Événements**.

3. Double-cliquez sur l'événement que vous voulez créer, par exemple l'événement `Load`.

   Visual C# crée une méthode de gestionnaire d'événements vide et l'ajoute à votre code. Vous pouvez également ajouter le code manuellement en mode **Code**. Par exemple, les lignes de code suivantes déclarent une méthode de gestionnaire d'événements qui est appelée lorsque la classe `Form` déclenche l'événement `Load`.

   C#

   ```
   private void Form1_Load(object sender, System.EventArgs e)
   {
       // Add your form load event handling code here.
   }
   ```

   La ligne de code qui est nécessaire pour s'abonner à l'événement est aussi générée automatiquement dans la méthode `InitializeComponent`, dans le fichier Form1.Designer.cs de votre projet. Elle ressemble à ceci :

   C#

```
this.Load += new System.EventHandler(this.Form1_Load);
```

Pour s'abonner aux événements par programmation

1. Définissez une méthode de gestionnaire d'événements dont la signature correspond à la signature du délégué de l'événement. Par exemple, si l'événement est basé sur le type délégué EventHandler, le code suivant représente le stub de méthode :

   C#

   ```
   void HandleCustomEvent(object sender, CustomEventArgs a)
   {
      // Do something useful here.
   }
   ```

2. Utilisez l'opérateur d'affectation d'addition (`+=`) pour attacher un gestionnaire d'événements à l'événement. Dans l'exemple suivant, nous allons supposer qu'un objet nommé `publisher` a un événement nommé `RaiseCustomEvent`. Notez que la classe d'abonné nécessite une référence à la classe d'éditeur pour s'abonner à ses événements.

   C#

   ```
   publisher.RaiseCustomEvent += HandleCustomEvent;
   ```

   Vous pouvez également utiliser une expression lambda pour spécifier un gestionnaire d'événements :

   C#

   ```
   public Form1()
   {
       InitializeComponent();
       this.Click += (s,e) =>
           {
               MessageBox.Show(((MouseEventArgs)e).Location.ToString());
           };
   }
   ```

Pour s'abonner aux événements à l'aide d'une fonction anonyme

Si vous n'avez pas besoin de vous désabonner d'un événement ultérieurement, vous pouvez utiliser l'opérateur d'assignation d'addition (`+=`) pour attacher une fonction anonyme en tant que gestionnaire d'événements. Dans l'exemple suivant, nous supposons qu'un objet nommé `publisher` a un événement nommé `RaiseCustomEvent`, et qu'une classe `CustomEventArgs` a également été définie pour contenir des informations d'événements spécialisés. Notez que la classe d'abonné nécessite une référence à `publisher` pour s'abonner à ses événements.

C#

```
publisher.RaiseCustomEvent += (object o, CustomEventArgs e) =>
{
  string s = o.ToString() + " " + e.ToString();
  Console.WriteLine(s);
};
```

Vous ne pouvez pas vous désabonner facilement d'un événement si vous avez utilisé une fonction anonyme pour vous y inscrire. Pour vous désabonner dans ce scénario, accédez au code dans lequel vous vous êtes abonné à l'événement, stockez la fonction anonyme dans une variable de délégué, puis ajoutez le délégué à l'événement. Nous recommandons de ne pas utiliser de fonctions anonymes pour vous abonner aux événements si vous devez vous en désabonner plus tard dans votre code. Pour plus d'informations sur les fonctions anonymes, consultez Expressions lambda.

Désabonnement

Pour éviter que votre gestionnaire d'événements ne soit appelé lorsque l'événement est déclenché, désabonnez-vous de l'événement. Pour empêcher les fuites de ressources, vous devez vous désabonner des événements avant d'éliminer un objet d'abonné. Tant que vous êtes abonné à un événement, le délégué de multidiffusion qui se trouve sous l'événement, dans l'objet de publication, comporte une référence au délégué qui encapsule le gestionnaire d'événements de l'abonné. Tant que l'objet de publication contient cette référence, le garbage collection ne supprime pas l'objet d'abonné.

Pour se désabonner d'un événement

- Utilisez l'opérateur d'assignation de soustraction (`-=`) pour vous désabonner d'un événement :

  C#

  ```
  publisher.RaiseCustomEvent -= HandleCustomEvent;
  ```

Lorsque tous les abonnés se sont désabonnés d'un événement, l'instance d'événement de la classe d'éditeur est définie sur `null`.

Voir aussi

- Événements
- event
- Comment publier des événements conformes aux indications de .NET
- - et -=, opérateurs
- + et +=, opérateurs

Guide pratique pour publier des événements conformes aux lignes directrices de .NET (Guide de programmation C#)

Article • 09/05/2023

La procédure suivante montre comment ajouter des événements qui respectent le modèle .NET standard à vos classes et structs. Tous les événements de la bibliothèque de classes .NET sont basés sur le délégué EventHandler, qui est défini comme suit :

C#

```
public delegate void EventHandler(object sender, EventArgs e);
```

> ⓘ **Notes**
>
> .NET Framework 2.0 comprend une version générique de ce délégué, **EventHandler<TEventArgs>**. Les exemples suivants montrent comment utiliser les deux versions.

Bien que les événements des classes que vous définissez puissent être basés sur n'importe quel type délégué valide, même les délégués qui retournent une valeur, il est généralement recommandé de baser vos événements sur le modèle .NET à l'aide de EventHandler, comme indiqué dans l'exemple suivant.

Le nom `EventHandler` peut entraîner un peu de confusion, car il ne gère pas réellement l'événement. Le EventHandler et EventHandler<TEventArgs> générique sont des types délégués. Une méthode ou une expression lambda dont la signature correspond à la définition du délégué est le *gestionnaire d'événements* et est appelée lorsque l'événement est déclenché.

Pour publier des événements basés sur le modèle EventHandler

1. (Ignorez cette étape et passez à l'étape 3a si vous n'avez pas besoin d'envoyer des données personnalisées avec votre événement.) Déclarez la classe pour vos données personnalisées dans une étendue visible à la fois par vos classes d'éditeur

et d'abonné. Ajoutez ensuite les membres nécessaires pour contenir vos données d'événement personnalisées. Dans cet exemple, une simple chaîne est retournée.

```
C#
```

```
public class CustomEventArgs : EventArgs
{
    public CustomEventArgs(string message)
    {
        Message = message;
    }

    public string Message { get; set; }
}
```

2. (Ignorez cette étape si vous utilisez la version générique de EventHandler<TEventArgs>.) Déclarez un délégué dans votre classe de publication. Donnez-lui un nom qui se termine par `EventHandler`. Le deuxième paramètre spécifie votre type `EventArgs` personnalisé.

```
C#
```

```
public delegate void CustomEventHandler(object sender, CustomEventArgs
args);
```

3. Déclarez l'événement dans votre classe de publication en effectuant l'une des étapes suivantes.

   a. Si vous n'avez aucune classe EventArgs personnalisée, votre type d'événement sera le délégué EventHandler non générique. Il est inutile de déclarer le délégué, car il l'est déjà dans l'espace de noms System inclus au moment où vous créez votre projet C#. Ajoutez le code suivant à votre classe de serveur de publication.

   ```
   C#
   ```

   ```
   public event EventHandler RaiseCustomEvent;
   ```

   b. Si vous utilisez la version non générique de EventHandler et que vous avez une classe personnalisée dérivée de EventArgs, déclarez votre événement à l'intérieur de votre classe de publication et utilisez votre délégué de l'étape 2 comme type.

   ```
   C#
   ```

```
public event CustomEventHandler RaiseCustomEvent;
```

c. Si vous utilisez la version générique, vous n'avez pas besoin de délégué personnalisé. Au lieu de cela, dans votre classe de publication, vous spécifiez votre type d'événement en tant que `EventHandler<CustomEventArgs>`, en insérant le nom de votre propre classe entre les crochets.

```
C#

public event EventHandler<CustomEventArgs> RaiseCustomEvent;
```

Exemple

L'exemple suivant est une illustration des étapes précédentes avec l'utilisation d'une classe EventArgs personnalisée et de EventHandler<TEventArgs> comme type d'événement.

```
C#

using System;

namespace DotNetEvents
{
    // Define a class to hold custom event info
    public class CustomEventArgs : EventArgs
    {
        public CustomEventArgs(string message)
        {
            Message = message;
        }

        public string Message { get; set; }
    }

    // Class that publishes an event
    class Publisher
    {
        // Declare the event using EventHandler<T>
        public event EventHandler<CustomEventArgs> RaiseCustomEvent;

        public void DoSomething()
        {
            // Write some code that does something useful here
            // then raise the event. You can also raise an event
            // before you execute a block of code.
            OnRaiseCustomEvent(new CustomEventArgs("Event triggered"));
        }
```

```
        // Wrap event invocations inside a protected virtual method
        // to allow derived classes to override the event invocation
behavior
        protected virtual void OnRaiseCustomEvent(CustomEventArgs e)
        {
            // Make a temporary copy of the event to avoid possibility of
            // a race condition if the last subscriber unsubscribes
            // immediately after the null check and before the event is
raised.
            EventHandler<CustomEventArgs> raiseEvent = RaiseCustomEvent;

            // Event will be null if there are no subscribers
            if (raiseEvent != null)
            {
                // Format the string to send inside the CustomEventArgs
parameter
                e.Message += $" at {DateTime.Now}";

                // Call to raise the event.
                raiseEvent(this, e);
            }
        }
    }

    //Class that subscribes to an event
    class Subscriber
    {
        private readonly string _id;

        public Subscriber(string id, Publisher pub)
        {
            _id = id;

            // Subscribe to the event
            pub.RaiseCustomEvent += HandleCustomEvent;
        }

        // Define what actions to take when the event is raised.
        void HandleCustomEvent(object sender, CustomEventArgs e)
        {
            Console.WriteLine($"{_id} received this message: {e.Message}");
        }
    }

    class Program
    {
        static void Main()
        {
            var pub = new Publisher();
            var sub1 = new Subscriber("sub1", pub);
            var sub2 = new Subscriber("sub2", pub);

            // Call the method that raises the event.
            pub.DoSomething();
```

```
            // Keep the console window open
            Console.WriteLine("Press any key to continue...");
            Console.ReadLine();
        }
    }
}
```

Voir aussi

- Delegate
- Guide de programmation C#
- Événements
- Délégués

Guide pratique pour déclencher les événements de la classe de base dans les classes dérivées (Guide de programmation C#)

Article • 10/05/2023

L'exemple suivant montre la méthode standard employée pour déclarer des événements dans une classe de base pour qu'ils puissent être déclenchés à partir de classes dérivées. Ce modèle est largement utilisé dans les classes Windows Forms des bibliothèques de classes .NET.

Lorsque vous créez une classe qui peut être utilisée comme classe de base pour d'autres classes, tenez compte du fait que les événements constituent un type spécial de délégué qui ne peut être appelé qu'à partir de la classe qui les a déclarés. Les classes dérivées ne peuvent pas appeler directement les événements qui sont déclarés dans la classe de base. Même si vous pouvez avoir besoin d'un événement qui ne peut être déclenché que par la classe de base, la plupart du temps, vous devez permettre à la classe dérivée d'appeler les événements de la classe de base. Pour ce faire, vous pouvez créer une méthode d'appel protégée dans la classe de base qui encapsule l'événement. En appelant ou en substituant cette méthode d'appel, les classes dérivées peuvent appeler indirectement l'événement.

> ⓘ **Notes**
>
> Ne déclarez pas les événements virtuels dans une classe de base et substituez-les dans une classe dérivée. Le compilateur C# ne gère pas correctement ce type d'événement et il n'est pas possible de prévoir si un abonné de l'événement dérivé s'abonnera à l'événement de classe de base.

Exemple

C#

```
namespace BaseClassEvents
{
    // Special EventArgs class to hold info about Shapes.
    public class ShapeEventArgs : EventArgs
```

```
    {
        public ShapeEventArgs(double area)
        {
            NewArea = area;
        }

        public double NewArea { get; }
    }

    // Base class event publisher
    public abstract class Shape
    {
        protected double _area;

        public double Area
        {
            get => _area;
            set => _area = value;
        }

        // The event. Note that by using the generic EventHandler<T> event
type
        // we do not need to declare a separate delegate type.
        public event EventHandler<ShapeEventArgs> ShapeChanged;

        public abstract void Draw();

        //The event-invoking method that derived classes can override.
        protected virtual void OnShapeChanged(ShapeEventArgs e)
        {
            // Safely raise the event for all subscribers
            ShapeChanged?.Invoke(this, e);
        }
    }

    public class Circle : Shape
    {
        private double _radius;

        public Circle(double radius)
        {
            _radius = radius;
            _area = 3.14 * _radius * _radius;
        }

        public void Update(double d)
        {
            _radius = d;
            _area = 3.14 * _radius * _radius;
            OnShapeChanged(new ShapeEventArgs(_area));
        }

        protected override void OnShapeChanged(ShapeEventArgs e)
        {
            // Do any circle-specific processing here.
```

```
            // Call the base class event invocation method.
            base.OnShapeChanged(e);
        }

        public override void Draw()
        {
            Console.WriteLine("Drawing a circle");
        }
    }

    public class Rectangle : Shape
    {
        private double _length;
        private double _width;

        public Rectangle(double length, double width)
        {
            _length = length;
            _width = width;
            _area = _length * _width;
        }

        public void Update(double length, double width)
        {
            _length = length;
            _width = width;
            _area = _length * _width;
            OnShapeChanged(new ShapeEventArgs(_area));
        }

        protected override void OnShapeChanged(ShapeEventArgs e)
        {
            // Do any rectangle-specific processing here.

            // Call the base class event invocation method.
            base.OnShapeChanged(e);
        }

        public override void Draw()
        {
            Console.WriteLine("Drawing a rectangle");
        }
    }

    // Represents the surface on which the shapes are drawn
    // Subscribes to shape events so that it knows
    // when to redraw a shape.
    public class ShapeContainer
    {
        private readonly List<Shape> _list;

        public ShapeContainer()
        {
            _list = new List<Shape>();
```

```
        }

        public void AddShape(Shape shape)
        {
            _list.Add(shape);

            // Subscribe to the base class event.
            shape.ShapeChanged += HandleShapeChanged;
        }

        // ...Other methods to draw, resize, etc.

        private void HandleShapeChanged(object sender, ShapeEventArgs e)
        {
            if (sender is Shape shape)
            {
                // Diagnostic message for demonstration purposes.
                Console.WriteLine($"Received event. Shape area is now
{e.NewArea}");

                // Redraw the shape here.
                shape.Draw();
            }
        }
    }

    class Test
    {
        static void Main()
        {
            //Create the event publishers and subscriber
            var circle = new Circle(54);
            var rectangle = new Rectangle(12, 9);
            var container = new ShapeContainer();

            // Add the shapes to the container.
            container.AddShape(circle);
            container.AddShape(rectangle);

            // Cause some events to be raised.
            circle.Update(57);
            rectangle.Update(7, 7);

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to continue...");
            Console.ReadKey();
        }
    }
}
/* Output:
        Received event. Shape area is now 10201.86
        Drawing a circle
        Received event. Shape area is now 49
        Drawing a rectangle
 */
```

Voir aussi

- Guide de programmation C#
- Événements
- Délégués
- Modificateurs d’accès
- Création de gestionnaires d'événements dans les Windows Forms

Guide pratique pour implémenter des événements d'interface (Guide de programmation C#)

Article • 10/03/2023

Une interface peut déclarer un événement. L'exemple suivant montre comment implémenter des événements d'interface dans une classe. En gros, les règles sont les mêmes que quand vous implémentez une propriété ou une méthode d'interface.

Pour implémenter des événements d'interface dans une classe

Déclarez l'événement dans votre classe, puis appelez-le aux emplacements appropriés.

C#

```
namespace ImplementInterfaceEvents
{
    public interface IDrawingObject
    {
        event EventHandler ShapeChanged;
    }
    public class MyEventArgs : EventArgs
    {
        // class members
    }
    public class Shape : IDrawingObject
    {
        public event EventHandler ShapeChanged;
        void ChangeShape()
        {
            // Do something here before the event…

            OnShapeChanged(new MyEventArgs(/*arguments*/));

            // or do something here after the event.
        }
        protected virtual void OnShapeChanged(MyEventArgs e)
        {
            ShapeChanged?.Invoke(this, e);
        }
    }

}
```

Exemple

L'exemple suivant montre comment gérer la situation peu courante dans laquelle votre classe hérite de plusieurs interfaces et chaque interface a un événement ayant le même nom. Dans ce cas, vous devez fournir une implémentation d'interface explicite pour au moins l'un des événements. Quand vous écrivez une implémentation d'interface explicite pour un événement, vous devez également écrire les accesseurs d'événement `add` et `remove`. Normalement, ces éléments sont fournis par le compilateur, mais dans le cas en question le compilateur ne peut pas les fournir.

En fournissant vos propres accesseurs, vous pouvez spécifier si les deux événements sont représentés par le même événement dans votre classe ou par des événements différents. Par exemple, si les événements doivent être déclenchés à des moments différents en fonction des spécifications de l'interface, vous pouvez associer chaque événement à une implémentation distincte dans votre classe. Dans l'exemple suivant, les abonnés déterminent l'événement `OnDraw` qu'il recevront en effectuant un cast de la référence de la forme en `IShape` ou `IDrawingObject`.

C#

```
namespace WrapTwoInterfaceEvents
{
    using System;

    public interface IDrawingObject
    {
        // Raise this event before drawing
        // the object.
        event EventHandler OnDraw;
    }
    public interface IShape
    {
        // Raise this event after drawing
        // the shape.
        event EventHandler OnDraw;
    }

    // Base class event publisher inherits two
    // interfaces, each with an OnDraw event
    public class Shape : IDrawingObject, IShape
    {
        // Create an event for each interface event
        event EventHandler PreDrawEvent;
        event EventHandler PostDrawEvent;

        object objectLock = new Object();

        // Explicit interface implementation required.
```

```
        // Associate IDrawingObject's event with
        // PreDrawEvent
        #region IDrawingObjectOnDraw
        event EventHandler IDrawingObject.OnDraw
        {
            add
            {
                lock (objectLock)
                {
                    PreDrawEvent += value;
                }
            }
            remove
            {
                lock (objectLock)
                {
                    PreDrawEvent -= value;
                }
            }
        }
        #endregion
        // Explicit interface implementation required.
        // Associate IShape's event with
        // PostDrawEvent
        event EventHandler IShape.OnDraw
        {
            add
            {
                lock (objectLock)
                {
                    PostDrawEvent += value;
                }
            }
            remove
            {
                lock (objectLock)
                {
                    PostDrawEvent -= value;
                }
            }
        }

        // For the sake of simplicity this one method
        // implements both interfaces.
        public void Draw()
        {
            // Raise IDrawingObject's event before the object is drawn.
            PreDrawEvent?.Invoke(this, EventArgs.Empty);

            Console.WriteLine("Drawing a shape.");

            // Raise IShape's event after the object is drawn.
            PostDrawEvent?.Invoke(this, EventArgs.Empty);
        }
    }
```

```
    public class Subscriber1
    {
        // References the shape object as an IDrawingObject
        public Subscriber1(Shape shape)
        {
            IDrawingObject d = (IDrawingObject)shape;
            d.OnDraw += d_OnDraw;
        }

        void d_OnDraw(object sender, EventArgs e)
        {
            Console.WriteLine("Sub1 receives the IDrawingObject event.");
        }
    }
    // References the shape object as an IShape
    public class Subscriber2
    {
        public Subscriber2(Shape shape)
        {
            IShape d = (IShape)shape;
            d.OnDraw += d_OnDraw;
        }

        void d_OnDraw(object sender, EventArgs e)
        {
            Console.WriteLine("Sub2 receives the IShape event.");
        }
    }

    public class Program
    {
        static void Main(string[] args)
        {
            Shape shape = new Shape();
            Subscriber1 sub = new Subscriber1(shape);
            Subscriber2 sub2 = new Subscriber2(shape);
            shape.Draw();

            // Keep the console window open in debug mode.
            System.Console.WriteLine("Press any key to exit.");
            System.Console.ReadKey();
        }
    }
}
/* Output:
    Sub1 receives the IDrawingObject event.
    Drawing a shape.
    Sub2 receives the IShape event.
*/
```

Voir aussi

- Guide de programmation C#
- Événements
- Délégués
- Implémentation d'interface explicite
- Comment déclencher des événements de la classe de base dans les classes dérivées

Comment : implémenter des accesseurs d'événement personnalisés (Guide de programmation C#)

Article • 10/03/2023

Un événement constitue un genre spécial de délégué multicast qui peut être appelé uniquement à partir de la classe dans laquelle il est déclaré. Le code client s'abonne à l'événement en fournissant une référence à une méthode qui doit être appelée quand l'événement est déclenché. Ces méthodes sont ajoutées à la liste d'invocation du délégué par le biais des accesseurs d'événement, qui ressemblent aux accesseurs de propriété, sauf que les accesseurs d'événement sont nommés `add` et `remove`. Dans la plupart des cas, vous n'avez pas à fournir d'accesseurs d'événement personnalisés. Quand aucun accesseur d'événement personnalisé n'est fourni dans votre code, le compilateur les ajoute automatiquement. Toutefois, vous devez parfois fournir un comportement personnalisé. Ce genre de cas est illustré dans la rubrique Guide pratique pour implémenter des événements d'interface.

Exemple

L'exemple suivant indique comment implémenter des accesseurs d'événement add et remove. Même si vous pouvez remplacer le code à l'intérieur des accesseurs, nous vous recommandons de verrouiller l'événement avant d'ajouter ou de supprimer une nouvelle méthode de gestionnaire d'événements.

C#

```
event EventHandler IDrawingObject.OnDraw
{
    add
    {
        lock (objectLock)
        {
            PreDrawEvent += value;
        }
    }
    remove
    {
        lock (objectLock)
        {
            PreDrawEvent -= value;
        }
```

```
    }
}
```

Voir aussi

- Événements
- event

Paramètres de type générique (Guide de programmation C#)

Article • 17/08/2023

Dans une définition de méthode ou de type générique, le paramètre de type représente un espace réservé pour un type spécifié par le client au moment de créer une instance du type générique. Une classe générique, telle que `GenericList<T>` répertoriée dans Introduction aux génériques, ne peut pas être utilisée en l'état car il ne s'agit pas vraiment d'un type, mais plutôt d'un modèle pour un type. Pour utiliser `GenericList<T>`, le code client doit déclarer et instancier un type construit en spécifiant un argument de type à l'intérieur de crochets pointus. L'argument de type pour cette classe particulière peut être tout type reconnu par le compilateur. Il est possible de créer un nombre quelconque d'instances de type construit, chacune avec un argument de type différent, comme suit :

C#

```
GenericList<float> list1 = new GenericList<float>();
GenericList<ExampleClass> list2 = new GenericList<ExampleClass>();
GenericList<ExampleStruct> list3 = new GenericList<ExampleStruct>();
```

Dans chacune de ces instances de `GenericList<T>`, chaque occurrence de `T` dans la classe est remplacée à l'exécution par l'argument de type. Par cette substitution, nous avons créé trois objets distincts de type sécurisé et efficaces à l'aide d'une seule définition de classe. Pour plus d'informations sur la façon dont cette substitution est effectuée par le CLR, consultez Génériques dans le runtime.

Vous pouvez découvrir les conventions de dénomination des paramètres de type générique dans l'article sur les conventions de dénomination.

Voir aussi

- System.Collections.Generic
- Guide de programmation C#
- Génériques
- Différences entre les modèles C++ et les génériques C#

Contraintes sur les paramètres de type (Guide de programmation C#)

Article • 10/05/2023

Les contraintes informent le compilateur sur les fonctionnalités que doit avoir un argument de type. Sans contrainte, l'argument de type peut être n'importe quel type. Le compilateur peut seulement deviner les membres de System.Object, qui est la classe de base par excellence de tous les types .NET. Pour plus d'informations, consultez Pourquoi utiliser des contraintes. Si le code client utilise un type qui ne satisfait pas à une contrainte, le compilateur émet une erreur. Les contraintes sont spécifiées à l'aide du mot clé contextuel `where`. Le tableau suivant liste plusieurs types de contrainte :

Contrainte	Description
`where T : struct`	L'argument de type doit être un type valeur non-nullable. Pour plus d'informations sur les types valeur pouvant accepter la valeur Null, consultez Types valeur pouvant accepter la valeur Null. Étant donné que tous les types valeur ont un constructeur sans paramètre accessible, la contrainte `struct` implique la contrainte `new()` et ne peut pas être combinée avec la contrainte `new()`. Vous ne pouvez pas combiner la contrainte `struct` avec la contrainte `unmanaged`.
`where T : class`	L'argument de type doit être un type référence. Cette contrainte s'applique également à tous les types de classe, d'interface, de délégué ou de tableau. Dans un contexte pouvant accepter la valeur Null, `T` doit être un type référence non nullable.
`where T : class?`	L'argument de type doit être un type référence, pouvant accepter la valeur Null ou non nullable. Cette contrainte s'applique également à tous les types de classe, d'interface, de délégué ou de tableau.
`where T : notnull`	L'argument de type doit être un type non-nullable. L'argument peut être un type référence non nullable ou un type valeur non nullable.
`where T : default`	Cette contrainte résout l'ambiguïté lorsque vous devez spécifier un paramètre de type sans contrainte lorsque vous remplacez une méthode ou fournissez une implémentation d'interface explicite. La contrainte `default` implique la méthode de base sans la contrainte `class` ou `struct`. Pour plus d'informations, consultez la proposition de spécification default de contrainte.
`where T : unmanaged`	L'argument de type doit être un type non généré non-nullable. La contrainte `unmanaged` implique la contrainte `struct` et ne peut pas être combinée avec les contraintes `struct` ou `new()`.

Contrainte	Description
`where T : new()`	L'argument de type doit avoir un constructeur sans paramètre public. Quand vous utilisez la contrainte `new()` avec d'autres contraintes, elle doit être spécifiée en dernier. La contrainte `new()` ne peut pas être combinée avec les contraintes `struct` et `unmanaged`.
`where T :` *<nom de la classe de base>*	L'argument de type doit être la classe de base spécifiée ou en dériver. Dans un contexte pouvant accepter la valeur Null, `T` doit être un type référence non nullable dérivé de la classe de base spécifiée.
`where T :` *<nom de la classe de base >?*	L'argument de type doit être la classe de base spécifiée ou en dériver. Dans un contexte pouvant accepter la valeur Null, `T` peut être un type pouvant accepter la valeur Null ou non nullable dérivé de la classe de base spécifiée.
`where T :` *<nom de l'interface>*	L'argument de type doit être ou implémenter l'interface spécifiée. Plusieurs contraintes d'interface peuvent être spécifiées. L'interface qui impose les contraintes peut également être générique. Dans un contexte pouvant accepter la valeur Null, `T` doit être un type non nullable qui implémente l'interface spécifiée.
`where T :` *<nom de l'interface >?*	L'argument de type doit être ou implémenter l'interface spécifiée. Plusieurs contraintes d'interface peuvent être spécifiées. L'interface qui impose les contraintes peut également être générique. Dans un contexte pouvant accepter la valeur Null, `T` peut être un type référence pouvant accepter la valeur Null, un type référence non nullable ou un type valeur. `T` peut ne pas être un type valeur pouvant accepter la valeur Null.
`where T : U`	L'argument de type fourni pour `T` doit être l'argument fourni pour `U` ou en dériver. Dans un contexte pouvant accepter la valeur Null, si `U` est un type référence non nullable, `T` doit être un type référence non nullable. Si `U` est un type référence pouvant accepter la valeur Null, `T` peut accepter la valeur Null ou être non nullable.

Pourquoi utiliser des contraintes

Les contraintes spécifient les fonctionnalités et les attentes d'un paramètre de type. La déclaration de ces contraintes signifie que vous pouvez utiliser les opérations et les appels de méthode du type de contrainte. Si votre classe ou votre méthode générique utilise une opération sur les membres génériques au-delà de l'assignation simple ou l'appel de toute méthode non prise en charge par System.Object, vous devez appliquer des contraintes au paramètre de type. Par exemple, la contrainte de classe de base indique au compilateur que seuls les objets de ce type ou dérivés de ce type seront utilisés comme arguments de type. Une fois que le compilateur a cette garantie, il peut autoriser les méthodes de ce type à être appelées dans la classe générique. L'exemple

de code suivant illustre la fonctionnalité que vous pouvez ajouter à la classe `GenericList<T>` (dans Introduction aux génériques) en appliquant une contrainte de classe de base.

C#

```
public class Employee
{
    public Employee(string name, int id) => (Name, ID) = (name, id);
    public string Name { get; set; }
    public int ID { get; set; }
}

public class GenericList<T> where T : Employee
{
    private class Node
    {
        public Node(T t) => (Next, Data) = (null, t);

        public Node? Next { get; set; }
        public T Data { get; set; }
    }

    private Node? head;

    public void AddHead(T t)
    {
        Node n = new Node(t) { Next = head };
        head = n;
    }

    public IEnumerator<T> GetEnumerator()
    {
        Node? current = head;

        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }

    public T? FindFirstOccurrence(string s)
    {
        Node? current = head;
        T? t = null;

        while (current != null)
        {
            //The constraint enables access to the Name property.
            if (current.Data.Name == s)
            {
                t = current.Data;
```

```
                break;
            }
            else
            {
                current = current.Next;
            }
        }
        return t;
    }
}
```

La contrainte permet à la classe générique d'utiliser la propriété `Employee.Name`. La contrainte spécifie que tous les éléments de type `T` sont soit un objet `Employee`, soit un objet qui hérite de `Employee`, et rien d'autre.

Plusieurs contraintes peuvent être appliquées au même paramètre de type, et les contraintes elles-mêmes peuvent être des types génériques, comme suit :

C#

```
class EmployeeList<T> where T : Employee, IEmployee, System.IComparable<T>,
new()
{
    // ...
}
```

En appliquant la contrainte `where T : class`, évitez d'utiliser les opérateurs `==` et `!=` sur le paramètre de type, car ces opérateurs testent uniquement l'identité des références, et non l'égalité des valeurs. Ce comportement se produit même si ces opérateurs sont surchargés dans un type qui est utilisé comme argument. Le code suivant illustre ce point ; la sortie a la valeur false même si la classe String surcharge l'opérateur `==`.

C#

```
public static void OpEqualsTest<T>(T s, T t) where T : class
{
    System.Console.WriteLine(s == t);
}

private static void TestStringEquality()
{
    string s1 = "target";
    System.Text.StringBuilder sb = new System.Text.StringBuilder("target");
    string s2 = sb.ToString();
    OpEqualsTest<string>(s1, s2);
}
```

Au moment de la compilation, le compilateur sait uniquement que `T` est un type référence et doit utiliser les opérateurs par défaut qui sont valides pour tous les types référence. Si vous devez tester l'égalité des valeurs, il est recommandé d'appliquer également la contrainte `where T : IEquatable<T>` ou `where T : IComparable<T>` et d'implémenter l'interface dans toute classe qui sera utilisée pour construire la classe générique.

Utilisation de contraintes dans plusieurs paramètres

Vous pouvez appliquer des contraintes à plusieurs paramètres et plusieurs contraintes à un seul paramètre, comme indiqué dans l'exemple suivant :

C#

```
class Base { }
class Test<T, U>
    where U : struct
    where T : Base, new()
{ }
```

Paramètres de type unbounded

Les paramètres de type qui n'ont aucune contrainte, tels que T dans la classe publique `SampleClass<T>{}`, sont appelés paramètres de type unbounded. Les paramètres de type unbounded obéissent aux règles suivantes :

- Les opérateurs `!=` et `==` ne peuvent pas être utilisés, car il n'est pas garanti que l'argument de type concret les prendra en charge.
- Ils peuvent être convertis vers et depuis `System.Object` ou être explicitement convertis vers tout type d'interface.
- Vous pouvez les comparer à null. Si un paramètre unbounded est comparé à `null`, la comparaison retourne toujours la valeur false si l'argument de type est un type valeur.

Paramètres de type en tant que contraintes

L'utilisation d'un paramètre de type générique comme contrainte est utile quand une fonction membre dotée de son propre paramètre de type doit contraindre ce paramètre au paramètre du type conteneur, comme le montre l'exemple suivant :

C#

```
public class List<T>
{
    public void Add<U>(List<U> items) where U : T {/*...*/}
}
```

Dans l'exemple précédent, `T` est une contrainte de type dans le contexte de la méthode `Add` et un paramètre de type unbounded dans le contexte de la classe `List`.

Les paramètres de type peuvent également être utilisés comme contraintes dans les définitions de classes génériques. Le paramètre de type doit être déclaré entre crochets pointus, ainsi que tous les autres paramètres de type :

C#

```
//Type parameter V is used as a type constraint.
public class SampleClass<T, U, V> where T : V { }
```

L'utilité des paramètres de type en tant que contraintes avec les classes génériques est limitée, car le compilateur ne peut rien deviner à propos du paramètre de type en dehors du fait qu'il dérive de `System.Object`. Utilisez des paramètres de type en tant que contraintes sur les classes génériques dans les scénarios dans lesquels vous souhaitez mettre en application une relation d'héritage entre deux paramètres de type.

Contrainte `notnull`

Vous pouvez utiliser la contrainte `notnull` pour spécifier que l'argument de type doit être un type valeur non nullable ou un type référence non nullable. Contrairement à la plupart d'autres contraintes, si un argument de type enfreint la contrainte `notnull`, le compilateur génère un avertissement au lieu d'une erreur.

La contrainte `notnull` a un effet uniquement lorsqu'elle est utilisée dans un contexte pouvant accepter la valeur Null. Si vous ajoutez la contrainte `notnull` dans un contexte inconscient pouvant accepter la valeur Null, le compilateur ne génère pas d'avertissements ou d'erreurs pour les violations de la contrainte.

Contrainte `class`

La contrainte `class` dans un contexte pouvant accepter la valeur Null spécifie que l'argument de type doit être un type référence non nullable. Dans un contexte pouvant

accepter la valeur Null, lorsqu'un argument de type est un type référence pouvant accepter la valeur Null, le compilateur génère un avertissement.

Contrainte `default`

L'ajout de types référence pouvant accepter la valeur Null complique l'utilisation de `T?` dans un type ou une méthode générique. `T?` peut être utilisé avec la contrainte `struct` ou `class`, mais l'une d'elles doit être présente. Lorsque la contrainte `class` a été utilisée, `T?` fait référence au type de référence pouvant accepter la valeur nulle pour `T`. À compter de C# 9, `T?` peut être utilisé quand aucune contrainte n'est appliquée. Dans ce cas, `T?` est interprété comme `T?` pour les types valeur et les types référence. Toutefois, si `T` est une instance de Nullable<T>, `T?` est identique à `T`. En d'autres termes, il ne devient pas `T??`.

Étant donné que `T?` peut désormais être utilisé sans contrainte `class` ou `struct`, des ambiguïtés peuvent survenir dans les remplacements ou les implémentations d'interface explicites. Dans ces deux cas, le remplacement n'inclut pas les contraintes, mais les hérite de la classe de base. Lorsque la classe de base n'applique pas la contrainte `class` ou `struct`, les classes dérivées doivent spécifier un remplacement qui s'applique à la méthode de base sans l'une ou l'autre contrainte. C'est à ce moment que la méthode dérivée applique la contrainte `default`. La contrainte `default` ne clarifie *ni* la contrainte `class` ni la contrainte `struct`.

Contrainte non managée

Vous pouvez utiliser la contrainte `unmanaged` pour spécifier que le paramètre de type doit être un type non managé non-nullable. La contrainte `unmanaged` vous permet d'écrire des routines réutilisables à appliquer aux types qui peuvent être manipulés comme blocs de mémoire, comme illustré dans l'exemple suivant :

C#

```
unsafe public static byte[] ToByteArray<T>(this T argument) where T :
unmanaged
{
    var size = sizeof(T);
    var result = new Byte[size];
    Byte* p = (byte*)&argument;
    for (var i = 0; i < size; i++)
        result[i] = *p++;
    return result;
}
```

La méthode précédente doit être compilée dans un contexte `unsafe`, car elle utilise l'opérateur `sizeof` sur un type qui n'est pas connu pour être un type intégré. Sans la contrainte `unmanaged`, l'opérateur `sizeof` n'est pas disponible.

La contrainte `unmanaged` implique la contrainte `struct` et ne peut pas être combinée avec. Étant donné que la contrainte `struct` implique la contrainte `new()`, mais la contrainte `unmanaged` ne peut pas être combinée avec la contrainte `new()`.

Contraintes de délégué

Vous pouvez utiliser System.Delegate ou System.MulticastDelegate comme contrainte de classe de base. Le CLR a toujours autorisé cette contrainte, contrairement au langage C#. La contrainte `System.Delegate` vous permet d'écrire du code qui fonctionne avec les délégués en mode type sécurisé. Le code suivant définit une méthode d'extension qui combine deux délégués du moment qu'ils sont du même type :

C#

```
public static TDelegate? TypeSafeCombine<TDelegate>(this TDelegate source,
TDelegate target)
    where TDelegate : System.Delegate
    => Delegate.Combine(source, target) as TDelegate;
```

Vous pouvez utiliser la méthode ci-dessus pour combiner des délégués qui sont du même type :

C#

```
Action first = () => Console.WriteLine("this");
Action second = () => Console.WriteLine("that");

var combined = first.TypeSafeCombine(second);
combined!();

Func<bool> test = () => true;
// Combine signature ensures combined delegates must
// have the same type.
//var badCombined = first.TypeSafeCombine(test);
```

Si vous supprimez les commentaires de la dernière ligne, il ne sera pas compilé. `first` et `test` sont tous deux des types délégués, mais des types délégués différents.

Contraintes d'enum

Vous pouvez également spécifier le type System.Enum comme contrainte de classe de base. Le CLR a toujours autorisé cette contrainte, contrairement au langage C#. Les génériques utilisant `System.Enum` fournissent une programmation de type sécurisé aux résultats de cache issus de l'utilisation de méthodes statiques dans `System.Enum`. L'exemple suivant recherche toutes les valeurs valides d'un type enum, puis génère un dictionnaire qui mappe ces valeurs à sa représentation sous forme de chaîne.

C#

```
public static Dictionary<int, string> EnumNamedValues<T>() where T :
System.Enum
{
    var result = new Dictionary<int, string>();
    var values = Enum.GetValues(typeof(T));

    foreach (int item in values)
        result.Add(item, Enum.GetName(typeof(T), item)!);
    return result;
}
```

`Enum.GetValues` et `Enum.GetName` utilisent la réflexion, ce qui a des répercussions sur les performances. Vous pouvez appeler `EnumNamedValues` pour générer une collection qui est mise en cache et réutilisée, plutôt que de répéter les appels qui nécessitent la réflexion.

Vous pouvez l'utiliser comme montré dans l'exemple suivant pour créer un enum et générer un dictionnaire de ses valeurs et de ses noms :

C#

```
enum Rainbow
{
    Red,
    Orange,
    Yellow,
    Green,
    Blue,
    Indigo,
    Violet
}
```

C#

```
var map = EnumNamedValues<Rainbow>();

foreach (var pair in map)
    Console.WriteLine($"{pair.Key}:\t{pair.Value}");
```

Arguments de type implémentant l'interface déclarée

Certains scénarios nécessitent qu'un argument fourni pour un paramètre de type implémente cette interface. Par exemple :

C#

```
public interface IAdditionSubtraction<T> where T : IAdditionSubtraction<T>
{
    public abstract static T operator +(T left, T right);
    public abstract static T operator -(T left, T right);
}
```

Ce modèle permet au compilateur C# de déterminer le type contenant pour les opérateurs surchargés, ou toute `static virtual` méthode ou `static abstract`. Il fournit la syntaxe afin que les opérateurs d'addition et de soustraction puissent être définis sur un type contenant. Sans cette contrainte, les paramètres et les arguments doivent être déclarés comme interface, plutôt que comme paramètre de type :

C#

```
public interface IAdditionSubtraction<T> where T : IAdditionSubtraction<T>
{
    public abstract static IAdditionSubtraction<T> operator +(
        IAdditionSubtraction<T> left,
        IAdditionSubtraction<T> right);

    public abstract static IAdditionSubtraction<T> operator -(
        IAdditionSubtraction<T> left,
        IAdditionSubtraction<T> right);
}
```

La syntaxe précédente exigerait que les implémenteurs utilisent l'implémentation d'interface explicite pour ces méthodes. La fourniture de la contrainte supplémentaire permet à l'interface de définir les opérateurs en termes de paramètres de type. Les types qui implémentent l'interface peuvent implémenter implicitement les méthodes d'interface.

Voir aussi

- System.Collections.Generic
- Guide de programmation C#
- Introduction aux génériques
- Classes génériques
- new, contrainte

Classes génériques (guide de programmation C#)

Article • 10/05/2023

Les classes génériques encapsulent des opérations qui ne sont pas spécifiques à un type de données particulier. Les classes génériques sont le plus souvent utilisées avec les collections telles que les listes liées, les tables de hachage, les piles, les files d'attente, les arborescences, etc. Les opérations telles que l'ajout et la suppression d'éléments dans la collection sont exécutées fondamentalement de la même manière quel que soit le type des données stockées.

Pour la plupart des scénarios qui nécessitent des classes de collections, l'approche recommandée consiste à utiliser celles fournies dans la bibliothèque de classes .NET. Pour plus d'informations sur l'utilisation de ces classes, consultez Collections génériques dans .NET.

En général, vous créez des classes génériques en commençant par une classe concrète existante et en changeant des types en paramètres de type individuellement jusqu'à ce que vous atteigniez l'équilibre optimal au niveau de la généralisation et de la facilité d'utilisation. Lors de la création de vos propres classes génériques, les aspects importants à prendre en considération sont notamment les suivants :

- Quels types généraliser en paramètres de type.

  En règle générale, plus le nombre de types que vous pouvez paramétrer est important, plus votre code est souple et réutilisable. Toutefois, une généralisation excessive peut produire un code qui est difficile à lire ou à comprendre par d'autres développeurs.

- Quelles contraintes, le cas échéant, appliquer aux paramètres de type (consultez Contraintes sur les paramètres de type).

  Il est pertinent d'appliquer le maximum de contraintes possible vous permettant néanmoins de gérer les types que vous devez gérer. Par exemple, si vous savez que votre classe générique est destinée uniquement à être utilisée avec les types référence, appliquez la contrainte de classe. Cela empêchera l'utilisation involontaire de votre classe avec les types valeur et vous permettra d'utiliser l'opérateur `as` sur `T` et de rechercher les valeurs nulles.

- S'il convient de prendre en compte le comportement générique dans les sous-classes et les classes de base.

Les classes génériques pouvant servir de classes de base, les considérations de conception qui s'appliquent ici sont les mêmes que pour les classes non génériques. Consultez les règles relatives à l'héritage à partir des classes de base génériques plus loin dans cette rubrique.

- S'il convient d'implémenter une ou plusieurs interfaces génériques.

  Par exemple, si vous concevez une classe qui sera utilisée pour créer des éléments dans une collection basée sur les génériques, vous pouvez avoir besoin d'implémenter une interface comme IComparable<T>, où `T` est le type de votre classe.

Pour obtenir un exemple d'une classe générique simple, consultez Introduction aux génériques.

Les règles pour les paramètres de type et les contraintes ont plusieurs conséquences sur le comportement de classe générique, surtout en ce qui concerne l'héritage et l'accessibilité des membres. Avant de continuer, vous devez comprendre quelques termes. Pour une classe générique `Node<T>,`, le code client peut référencer la classe en spécifiant un argument de type pour créer un type construit fermé (`Node<int>`) ou en laissant le paramètre de type non spécifié, par exemple lorsque vous spécifiez une classe de base générique, pour créer un type construit ouvert (`Node<T>`). Les classes génériques peuvent hériter de classes de base concrètes, construites fermées ou construites ouvertes :

C#

```
class BaseNode { }
class BaseNodeGeneric<T> { }

// concrete type
class NodeConcrete<T> : BaseNode { }

//closed constructed type
class NodeClosed<T> : BaseNodeGeneric<int> { }

//open constructed type
class NodeOpen<T> : BaseNodeGeneric<T> { }
```

Les classes non génériques, également appelées concrètes, peuvent hériter des classes de base construites fermées, mais pas des classes construites ouvertes ni des paramètres de type parce que le code client ne peut en aucune façon fournir au moment de l'exécution l'argument de type requis pour instancier la classe de base.

C#

```
//No error
class Node1 : BaseNodeGeneric<int> { }

//Generates an error
//class Node2 : BaseNodeGeneric<T> {}

//Generates an error
//class Node3 : T {}
```

Les classes génériques qui héritent de types construits ouverts doivent fournir des arguments de type pour tout paramètre de type de classe de base qui n'est pas partagé par la classe qui hérite, comme cela est indiqué dans le code suivant :

C#

```
class BaseNodeMultiple<T, U> { }

//No error
class Node4<T> : BaseNodeMultiple<T, int> { }

//No error
class Node5<T, U> : BaseNodeMultiple<T, U> { }

//Generates an error
//class Node6<T> : BaseNodeMultiple<T, U> {}
```

Les classes génériques qui héritent des types construits ouverts doivent spécifier des contraintes qui sont un sur-ensemble, ou qui impliquent, des contraintes sur le type de base :

C#

```
class NodeItem<T> where T : System.IComparable<T>, new() { }
class SpecialNodeItem<T> : NodeItem<T> where T : System.IComparable<T>,
new() { }
```

Les types génériques peuvent utiliser plusieurs paramètres de type et contraintes, comme suit :

C#

```
class SuperKeyType<K, V, U>
    where U : System.IComparable<U>
    where V : new()
{ }
```

Les types construits ouverts et les types construits fermés peuvent être utilisés comme paramètres de méthode :

```
C#
```

```
void Swap<T>(List<T> list1, List<T> list2)
{
    //code to swap items
}

void Swap(List<int> list1, List<int> list2)
{
    //code to swap items
}
```

Si une classe générique implémente une interface, toutes les instances de cette classe peuvent être castées à cette interface.

Les classes génériques sont indifférentes. En d'autres termes, si un paramètre d'entrée spécifie un `List<BaseClass>`, vous obtiendrez une erreur de compilation si vous essayez de fournir un `List<DerivedClass>`.

Voir aussi

- System.Collections.Generic
- Guide de programmation C#
- Génériques
- Enregistrement de l'état des énumérateurs
- An Inheritance Puzzle, Part One

Interfaces génériques (guide de programmation C#)

Article • 09/05/2023

Il est souvent utile de définir des interfaces pour les classes de collections génériques, ou pour les classes génériques qui représentent des éléments dans la collection. Pour éviter les opérations boxing et unboxing sur les types valeurs, il est préférable d'utiliser des interfaces génériques, telles que IComparable<T>, sur des classes génériques. La bibliothèque de classes .NET définit plusieurs interfaces génériques à utiliser avec les classes de collections dans l'espace de noms System.Collections.Generic. Pour plus d'informations sur ces interfaces, consultez Interfaces génériques.

Quand une interface est spécifiée comme une contrainte sur un paramètre de type, seuls les types qui implémentent l'interface peuvent être utilisés. L'exemple de code suivant illustre une classe `SortedList<T>` qui dérive de la classe `GenericList<T>`. Pour plus d'informations, consultez Introduction aux génériques. `SortedList<T>` ajoute la contrainte `where T : IComparable<T>`. Cette contrainte permet à la méthode `BubbleSort` dans `SortedList<T>` d'utiliser la méthode générique CompareTo sur les éléments de liste. Dans cet exemple, les éléments de liste sont une classe simple, `Person`, qui implémente `IComparable<Person>`.

C#

```
//Type parameter T in angle brackets.
public class GenericList<T> : System.Collections.Generic.IEnumerable<T>
{
    protected Node head;
    protected Node current = null;

    // Nested class is also generic on T
    protected class Node
    {
        public Node next;
        private T data;  //T as private member datatype

        public Node(T t)  //T used in non-generic constructor
        {
            next = null;
            data = t;
        }

        public Node Next
        {
            get { return next; }
            set { next = value; }
```

```
        }

        public T Data  //T as return type of property
        {
            get { return data; }
            set { data = value; }
        }
    }

    public GenericList()  //constructor
    {
        head = null;
    }

    public void AddHead(T t)  //T as method parameter type
    {
        Node n = new Node(t);
        n.Next = head;
        head = n;
    }

    // Implementation of the iterator
    public System.Collections.Generic.IEnumerator<T> GetEnumerator()
    {
        Node current = head;
        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }

    // IEnumerable<T> inherits from IEnumerable, therefore this class
    // must implement both the generic and non-generic versions of
    // GetEnumerator. In most cases, the non-generic method can
    // simply call the generic method.
    System.Collections.IEnumerator
System.Collections.IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}

public class SortedList<T> : GenericList<T> where T : System.IComparable<T>
{
    // A simple, unoptimized sort algorithm that
    // orders list elements from lowest to highest:

    public void BubbleSort()
    {
        if (null == head || null == head.Next)
        {
            return;
        }
        bool swapped;
```

```
        do
        {
            Node previous = null;
            Node current = head;
            swapped = false;

            while (current.next != null)
            {
                //  Because we need to call this method, the SortedList
                //  class is constrained on IComparable<T>
                if (current.Data.CompareTo(current.next.Data) > 0)
                {
                    Node tmp = current.next;
                    current.next = current.next.next;
                    tmp.next = current;

                    if (previous == null)
                    {
                        head = tmp;
                    }
                    else
                    {
                        previous.next = tmp;
                    }
                    previous = tmp;
                    swapped = true;
                }
                else
                {
                    previous = current;
                    current = current.next;
                }
            }
        } while (swapped);
    }
}

// A simple class that implements IComparable<T> using itself as the
// type argument. This is a common design pattern in objects that
// are stored in generic lists.
public class Person : System.IComparable<Person>
{
    string name;
    int age;

    public Person(string s, int i)
    {
        name = s;
        age = i;
    }

    // This will cause list elements to be sorted on age values.
    public int CompareTo(Person p)
    {
```

```
        return age - p.age;
    }

    public override string ToString()
    {
        return name + ":" + age;
    }

    // Must implement Equals.
    public bool Equals(Person p)
    {
        return (this.age == p.age);
    }
}

public class Program
{
    public static void Main()
    {
        //Declare and instantiate a new generic SortedList class.
        //Person is the type argument.
        SortedList<Person> list = new SortedList<Person>();

        //Create name and age values to initialize Person objects.
        string[] names = new string[]
        {
            "Franscoise",
            "Bill",
            "Li",
            "Sandra",
            "Gunnar",
            "Alok",
            "Hiroyuki",
            "Maria",
            "Alessandro",
            "Raul"
        };

        int[] ages = new int[] { 45, 19, 28, 23, 18, 9, 108, 72, 30, 35 };

        //Populate the list.
        for (int x = 0; x < 10; x++)
        {
            list.AddHead(new Person(names[x], ages[x]));
        }

        //Print out unsorted list.
        foreach (Person p in list)
        {
            System.Console.WriteLine(p.ToString());
        }
        System.Console.WriteLine("Done with unsorted list");

        //Sort the list.
        list.BubbleSort();
```

```
        //Print out sorted list.
        foreach (Person p in list)
        {
            System.Console.WriteLine(p.ToString());
        }
        System.Console.WriteLine("Done with sorted list");
    }
}
```

Plusieurs interfaces peuvent être spécifiées en tant que contraintes sur un seul type, comme suit :

C#

```
class Stack<T> where T : System.IComparable<T>, IEnumerable<T>
{
}
```

Une interface peut définir plusieurs paramètres de type, comme suit :

C#

```
interface IDictionary<K, V>
{
}
```

Les règles d'héritage qui s'appliquent aux classes s'appliquent également aux interfaces :

C#

```
interface IMonth<T> { }

interface IJanuary : IMonth<int> { }  //No error
interface IFebruary<T> : IMonth<int> { }  //No error
interface IMarch<T> : IMonth<T> { }    //No error
                                       //interface IApril<T>  : IMonth<T, U>
{}  //Error
```

Les interfaces génériques peuvent hériter d'interfaces non génériques si l'interface générique est de type covariante, ce qui signifie qu'elle utilise uniquement son paramètre de type comme valeur de retour. Dans la bibliothèque de classes .NET, IEnumerable<T> hérite d'IEnumerable, car IEnumerable<T> utilise uniquement `T` dans la valeur de retour de GetEnumerator et dans l'accesseur de propriété Current.

Les classes concrètes peuvent implémenter des interfaces construites fermées, comme suit :

C#

```
interface IBaseInterface<T> { }

class SampleClass : IBaseInterface<string> { }
```

Les classes génériques peuvent implémenter des interfaces génériques ou des interfaces construites fermées tant que la liste de paramètres de classe fournit tous les arguments requis par l'interface, comme suit :

C#

```
interface IBaseInterface1<T> { }
interface IBaseInterface2<T, U> { }

class SampleClass1<T> : IBaseInterface1<T> { }          //No error
class SampleClass2<T> : IBaseInterface2<T, string> { }  //No error
```

Les règles qui déterminent la surcharge des méthodes sont les mêmes pour les méthodes dans les classes génériques, les structs génériques ou les interfaces génériques. Pour plus d'informations, consultez Méthodes génériques.

À compter de C# 11, les interfaces peuvent déclarer des membres `static abstract` ou `static virtual`. Les interfaces qui déclarent des membres `static abstract` ou `static virtual` sont presque toujours des interfaces génériques. Le compilateur doit résoudre les appels aux méthodes `static virtual` et `static abstract` au moment de la compilation. Les méthodes `static virtual` et `static abstract` déclarées dans les interfaces n'ont pas de mécanisme de dispatch d'exécution analogue aux méthodes `virtual` ou `abstract` déclarées dans les classes. Au lieu de cela, le compilateur utilise des informations de type disponibles au moment de la compilation. Ces membres sont généralement déclarés dans les interfaces génériques. En outre, la plupart des interfaces qui déclarent les méthodes `static virtual` ou `static abstract` notifient que l'un des paramètres de types doit implémenter l'interface déclarée. Le compilateur utilise ensuite les arguments de types fournis pour résoudre le type du membre déclaré.

Voir aussi

- Guide de programmation C#
- Introduction aux génériques
- interface
- Génériques

Méthodes génériques (guide de programmation C#)

Article • 07/04/2023

Une méthode générique est une méthode qui est déclarée avec des paramètres de type, comme suit :

C#

```
static void Swap<T>(ref T lhs, ref T rhs)
{
    T temp;
    temp = lhs;
    lhs = rhs;
    rhs = temp;
}
```

L'exemple de code suivant montre une manière d'appeler la méthode, avec `int` comme argument de type :

C#

```
public static void TestSwap()
{
    int a = 1;
    int b = 2;

    Swap<int>(ref a, ref b);
    System.Console.WriteLine(a + " " + b);
}
```

Vous pouvez également omettre l'argument de type et le compilateur le déduira. L'appel suivant à `Swap` est équivalent à l'appel précédent :

C#

```
Swap(ref a, ref b);
```

Les mêmes règles d'inférence de type s'appliquent aux méthodes statiques et aux méthodes d'instance. Le compilateur peut déduire les paramètres de type d'après les arguments de méthode que vous passez. Il ne peut pas déduire les paramètres de type uniquement à partir d'une valeur de contrainte ou de retour. Par conséquent, l'inférence de type ne fonctionne pas avec les méthodes qui n'ont pas de paramètres. L'inférence

de type se produit au moment de la compilation avant que le compilateur n'essaie de résoudre toute signature de méthode surchargée. Le compilateur applique la logique d'inférence de type à toutes les méthodes génériques qui partagent le même nom. À l'étape de résolution de la surcharge, le compilateur inclut uniquement les méthodes génériques sur lesquelles l'inférence de type a réussi.

Dans une classe générique, les méthodes non génériques peuvent accéder aux paramètres de type de niveau classe, comme suit :

C#

```
class SampleClass<T>
{
    void Swap(ref T lhs, ref T rhs) { }
}
```

Si vous définissez une méthode générique qui accepte les mêmes paramètres de type que la classe conteneur, le compilateur génère l'avertissement CS0693, car l'argument fourni à l'intérieur de la portée de la méthode pour le `T` interne masque l'argument fourni pour le `T` externe. Si vous avez éventuellement besoin d'appeler une méthode de classe générique avec des arguments de type différents de ceux qui ont été fournis quand la classe a été instanciée, envisagez de fournir un autre identificateur pour le paramètre de type de la méthode, comme indiqué dans `GenericList2<T>` dans l'exemple suivant.

C#

```
class GenericList<T>
{
    // CS0693.
    void SampleMethod<T>() { }
}

class GenericList2<T>
{
    // No warning.
    void SampleMethod<U>() { }
}
```

Utilisez des contraintes pour permettre des opérations plus spécialisées sur les paramètres de type dans les méthodes. Cette version de `Swap<T>`, maintenant appelée `SwapIfGreater<T>`, peut être utilisée avec des arguments de type qui implémentent IComparable<T> uniquement.

C#

```
void SwapIfGreater<T>(ref T lhs, ref T rhs) where T : System.IComparable<T>
{
    T temp;
    if (lhs.CompareTo(rhs) > 0)
    {
        temp = lhs;
        lhs = rhs;
        rhs = temp;
    }
}
```

Les méthodes génériques peuvent être surchargées sur plusieurs paramètres de type. Par exemple, les méthodes suivantes peuvent toutes être dans la même classe :

C#

```
void DoWork() { }
void DoWork<T>() { }
void DoWork<T, U>() { }
```

Spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#.

Voir aussi

- System.Collections.Generic
- Guide de programmation C#
- Introduction aux génériques
- Méthodes

Génériques et tableaux (guide de programmation C#)

Article • 07/04/2023

Les tableaux unidimensionnels qui ont une limite inférieure à zéro implémentent automatiquement IList<T>. Cela vous permet de créer des méthodes génériques qui peuvent utiliser le même code pour itérer au sein de tableaux et d'autres types de collections. Cette technique est essentiellement utile pour lire les données dans des collections. L'interface IList<T> ne peut pas être utilisée pour ajouter ni supprimer des éléments dans un tableau. Une exception est levée si vous essayez d'appeler une méthode IList<T> telle que RemoveAt sur un tableau dans ce contexte.

L'exemple de code suivant montre comment une méthode générique seule qui prend un paramètre d'entrée IList<T> peut itérer à la fois au sein d'une liste et d'un tableau (en l'occurrence un tableau d'entiers).

C#

```
class Program
{
    static void Main()
    {
        int[] arr = { 0, 1, 2, 3, 4 };
        List<int> list = new List<int>();

        for (int x = 5; x < 10; x++)
        {
            list.Add(x);
        }

        ProcessItems<int>(arr);
        ProcessItems<int>(list);
    }

    static void ProcessItems<T>(IList<T> coll)
    {
        // IsReadOnly returns True for the array and False for the List.
        System.Console.WriteLine
            ("IsReadOnly returns {0} for this collection.",
            coll.IsReadOnly);

        // The following statement causes a run-time exception for the
        // array, but not for the List.
        //coll.RemoveAt(4);

        foreach (T item in coll)
        {
```

```
            System.Console.Write(item?.ToString() + " ");
        }
        System.Console.WriteLine();
    }
}
```

Voir aussi

- System.Collections.Generic
- Guide de programmation C#
- Génériques
- Tableaux
- Génériques

Délégués génériques (guide de programmation C#)

Article • 07/04/2023

Un délégué peut définir ses propres paramètres de type. Le code qui référence le délégué générique peut spécifier l'argument de type pour créer un type construit fermé, comme lors de l'instanciation d'une classe générique ou d'un appel d'une méthode générique, ainsi que l'illustre l'exemple ci-dessous :

C#

```
public delegate void Del<T>(T item);
public static void Notify(int i) { }

Del<int> m1 = new Del<int>(Notify);
```

Le langage C# version 2.0 propose une nouvelle fonctionnalité appelée conversion de groupe de méthodes, qui s'applique aussi bien aux types concrets qu'aux types délégués génériques et vous permet d'écrire la ligne précédente avec la syntaxe simplifiée suivante :

C#

```
Del<int> m2 = Notify;
```

Les délégués définis dans une classe générique peuvent utiliser les paramètres de type de classe générique, à l'instar des méthodes de classe.

C#

```
class Stack<T>
{
    public delegate void StackDelegate(T[] items);
}
```

Le code qui référence le délégué doit spécifier l'argument de type de la classe conteneur, comme suit :

C#

```
private static void DoWork(float[] items) { }

public static void TestStack()
```

```
{
    Stack<float> s = new Stack<float>();
    Stack<float>.StackDelegate d = DoWork;
}
```

Les délégués génériques sont particulièrement utiles pour définir des événements basés sur le modèle de design type parce que l'argument de l'expéditeur peut être fortement typé et il n'est plus nécessaire d'en effectuer un cast vers et depuis Object.

C#

```
delegate void StackEventHandler<T, U>(T sender, U eventArgs);

class Stack<T>
{
    public class StackEventArgs : System.EventArgs { }
    public event StackEventHandler<Stack<T>, StackEventArgs>? StackEvent;

    protected virtual void OnStackChanged(StackEventArgs a)
    {
        if (StackEvent is not null)
            StackEvent(this, a);
    }
}

class SampleClass
{
    public void HandleStackChange<T>(Stack<T> stack, Stack<T>.StackEventArgs
args) { }
}

public static void Test()
{
    Stack<double> s = new Stack<double>();
    SampleClass o = new SampleClass();
    s.StackEvent += o.HandleStackChange;
}
```

Voir aussi

- System.Collections.Generic
- Guide de programmation C#
- Introduction aux génériques
- Méthodes génériques
- Classes génériques
- Interfaces génériques
- Délégués

- Génériques

Différences entre les modèles C++ et les génériques C# (Guide de programmation C#)

Article • 07/04/2023

Les génériques C# et les modèles C++ sont des fonctionnalités linguistiques qui prennent en charge les types paramétrables. Toutefois, il existe de nombreuses différences entre les deux. Au niveau de la syntaxe, les génériques C# constituent une approche plus simple des types paramétrables, sans la complexité des modèles C++. De plus, C# n'essaie pas de fournir toutes les fonctionnalités fournies par les modèles C++. Au niveau de l'implémentation, la principale différence est que les substitutions de type générique C# sont effectuées au moment de l'exécution. Ainsi, les informations de type générique sont conservées pour les objets instanciés. Pour plus d'informations, consultez Génériques dans le runtime.

Voici les principales différences entre les modèles C++ et les génériques C# :

- Les génériques C# n'offrent pas le même niveau de flexibilité que les modèles C++. Par exemple, il n'est pas possible d'appeler des opérateurs arithmétiques dans une classe générique C#, bien qu'il soit possible d'appeler des opérateurs définis par l'utilisateur.
- C# n'autorise pas les paramètres de modèle sans type, tels que `template C<int i> {}`.
- C# ne prend pas en charge la spécialisation explicite (une implémentation personnalisée d'un modèle pour un type spécifique).
- C# ne prend pas en charge la spécialisation partielle (une implémentation personnalisée pour un sous-ensemble des arguments de type).
- C# n'autorise pas l'utilisation du paramètre de type comme classe de base pour le type générique.
- C# n'autorise pas les paramètres de type à avoir des types par défaut.
- En C#, un paramètre de type générique ne peut pas être lui-même un générique, bien que les types construits puissent être utilisés comme des génériques. C++ autorise les paramètres de modèle.

- C++ autorise le code qui peut ne pas être valide pour tous les paramètres de type dans le modèle, qui est ensuite vérifié pour identifier le type spécifique utilisé comme paramètre de type. C# exige que le code dans une classe soit écrit de telle sorte qu'il fonctionne avec tout type qui satisfait aux contraintes. Par exemple, en C++ il est possible d'écrire une fonction qui utilise les opérateurs arithmétiques `+` et `-` sur des objets du paramètre de type, ce qui générera une erreur au moment de l'instanciation du modèle avec un type qui ne prend pas en charge ces opérateurs. Ceci n'est pas autorisé par C#. Les seules constructions de langage autorisées sont celles qui peuvent être déduites des contraintes.

Voir aussi

- Guide de programmation C#
- Introduction aux génériques
- Modèles

Génériques dans le runtime (guide de programmation C#)

Article • 07/04/2023

Quand une méthode ou un type générique est compilé dans le langage intermédiaire Microsoft (MSIL), il ou elle contient des métadonnées qui l'identifient comme ayant des paramètres de type. La façon dont le langage MSIL d'un type générique est utilisé diffère selon que le paramètre de type fourni est un type valeur ou référence.

Quand un type générique est construit en premier avec un type valeur comme paramètre, le runtime crée un type générique spécialisé avec le paramètre fourni ou les paramètres substitués aux emplacements appropriés dans le langage MSIL. Des types génériques spécialisés sont créés une fois pour chaque type valeur unique qui est utilisé comme paramètre.

Par exemple, supposons que votre code de programme ait déclaré une pile construite d'entiers :

C#

```
Stack<int>? stack;
```

À ce stade, le runtime génère une version spécialisée de la classe Stack<T> dont l'entier a été substitué correctement à son paramètre. Maintenant, chaque fois que le code de programme utilise une pile d'entiers, le runtime réutilise la classe Stack<T> spécialisée générée. Dans l'exemple suivant, deux instances d'une pile d'entiers sont créées, et elles partagent une instance unique du code `Stack<int>` :

C#

```
Stack<int> stackOne = new Stack<int>();
Stack<int> stackTwo = new Stack<int>();
```

Toutefois, supposons qu'une autre classe Stack<T> avec un type valeur différent tel qu'un type `long` ou une structure définie par l'utilisateur comme son paramètre est créée à un autre endroit dans votre code. Le runtime génère alors une autre version du type générique et substitue un type `long` aux emplacements appropriés dans le langage MSIL. Les conversions ne sont plus nécessaires parce que chaque classe générique spécialisée contient le type valeur en mode natif.

Les génériques fonctionnent un peu différemment pour les types référence. La première fois qu'un type générique est construit avec un type référence quelconque, le runtime crée un type générique spécialisé avec les références d'objet substituées aux paramètres dans le langage MSIL. Ensuite, chaque fois qu'un type construit est instancié avec un type référence comme son paramètre, indépendamment de son type, le runtime réutilise la version spécialisée créée précédemment pour le type générique. Ceci est possible parce que toutes les références ont la même taille.

Par exemple, supposons que vous ayez deux types référence, une classe `Customer` et une classe `Order`, et supposons également que vous ayez créé une pile de types `Customer` :

C#

```
class Customer { }
class Order { }
```

C#

```
Stack<Customer> customers;
```

À ce stade, le runtime génère une version spécialisée de la classe Stack<T> qui stocke des références d'objet qui seront remplies ultérieurement au lieu de stocker des données. Supposons que la ligne de code suivante crée une pile d'un autre type référence, qui est nommé `Order` :

C#

```
Stack<Order> orders = new Stack<Order>();
```

Contrairement aux types valeur, une autre version spécialisée de la classe Stack<T> n'est pas créée pour le type `Order`. À la place, une instance de la version spécialisée de la classe Stack<T> est créée et la variable `orders` est définie pour la référencer. Supposons que vous ayez rencontré ensuite une ligne de code pour créer une pile d'un type `Customer` :

C#

```
customers = new Stack<Customer>();
```

Comme avec l'utilisation précédente de la classe Stack<T> créée à l'aide du type `Order`, une autre instance de la classe Stack<T> spécialisée est créée. Les pointeurs contenus ici

sont configurés pour référencer une zone de mémoire de la taille d'un type `Customer`. Dans la mesure où le nombre de types référence peut varier de manière importante d'un programme à l'autre, l'implémentation C# de génériques réduit sensiblement le volume du code en ramenant à une unité le nombre de classes spécialisées créées par le compilateur pour les classes génériques de types référence.

De plus, quand une classe C# générique est instanciée à l'aide d'un paramètre de type valeur ou référence, la réflexion peut l'interroger au moment du temps d'exécution, et son type réel ainsi que son paramètre de type peuvent être vérifiés.

Voir aussi

- System.Collections.Generic
- Guide de programmation C#
- Introduction aux génériques
- Génériques

Génériques et réflexion

Article • 20/03/2023

Étant donné que le common language runtime (CLR) a accès aux informations concernant les types génériques au moment de l'exécution, vous pouvez utiliser la réflexion pour obtenir des informations sur les types génériques de la même manière que pour les types non génériques. Pour plus d'informations, consultez Génériques dans le Runtime.

L'espace de noms System.Reflection.Emit contient également de nouveaux membres qui prennent en charge les génériques. Consultez Comment : définir un type générique avec l'émission de réflexion.

Pour obtenir la liste des conditions invariantes pour les termes utilisés dans la réflexion générique, consultez les remarques sur la propriété IsGenericType :

- IsGenericType : retourne la valeur true si un type est générique.
- GetGenericArguments : retourne un tableau d'objets `Type` qui représentent les arguments de type fournis pour un type construit, ou les paramètres de type d'une définition de type générique.
- GetGenericTypeDefinition : retourne la définition de type générique sous-jacente pour le type construit actuel.
- GetGenericParameterConstraints : retourne un tableau d'objets `Type` représentant les contraintes qui s'exercent sur le paramètre de type générique actuel.
- ContainsGenericParameters : retourne la valeur true si le type ou l'un de ses types ou méthodes englobants contient des paramètres de type pour lesquels des types spécifiques n'ont pas été fournis.
- GenericParameterAttributes : obtient une combinaison d'indicateurs `GenericParameterAttributes` décrivant les contraintes spéciales du paramètre de type générique actuel.
- GenericParameterPosition : pour un objet `Type` qui représente un paramètre de type, obtient la position du paramètre de type dans la liste de paramètres de type de la définition de type générique ou de la définition de méthode générique qui a déclaré le paramètre de type.
- IsGenericParameter : obtient une valeur indiquant si le `Type` actuel représente un paramètre de type d'un type générique ou d'une définition de méthode.
- IsGenericTypeDefinition : obtient une valeur qui indique si le Type actuel représente une définition de type générique, à partir de laquelle d'autres types génériques peuvent être construits. Retourne la valeur true si le type représente la définition d'un type générique.

- DeclaringMethod : retourne la méthode générique qui a défini le paramètre de type générique actuel, ou Null si le paramètre de type n'a pas été défini par une méthode générique.
- MakeGenericType : substitue les éléments d'un tableau de types aux paramètres de type de la définition du type générique actuel et retourne un objet Type qui représente le type construit résultant.

De plus, les membres de la classe MethodInfo permettent la génération d'informations d'exécution pour les méthodes génériques. Pour obtenir la liste des conditions invariantes des termes utilisés pour réfléchir les méthodes génériques, consultez les remarques sur la propriété IsGenericMethod :

- IsGenericMethod : retourne la valeur true si une méthode est générique.
- GetGenericArguments : retourne un tableau d'objets de type qui représentent les arguments de type d'une méthode générique construite, ou les paramètres de type d'une définition de méthode générique.
- GetGenericMethodDefinition : retourne la définition de méthode générique sous-jacente pour la méthode construite actuelle.
- ContainsGenericParameters : retourne la valeur true si la méthode ou l'un de ses types englobants contient des paramètres de type pour lesquels des types spécifiques n'ont pas été fournis.
- IsGenericMethodDefinition : retourne la valeur true si le MethodInfo actuel représente la définition d'une méthode générique.
- MakeGenericMethod : substitue les éléments d'un tableau de types aux paramètres de type de la définition de méthode générique actuelle et retourne un objet MethodInfo représentant la méthode construite résultante.

Voir aussi

- Génériques
- Réflexion et types génériques
- Génériques

Génériques et attributs

Article • 20/03/2023

Les attributs peuvent être appliqués aux types génériques de la même manière qu'aux types non génériques. Toutefois, vous pouvez appliquer des attributs uniquement sur des *types génériques ouverts* et des *types génériques construits fermés*. Vous ne pouvez pas attribuer d'attributs sur des *types génériques partiellement construits*. Un *type générique ouvert* est un type dans lequel aucun des arguments de type n'est spécifié, comme `Dictionary<TKey, TValue>`. Un *type générique construit fermé* spécifie tous les arguments de type, par exemple `Dictionary<string, object>`. Un *type générique partiellement construit* spécifie certains arguments de type, mais pas tous. par exemple `Dictionary<string, TValue>`.

Les exemples suivants utilisent cet attribut personnalisé :

C#

```
class CustomAttribute : Attribute
{
    public object? info;
}
```

Un attribut peut référencer un type générique ouvert :

C#

```
public class GenericClass1<T> { }

[CustomAttribute(info = typeof(GenericClass1<>))]
class ClassA { }
```

Spécifiez plusieurs paramètres de type à l'aide du nombre approprié de virgules. Dans cet exemple, `GenericClass2` a deux paramètres de type :

C#

```
public class GenericClass2<T, U> { }

[CustomAttribute(info = typeof(GenericClass2<,>))]
class ClassB { }
```

Un attribut peut référencer un type générique construit fermé :

C#

```
public class GenericClass3<T, U, V> { }

[CustomAttribute(info = typeof(GenericClass3<int, double, string>))]
class ClassC { }
```

Un attribut qui référence un paramètre de type générique entraîne une erreur de compilation :

C#

```
[CustomAttribute(info = typeof(GenericClass3<int, T, string>))]  //Error
CS0416
class ClassD<T> { }
```

À compter de C# 11, un type générique peut hériter de Attribute :

C#

```
public class CustomGenericAttribute<T> : Attribute { }  //Requires C# 11
```

Pour obtenir des informations sur un type générique ou un paramètre de type au moment de l'exécution, vous pouvez utiliser les méthodes de System.Reflection. Pour plus d'informations, consultez Génériques et réflexion.

Voir aussi

- Génériques
- Attributs

Fichier et flux de données E/S

Article • 01/08/2023

E/S de fichier et de flux (entrées/sorties) fait référence au transfert de données vers ou depuis un support de stockage. Dans .NET, les espaces de noms `System.IO` contiennent des types qui permettent la lecture et l'écriture, de façon synchrone ou asynchrone, sur les flux de données et les fichiers. Ces espaces de noms contiennent également les types qui exécutent la compression et la décompression sur les fichiers, et ceux qui permettent la communication via des canaux et des ports série.

Un fichier est une collection ordonnée et nommée d'octets ayant un stockage persistant. Lorsque vous travaillez sur des fichiers, vous manipulez des chemins d'accès aux répertoires, du stockage disque et des noms de fichiers et de répertoires. En revanche, un flux est une séquence d'octets que vous pouvez lire et écrire dans un magasin de stockage, qui peut être l'un des nombreux supports de stockage (par exemple, des disques ou de la mémoire). De même qu'il existe d'autres magasins de stockage que les disques, il existe d'autres flux que les flux de fichiers, tels que les flux de réseau, de mémoire et de canaux.

Fichiers et répertoires

Utilisez les types de l'espace de noms System.IO pour interagir avec les fichiers et les répertoires. Par exemple, obtenez et définissez les propriétés des fichiers et des répertoires, et extrayez les collections de fichiers et de répertoires en fonction de critères de recherche.

Pour les conventions d'affectation de noms des chemins et les nouvelles méthodes pour exprimer un chemin de fichier pour les systèmes Windows, notamment avec la syntaxe des appareils DOS prise en charge dans .NET Core 1.1 et ultérieur et .NET Framework 4.6.2 et ultérieur, consultez Formats de chemin de fichier sur les systèmes Windows.

Voici quelques classes de fichiers et de répertoires couramment utilisées :

- File fournit des méthodes statiques pour la création, la copie, la suppression, le déplacement et l'ouverture de fichiers et permet de créer un objet FileStream.
- FileInfo fournit des méthodes d'instance pour la création, la copie, la suppression, le déplacement et l'ouverture de fichiers et permet de créer un objet FileStream.

- Directory fournit des méthodes statiques pour la création, le déplacement et l'énumération dans les répertoires et les sous-répertoires.
- DirectoryInfo fournit des méthodes d'instance pour la création, le déplacement et l'énumération dans les répertoires et les sous-répertoires.
- Path fournit des méthodes et des propriétés pour le traitement des chaînes de répertoire entre plateformes.

Vous devez toujours fournir une gestion des exceptions robuste lors de l'appel des méthodes de système de fichiers. Pour plus d'informations, consultez Gestion des erreurs E/S.

En plus d'utiliser ces classes, les utilisateurs de Visual Basic peuvent utiliser les méthodes et les propriétés fournies par la classe Microsoft.VisualBasic.FileIO.FileSystem pour l'E/S de fichier.

Voir Comment : copier des répertoires, Comment : créer une liste de répertoires et Comment : énumérer des répertoires et des fichiers.

Flux

La classe abstraite de base Stream prend en charge la lecture et l'écriture d'octets. Toutes les classes qui représentent des flux héritent de la classe Stream. La classe Stream et ses classes dérivées donnent une vue globale des sources de données et des référentiels, isolant ainsi le programmeur des détails propres au système d'exploitation et aux périphériques sous-jacents.

Les flux impliquent trois opérations fondamentales :

- Lecture : transfert de données d'un flux vers une structure de données tel qu'un tableau d'octets.
- Écriture : transfert de données d'une source de données dans un flux.
- Recherche : envoi d'une requête concernant la position actuelle dans un flux et modification de cette dernière.

Selon la source de données sous-jacente ou le référentiel, un flux peut prendre en charge certaines de ces capacités. Par exemple, la classe PipeStream ne prend pas en charge la recherche. Les propriétés CanRead, CanWrite et CanSeek d'un flux spécifient les opérations que le flux prend en charge.

Voici quelques classes de flux couramment utilisées :

- FileStream pour la lecture et l'écriture dans un fichier.
- IsolatedStorageFileStream pour la lecture et l'écriture dans un fichier d'un stockage isolé.
- MemoryStream pour la lecture et l'écriture dans la mémoire en tant que magasin de stockage.
- BufferedStream pour l'amélioration des performances de vos opérations de lecture et d'écriture.
- NetworkStream pour la lecture et l'écriture sur vos sockets réseau.
- PipeStream pour la lecture et l'écriture sur des canaux nommés ou anonymes.
- CryptoStream pour la liaison des flux de données aux transformations de chiffrement.

Pour obtenir un exemple d'utilisation de flux de façon asynchrone, consultez E/S sur fichier asynchrones.

Lecteurs et writers

L'espace de noms System.IO fournit également des types pour la lecture des caractères encodés à partir des flux et l'écriture des caractères encodés dans les flux. En général, les flux sont conçus pour l'entrée et la sortie d'octets. Les types lecteur et writer gèrent la conversion des caractères encodés en octets et inversement pour permettre au flux de terminer l'opération. Chaque classe de lecteur et writer est associée à un flux, qui peut être récupéré via la propriété `BaseStream` de la classe.

Voici quelques classes de lecteur et writer couramment utilisées :

- BinaryReader et BinaryWriter pour la lecture et l'écriture des types de données primitifs comme valeurs binaires.
- StreamReader et StreamWriter pour la lecture et l'écriture à l'aide d'une valeur d'encodage pour convertir des caractères en octets et inversement.
- StringReader et StringWriter pour la lecture et l'écriture des caractères à partir de chaînes et inversement.
- TextReader et TextWriter servent de classes de base abstraites à d'autres lecteurs et writers qui lisent et écrivent des caractères et des chaînes, mais pas des données binaires.

Voir Comment : lire du texte dans un fichier, Comment : écrire du texte dans un fichier, Comment : lire les caractères d'une chaîne et Comment : écrire des caractères dans une chaîne.

Opérations d'E/S asynchrones

La lecture ou l'écriture de grandes quantités de données peut consommer beaucoup de ressources. Vous devez effectuer ces tâches de façon asynchrone si votre application doit rester réactive aux actions de l'utilisateur. Avec les opérations d'E/S synchrones, le thread d'interface utilisateur est bloqué jusqu'à ce que l'opération consommatrice de ressources soit terminée. Utilisez des opérations d'E/S asynchrones lors du développement d'applications Windows 8.x Store pour empêcher de donner l'impression que votre application a cessé de fonctionner.

Les membres asynchrones contiennent `Async` dans leurs noms, comme les méthodes CopyToAsync, FlushAsync, ReadAsync et WriteAsync. Utilisez ces méthodes avec `async` et les mots clés `await`.

Pour plus d'informations, consultez E/S de fichier asynchrone.

Compression

La compression désigne le processus de réduction de la taille d'un fichier pour le stockage. La décompression consiste à extraire le contenu d'un fichier compressé afin qu'il soit dans un format utilisable. L'espace de noms System.IO.Compression contient des types pour la compression et la décompression de fichiers et de flux.

Les classes suivantes sont fréquemment utilisées lors de la compression et de la décompression de fichiers et de flux :

- ZipArchive pour la création et l'extraction des entrées de l'archive zip.
- ZipArchiveEntry pour la représentation d'un fichier compressé.
- ZipFile : pour la création, l'extraction et l'ouverture d'un package compressé.
- ZipFileExtensions pour la création et l'extraction des entrées dans un package compressé.
- DeflateStream pour la compression et la décompression des flux en utilisant l'algorithme Deflate.

- GZipStream pour la compression et la décompression des flux au format de données gzip.

Voir Comment : compresser et extraire des fichiers.

Stockage isolé

Le stockage isolé est un mécanisme de stockage de données qui offre une isolation et une sécurité en définissant des méthodes standardisées pour associer du code à des données enregistrées. Le stockage fournit un système de fichiers virtuel qui est isolé par utilisateur, assembly et (éventuellement) domaine. Le stockage isolé est particulièrement utile lorsque votre application n'a pas l'autorisation d'accès aux fichiers utilisateur. Enregistrez les paramètres ou les fichiers de votre application d'une façon contrôlée par la stratégie de sécurité de l'ordinateur.

Le stockage isolé n'est pas disponible pour les applications Windows 8.x Store. Utilisez à la place les classes de données d'application de l'espace de noms Windows.Storage. Pour plus d'informations, consultez Données de l'application.

Les classes suivantes sont fréquemment utilisées lors d'une implémentation de stockage isolé :

- IsolatedStorage fournit la classe de base pour les implémentations de stockage isolé.
- IsolatedStorageFile fournit une zone de stockage isolé qui contient les fichiers et les répertoires.
- IsolatedStorageFileStream expose un fichier au sein d'un stockage isolé.

Voir Stockage isolé.

Opérations d'E/S dans les applications Windows Store

Les applications .NET pour Windows 8.x Store contiennent plusieurs types pour la lecture et l'écriture dans les flux. Toutefois, cet ensemble n'inclut pas tous les types d'E/S .NET.

Quelques différences importantes sont notables pour les opérations d'E/S dans les applications Windows 8.x Store :

- Les types spécifiquement liés aux opérations sur les fichiers, tels que File, FileInfo, Directory et DirectoryInfo ne sont pas inclus dans les applications .NET pour Windows 8.x Store. À la place, utilisez les types dans l'espace de noms Windows.Storage de Windows Runtime, par exemple StorageFile et StorageFolder.
- Le stockage isolé n'est pas disponible ; à la place, utilisez les données d'application.
- Utilisez les méthodes asynchrones, telles que ReadAsync et WriteAsync pour empêcher le blocage du thread d'interface utilisateur.
- Les types de compression ZipFile et ZipFileExtensions basés sur le chemin d'accès ne sont pas disponibles. À la place, utilisez les types dans l'espace de noms Windows.Storage.Compression.

Vous pouvez convertir entre les flux .NET Framework et les flux Windows Runtime, si nécessaire. Pour plus d'informations, consultez la section Comment : effectuer une conversion entre les flux .NET Framework et les flux Windows Runtime ou WindowsRuntimeStreamExtensions.

Pour plus d'informations sur les opérations d'E/S dans une application Windows 8.x Store, consultez la section Démarrage rapide : lecture et écriture de fichiers.

E/S et sécurité

Lorsque vous utilisez les classes de l'espace de noms System.IO, vous devez suivre les exigences de sécurité du système d'exploitation telles que les listes de contrôle d'accès (ACL) pour contrôler l'accès aux fichiers et aux répertoires. Ces spécifications s'ajoutent aux spécifications FileIOPermission existantes. Les listes de contrôle d'accès peuvent être gérées par programmation. Pour plus d'informations, consultez Comment : ajouter ou supprimer des entrées dans la liste de contrôle d'accès.

Les stratégies de sécurité par défaut empêchent les applications provenant d'Internet ou de l'intranet d'accéder aux fichiers sur l'ordinateur de l'utilisateur. Par conséquent, n'utilisez pas les classes d'E/S qui requièrent un chemin d'accès à un fichier physique lors de l'écriture du code qui sera téléchargé sur Internet ou sur l'intranet. Utilisez plutôt un stockage isolé pour les applications .NET.

La vérification de sécurité n'est exécutée qu'à la création du flux. Par conséquent, n'ouvrez pas de flux pour le passer ensuite à du code ou à des domaines d'application d'un niveau de sécurité inférieur.

Rubriques connexes

- Tâches d’E/S courantes
  Présente les tâches d’E/S associées aux fichiers, aux répertoires et aux flux, et des liens vers du contenu et des exemples appropriés pour chaque tâche.

- E/S sur fichier asynchrones
  Décrit les opérations élémentaires des E/S asynchrones et leurs avantages en termes de performances.

- Stockage isolé
  Décrit un dispositif de stockage des données qui assure l'isolation et la sécurité en définissant des solutions standardisées visant à associer le code aux données enregistrées.

- Canaux
  Décrit des opérations anonymes et de canal nommé dans .NET.

- Fichiers mappés en mémoire
  Décrit les fichiers mappés en mémoire, qui contiennent le contenu de fichiers stockés sur le disque dans la mémoire virtuelle. Vous pouvez utiliser des fichiers mappés en mémoire afin de modifier des fichiers très volumineux et de créer la mémoire partagée pour la communication entre processus.

Guide pratique pour énumérer des répertoires et des fichiers

Article • 03/06/2023

Les collections énumérables offrent de meilleures performances que les tableaux lorsque vous travaillez avec collections volumineuses de fichiers et de répertoires. Pour énumérer des répertoires et des fichiers, utilisez des méthodes qui retournent une collection énumérable de noms de répertoire ou de fichier ou leurs objets DirectoryInfo, FileInfo ou FileSystemInfo.

Si vous souhaitez parcourir et retourner uniquement les noms des répertoires ou des fichiers, utilisez les méthodes d'énumération de la classe Directory. Si vous souhaitez parcourir et retourner d'autres propriétés des répertoires ou des fichiers, utilisez les classes DirectoryInfo et FileSystemInfo.

Vous pouvez utiliser les collections énumérables issues de ces méthodes en tant que paramètre IEnumerable<T> pour les constructeurs de classes de collection, comme List<T>.

Le tableau suivant récapitule les méthodes qui retournent des collections énumérables de fichiers et de répertoires .

Pour parcourir et retourner	**Utiliser la méthode**
Noms de répertoires	Directory.EnumerateDirectories
Informations sur le répertoire (DirectoryInfo)	DirectoryInfo.EnumerateDirectories
Noms de fichiers	Directory.EnumerateFiles
Informations sur le fichier (FileInfo)	DirectoryInfo.EnumerateFiles
Noms d'entrée de système de fichiers	Directory.EnumerateFileSystemEntries
Informations sur les entrées de système de fichiers (FileSystemInfo)	DirectoryInfo.EnumerateFileSystemInfos
Noms de répertoires et de fichiers	Directory.EnumerateFileSystemEntries

Notes

Bien que vous puissiez énumérer immédiatement tous les fichiers dans les sous-répertoires d'un répertoire parent à l'aide de l'option **AllDirectories** de l'énumération **SearchOption** facultative, des erreurs **UnauthorizedAccessException**

peuvent rendre l'énumération incomplète. Vous pouvez intercepter ces exceptions en énumérant d'abord les répertoires, puis les fichiers.

Guide pratique pour utiliser la classe Directory

L'exemple suivant utilise la méthode Directory.EnumerateDirectories(String) pour obtenir une liste des noms de répertoires de niveau supérieur dans un chemin spécifié.

C#

```
using System;
using System.Collections.Generic;
using System.IO;

class Program
{
    private static void Main(string[] args)
    {
        try
        {
            // Set a variable to the My Documents path.
            string docPath =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

            List<string> dirs = new List<string>
(Directory.EnumerateDirectories(docPath));

            foreach (var dir in dirs)
            {
                Console.WriteLine($"
{dir.Substring(dir.LastIndexOf(Path.DirectorySeparatorChar) + 1)}");
            }
            Console.WriteLine($"{dirs.Count} directories found.");
        }
        catch (UnauthorizedAccessException ex)
        {
            Console.WriteLine(ex.Message);
        }
        catch (PathTooLongException ex)
        {
            Console.WriteLine(ex.Message);
        }
    }
}
```

L'exemple suivant utilise la méthode Directory.EnumerateFiles(String, String, SearchOption) pour énumérer de manière récursive tous les noms de fichiers dans un répertoire et dans les sous-répertoires qui correspondent à un certain modèle. Il lit

chaque ligne de chaque fichier et affiche les lignes qui contiennent la chaîne spécifiée, avec les noms de fichiers et chemins correspondants.

C#

```
using System;
using System.IO;
using System.Linq;

class Program
{
    static void Main(string[] args)
    {
        try
        {
            // Set a variable to the My Documents path.
            string docPath =

Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

            var files = from file in Directory.EnumerateFiles(docPath,
"*.txt", SearchOption.AllDirectories)
                        from line in File.ReadLines(file)
                        where line.Contains("Microsoft")
                        select new
                        {
                            File = file,
                            Line = line
                        };

            foreach (var f in files)
            {
                Console.WriteLine($"{f.File}\t{f.Line}");
            }
            Console.WriteLine($"{files.Count().ToString()} files found.");
        }
        catch (UnauthorizedAccessException uAEx)
        {
            Console.WriteLine(uAEx.Message);
        }
        catch (PathTooLongException pathEx)
        {
            Console.WriteLine(pathEx.Message);
        }
    }
}
```

Exemples : Utiliser la classe DirectoryInfo

L'exemple suivant utilise la méthode DirectoryInfo.EnumerateDirectories pour lister une collection de répertoires de niveau supérieur dont CreationTimeUtc est antérieur à une certaine valeur DateTime.

C#

```
using System;
using System.IO;

namespace EnumDir
{
    class Program
    {
        static void Main(string[] args)
        {
            // Set a variable to the Documents path.
            string docPath =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

            DirectoryInfo dirPrograms = new DirectoryInfo(docPath);
            DateTime StartOf2009 = new DateTime(2009, 01, 01);

            var dirs = from dir in dirPrograms.EnumerateDirectories()
            where dir.CreationTimeUtc > StartOf2009
            select new
            {
                ProgDir = dir,
            };

            foreach (var di in dirs)
            {
                Console.WriteLine($"{di.ProgDir.Name}");
            }
        }
    }
}
// </Snippet1>
```

L'exemple suivant utilise la méthode DirectoryInfo.EnumerateFiles pour lister tous les fichiers dont Length dépasse 10 Mo. Cet exemple énumère d'abord les répertoires de niveau supérieur, pour intercepter les éventuelles exceptions d'accès non autorisé, puis énumère les fichiers.

C#

```
using System;
using System.IO;

class Program
{
    static void Main(string[] args)
```

```
    {
        // Set a variable to the My Documents path.
        string docPath =
Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

        DirectoryInfo diTop = new DirectoryInfo(docPath);

        try
        {
            foreach (var fi in diTop.EnumerateFiles())
            {
                try
                {
                    // Display each file over 10 MB;
                    if (fi.Length > 10000000)
                    {
                        Console.WriteLine($"
{fi.FullName}\t\t{fi.Length.ToString("N0")}");
                    }
                }
                catch (UnauthorizedAccessException unAuthTop)
                {
                    Console.WriteLine($"{unAuthTop.Message}");
                }
            }

            foreach (var di in diTop.EnumerateDirectories("*"))
            {
                try
                {
                    foreach (var fi in di.EnumerateFiles("*",
SearchOption.AllDirectories))
                    {
                        try
                        {
                            // Display each file over 10 MB;
                            if (fi.Length > 10000000)
                            {
                                Console.WriteLine($"
{fi.FullName}\t\t{fi.Length.ToString("N0")}");
                            }
                        }
                        catch (UnauthorizedAccessException unAuthFile)
                        {
                            Console.WriteLine($"unAuthFile:
{unAuthFile.Message}");
                        }
                    }
                }
                catch (UnauthorizedAccessException unAuthSubDir)
                {
                    Console.WriteLine($"unAuthSubDir:
{unAuthSubDir.Message}");
                }
            }
```

```
        }
        catch (DirectoryNotFoundException dirNotFound)
        {
            Console.WriteLine($"{dirNotFound.Message}");
        }
        catch (UnauthorizedAccessException unAuthDir)
        {
            Console.WriteLine($"unAuthDir: {unAuthDir.Message}");
        }
        catch (PathTooLongException longPath)
        {
            Console.WriteLine($"{longPath.Message}");
        }
    }
}
```

Voir aussi

- E/S de fichier et de flux

Tâches d’E/S courantes

Article • 01/06/2023

L'espace de noms System.IO fournit plusieurs classes qui permettent d'exécuter différentes actions, telles que la lecture et l'écriture, sur des fichiers, des répertoires et des flux de données. Pour plus d’informations, consultez E/S de fichier et de flux.

Tâches de fichier courantes

Action à réaliser...	**Consultez l'exemple décrit dans cette rubrique...**
Créer un fichier texte	Méthode File.CreateText Méthode FileInfo.CreateText Méthode File.Create Méthode FileInfo.Create
Écrire dans un fichier texte	Procédure : écrire du texte dans un fichier Guide pratique pour écrire un fichier texte (C++-CLI)
Lire à partir d'un fichier texte	Procédure : lire le texte d’un fichier
Ajouter du texte dans un fichier	Procédure : ouvrir un fichier journal et y ajouter des éléments Méthode File.AppendText Méthode FileInfo.AppendText
Renommer ou déplacer un fichier	Méthode File.Move Méthode FileInfo.MoveTo
Supprimer un fichier	Méthode File.Delete Méthode FileInfo.Delete
Copier un fichier	Méthode File.Copy Méthode FileInfo.CopyTo
Obtenir la taille d'un fichier	Propriété FileInfo.Length

Action à réaliser...	Consultez l'exemple décrit dans cette rubrique...
Obtenir les attributs d'un fichier	Méthode File.GetAttributes
Définir les attributs d'un fichier	Méthode File.SetAttributes
Déterminer si un fichier existe	Méthode File.Exists
Lire à partir d'un fichier binaire	Procédure : lire et écrire dans un fichier de données créé récemment
Écrire dans un fichier binaire	Procédure : lire et écrire dans un fichier de données créé récemment
Récupérer une extension de nom de fichier	Méthode Path.GetExtension
Récupérer le chemin d'accès qualifié complet d'un fichier	Méthode Path.GetFullPath
Récupérer le nom de fichier et son extension à partir d'un chemin d'accès	Méthode Path.GetFileName
Modifier l'extension d'un fichier	Méthode Path.ChangeExtension

Tâches de répertoire courantes

Action à réaliser...	Consultez l'exemple décrit dans cette rubrique...
Accéder à un fichier dans un dossier spécial comme Mes documents	Procédure : écrire du texte dans un fichier
Créer un répertoire	Méthode Directory.CreateDirectory PropriétéFileInfo.Directory
Créer un sous-répertoire	Méthode DirectoryInfo.CreateSubdirectory
Renommer ou déplacer un répertoire	Méthode Directory.Move Méthode DirectoryInfo.MoveTo
Copier un répertoire	Procédure : Copier des répertoires
Supprimer un répertoire	Méthode Directory.Delete Méthode DirectoryInfo.Delete

Action à réaliser...	Consultez l'exemple décrit dans cette rubrique...
Afficher les fichiers et les sous-répertoires d'un répertoire	Procédure : énumérer des répertoires et des fichiers
Rechercher la taille d'un répertoire	Classe System.IO.Directory
Déterminer si un répertoire existe	Méthode Directory.Exists

Voir aussi

- Fichier et flux de données E/S
- Composition de flux
- E/S sur fichier asynchrones

Guide pratique pour lire et écrire dans un fichier de données nouvellement créé

Article • 03/06/2023

Les classes System.IO.BinaryWriter et System.IO.BinaryReader sont utilisées pour écrire et lire des données autres que des chaînes de caractères. L'exemple suivant montre comment créer un flux de fichier vide, y écrire des données, puis les lire.

L'exemple créé un fichier de données appelé *Test.data* dans le répertoire actif, crée les objets BinaryWriter et BinaryReader associés, puis utilise l'objet BinaryWriter pour écrire les entiers de 0 à 10 dans *Test.data*, ce qui laisse le pointeur de fichier à la fin du fichier. Ensuite, l'objet BinaryReader replace le pointeur de fichier au début, puis lit le contenu spécifié.

> ⓘ **Notes**
>
> Si *Test.data* existe déjà dans le répertoire actif, une exception **IOException** est levée. Utilisez l'option de mode de fichier **FileMode.Create** plutôt que **FileMode.CreateNew** pour toujours créer un fichier sans lever d'exception.

Exemple

C#

```
using System;
using System.IO;

class MyStream
{
    private const string FILE_NAME = "Test.data";

    public static void Main()
    {
        if (File.Exists(FILE_NAME))
        {
            Console.WriteLine($"{FILE_NAME} already exists!");
            return;
        }

        using (FileStream fs = new FileStream(FILE_NAME,
FileMode.CreateNew))
```

```
        {
            using (BinaryWriter w = new BinaryWriter(fs))
            {
                for (int i = 0; i < 11; i++)
                {
                    w.Write(i);
                }
            }
        }

        using (FileStream fs = new FileStream(FILE_NAME, FileMode.Open,
FileAccess.Read))
        {
            using (BinaryReader r = new BinaryReader(fs))
            {
                for (int i = 0; i < 11; i++)
                {
                    Console.WriteLine(r.ReadInt32());
                }
            }
        }
    }
}


// The example creates a file named "Test.data" and writes the integers 0
through 10 to it in binary format.
// It then writes the contents of Test.data to the console with each integer
on a separate line.
```

Voir aussi

- BinaryReader
- BinaryWriter
- FileStream
- FileStream.Seek
- SeekOrigin
- Guide pratique pour énumérer des répertoires et des fichiers
- Guide pratique pour ouvrir un fichier journal et y ajouter des éléments
- Guide pratique pour lire du texte dans un fichier
- Guide pratique pour écrire du texte dans un fichier
- Guide pratique pour lire les caractères d'une chaîne
- Guide pratique pour écrire des caractères dans une chaîne
- E/S de fichier et de flux

Comment : copier des répertoires

Article • 23/06/2023

Cet article montre comment utiliser les classes d'E/S pour copier de manière synchrone le contenu d'un répertoire vers un autre emplacement.

Pour obtenir un exemple de copie de fichier asynchrone, consultez E/S sur fichier asynchrones.

Cet exemple copie des sous-répertoires en définissant le paramètre `recursive` de la méthode `CopyDirectory` sur `true`. La méthode `CopyDirectory` copie les sous-répertoires de manière récursive en s'appelant elle-même sur chaque sous-répertoire jusqu'à ce qu'il n'y ait plus rien à copier.

Exemple

C#

```
using System.IO;

CopyDirectory(@".\", @".\copytest", true);

static void CopyDirectory(string sourceDir, string destinationDir, bool
recursive)
{
    // Get information about the source directory
    var dir = new DirectoryInfo(sourceDir);

    // Check if the source directory exists
    if (!dir.Exists)
        throw new DirectoryNotFoundException($"Source directory not found:
{dir.FullName}");

    // Cache directories before we start copying
    DirectoryInfo[] dirs = dir.GetDirectories();

    // Create the destination directory
    Directory.CreateDirectory(destinationDir);

    // Get the files in the source directory and copy to the destination
directory
    foreach (FileInfo file in dir.GetFiles())
    {
        string targetFilePath = Path.Combine(destinationDir, file.Name);
        file.CopyTo(targetFilePath);
    }

    // If recursive and copying subdirectories, recursively call this method
```

```
    if (recursive)
    {
        foreach (DirectoryInfo subDir in dirs)
        {
            string newDestinationDir = Path.Combine(destinationDir,
subDir.Name);
            CopyDirectory(subDir.FullName, newDestinationDir, true);
        }
    }
}
```

Voir aussi

- FileInfo
- DirectoryInfo
- FileStream
- E/S de fichier et de flux
- Tâches d'E/S courantes
- E/S sur fichier asynchrones

Registry Classe

Référence

Définition

Espace de noms: Microsoft.Win32

Assembly: Microsoft.Win32.Registry.dll

Fournit des objets RegistryKey qui représentent les clés racine du Registre Windows et des méthodes `static` permettant d'accéder à des paires clé/valeur.

C#

```
public static class Registry
```

Héritage Object → Registry

Exemples

Cette section contient deux exemples de code. Le premier exemple illustre les clés racines, et le second illustre les `static`GetValue méthodes et SetValue .

Exemple 1

L'exemple de code suivant montre comment récupérer les sous-clés de la clé HKEY_USERS et imprimer leurs noms à l'écran. Utilisez la OpenSubKey méthode pour créer un instance de la sous-clé particulière qui vous intéresse. Vous pouvez ensuite utiliser d'autres opérations dans `RegistryKey` pour manipuler cette clé.

C#

```
using System;
using Microsoft.Win32;

class Reg {
    public static void Main() {

        // Create a RegistryKey, which will access the HKEY_USERS
        // key in the registry of this machine.
        RegistryKey rk = Registry.Users;

        // Print out the keys.
        PrintKeys(rk);
```

```
    }

    static void PrintKeys(RegistryKey rkey) {

        // Retrieve all the subkeys for the specified key.
        string [] names = rkey.GetSubKeyNames();

        int icount = 0;

        Console.WriteLine("Subkeys of " + rkey.Name);
        Console.WriteLine("-----------------------------------------------
");

        // Print the contents of the array to the console.
        foreach (string s in names) {
            Console.WriteLine(s);

            // The following code puts a limit on the number
            // of keys displayed.  Comment it out to print the
            // complete list.
            icount++;
            if (icount >= 10)
                break;
        }
    }
}
```

Exemple 2

L'exemple de code suivant stocke des valeurs de plusieurs types de données dans un exemple de clé, en créant la clé comme il le fait, puis récupère et affiche les valeurs. L'exemple illustre le stockage et la récupération de la paire nom/valeur par défaut (sans nom), ainsi que l'utilisation de `defaultValue` quand une paire nom/valeur n'existe pas.

C#

```
using System;
using Microsoft.Win32;

public class Example
{
    public static void Main()
    {
        // The name of the key must include a valid root.
        const string userRoot = "HKEY_CURRENT_USER";
        const string subkey = "RegistrySetValueExample";
        const string keyName = userRoot + "\\" + subkey;

        // An int value can be stored without specifying the
        // registry data type, but long values will be stored
        // as strings unless you specify the type. Note that
        // the int is stored in the default name/value
```

```
        // pair.
        Registry.SetValue(keyName, "", 5280);
        Registry.SetValue(keyName, "TestLong", 12345678901234,
            RegistryValueKind.QWord);

        // Strings with expandable environment variables are
        // stored as ordinary strings unless you specify the
        // data type.
        Registry.SetValue(keyName, "TestExpand", "My path: %path%");
        Registry.SetValue(keyName, "TestExpand2", "My path: %path%",
            RegistryValueKind.ExpandString);

        // Arrays of strings are stored automatically as
        // MultiString. Similarly, arrays of Byte are stored
        // automatically as Binary.
        string[] strings = {"One", "Two", "Three"};
        Registry.SetValue(keyName, "TestArray", strings);

        // Your default value is returned if the name/value pair
        // does not exist.
        string noSuch = (string) Registry.GetValue(keyName,
            "NoSuchName",
            "Return this default if NoSuchName does not exist.");
        Console.WriteLine("\r\nNoSuchName: {0}", noSuch);

        // Retrieve the int and long values, specifying
        // numeric default values in case the name/value pairs
        // do not exist. The int value is retrieved from the
        // default (nameless) name/value pair for the key.
        int tInteger = (int) Registry.GetValue(keyName, "", -1);
        Console.WriteLine("(Default): {0}", tInteger);
        long tLong = (long) Registry.GetValue(keyName, "TestLong",
            long.MinValue);
        Console.WriteLine("TestLong: {0}", tLong);

        // When retrieving a MultiString value, you can specify
        // an array for the default return value.
        string[] tArray = (string[]) Registry.GetValue(keyName,
            "TestArray",
            new string[] {"Default if TestArray does not exist."});
        for(int i=0; i<tArray.Length; i++)
        {
            Console.WriteLine("TestArray({0}): {1}", i, tArray[i]);
        }

        // A string with embedded environment variables is not
        // expanded if it was stored as an ordinary string.
        string tExpand = (string) Registry.GetValue(keyName,
             "TestExpand",
             "Default if TestExpand does not exist.");
        Console.WriteLine("TestExpand: {0}", tExpand);

        // A string stored as ExpandString is expanded.
        string tExpand2 = (string) Registry.GetValue(keyName,
            "TestExpand2",
```

```
            "Default if TestExpand2 does not exist.");
        Console.WriteLine("TestExpand2: {0}...",
            tExpand2.Substring(0, 40));

        Console.WriteLine("\r\nUse the registry editor to examine the
key.");
        Console.WriteLine("Press the Enter key to delete the key.");
        Console.ReadLine();
        Registry.CurrentUser.DeleteSubKey(subkey);
    }
}
//
// This code example produces output similar to the following:
//
//NoSuchName: Return this default if NoSuchName does not exist.
//(Default): 5280
//TestLong: 12345678901234
//TestArray(0): One
//TestArray(1): Two
//TestArray(2): Three
//TestExpand: My path: %path%
//TestExpand2: My path: D:\Program Files\Microsoft.NET\...
//
//Use the registry editor to examine the key.
//Press the Enter key to delete the key.
```

Remarques

Cette classe fournit l'ensemble des clés racines standard trouvées dans le Registre sur les machines exécutant Windows. Le Registre est une installation de stockage qui permet d'obtenir des informations sur les applications, les utilisateurs et les paramètres système par défaut. Par exemple, les applications peuvent utiliser le Registre pour stocker les informations qui doivent être conservées après la fermeture de l'application et accéder à ces mêmes informations lorsque l'application est rechargée. Pour instance, vous pouvez stocker les préférences de couleur, les emplacements d'écran ou la taille de la fenêtre. Vous pouvez contrôler ces données pour chaque utilisateur en stockant les informations dans un autre emplacement dans le Registre.

Les instances de base ou racine RegistryKey exposées par la `Registry` classe délimitent le mécanisme de stockage de base pour les sous-clés et les valeurs dans le Registre. Toutes les clés sont en lecture seule, car le registre dépend de leur existence. Les clés exposées par `Registry` sont les suivantes :

CurrentUser Stocke des informations sur les préférences de l'utilisateur.

LocalMachine Stocke les informations de configuration de l'ordinateur local.

ClassesRoot Stocke des informations sur les types (et classes) et leurs propriétés.

Users Stocke des informations sur la configuration utilisateur par défaut.

PerformanceData Stocke des informations sur les performances des composants logiciels.

CurrentConfig Stocke des informations matérielles non spécifiques à l'utilisateur.

DynData Stocke les données dynamiques.

Une fois que vous avez identifié la clé racine sous laquelle vous souhaitez stocker/récupérer des informations à partir du Registre, vous pouvez utiliser la RegistryKey classe pour ajouter ou supprimer des sous-clés et manipuler les valeurs d'une clé donnée.

Les appareils matériels peuvent placer automatiquement des informations dans le registre à l'aide de l'interface Plug-and-Play. Les logiciels permettant d'installer des pilotes de périphérique peuvent placer des informations dans le Registre en écrivant dans les API standard.

Méthodes statiques pour obtenir et définir des valeurs

La Registry classe contient également des `static` GetValue méthodes et SetValue pour définir et récupérer des valeurs à partir de clés de Registre. Ces méthodes ouvrent et ferment des clés de Registre chaque fois qu'elles sont utilisées, de sorte qu'elles ne fonctionnent pas aussi bien que les méthodes analogues dans la RegistryKey classe, lorsque vous accédez à un grand nombre de valeurs.

La RegistryKey classe fournit également des méthodes qui vous permettent de définir la sécurité du contrôle d'accès Windows pour les clés de Registre, de tester le type de données d'une valeur avant de la récupérer et de supprimer des clés.

Champs

ClassesRoot	Définit les types (ou classes) de documents ainsi que les propriétés associées à ces types. Ce champ lit la clé de base HKEY_CLASSES_ROOT du Registre Windows.
CurrentConfig	Contient des informations de configuration concernant le matériel qui ne sont pas spécifiques à l'utilisateur. Ce champ lit la clé de base HKEY_CURRENT_CONFIG du Registre Windows.

CurrentUser	Contient les informations relatives aux préférences de l'utilisateur en cours. Ce champ lit la clé de base HKEY_CURRENT_USER du Registre Windows.
LocalMachine	Contient les informations de configuration pour l'ordinateur local. Ce champ lit la clé de base HKEY_LOCAL_MACHINE du Registre Windows.
PerformanceData	Contient des informations sur les performances des composants logiciels. Ce champ lit la clé de base HKEY_PERFORMANCE_DATA du Registre Windows.
Users	Contient les informations relatives à la configuration utilisateur par défaut. Ce champ lit la clé de base HKEY_USERS du Registre Windows.

Méthodes

GetValue(String, String, Object)	Récupère la valeur associée au nom spécifié, dans la clé de Registre spécifiée. Si le nom est introuvable dans la clé spécifiée, retourne une valeur par défaut que vous fournissez ou la valeur `null` si la clé spécifiée n'existe pas.
SetValue(String, String, Object)	Définit la paire nom/valeur spécifiée sur la clé de Registre spécifiée. Si la clé spécifiée n'existe pas, elle est créée.
SetValue(String, String, Object, RegistryValueKind)	Définit la paire nom/valeur sur la clé de Registre spécifiée, à l'aide du type de données de Registre spécifié. Si la clé spécifiée n'existe pas, elle est créée.

S’applique à

Produit	Versions
.NET	Core 1.0, Core 1.1, 6, 7, 8
.NET Framework	1.1, 2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Platform Extensions	2.1, 2.2, 3.0, 3.1, 5
Windows Desktop	3.0, 3.1, 5
Xamarin.iOS	10.8
Xamarin.Mac	3.0

Voir aussi

- RegistryHive
- RegistryKey

Interoperability Overview

Article • 02/25/2023

Interoperability enables you to preserve and take advantage of existing investments in unmanaged code. Code that runs under the control of the common language runtime (CLR) is *managed code*, and code that runs outside the CLR is *unmanaged code*. COM, COM+, C++ components, ActiveX components, and Microsoft Windows API are examples of unmanaged code.

.NET enables interoperability with unmanaged code through platform invoke services, the System.Runtime.InteropServices namespace, C++ interoperability, and COM interoperability (COM interop).

Platform Invoke

Platform invoke is a service that enables managed code to call unmanaged functions implemented in dynamic link libraries (DLLs), such as the Microsoft Windows API. It locates and invokes an exported function and marshals its arguments (integers, strings, arrays, structures, and so on) across the interoperation boundary as needed.

For more information, see Consuming Unmanaged DLL Functions and How to use platform invoke to play a WAV file.

> **Note**
>
> The **Common Language Runtime** (CLR) manages access to system resources. Calling unmanaged code that is outside the CLR bypasses this security mechanism, and therefore presents a security risk. For example, unmanaged code might call resources in unmanaged code directly, bypassing CLR security mechanisms. For more information, see **Security in .NET**.

C++ Interop

You can use C++ interop, also known as It Just Works (IJW), to wrap a native C++ class. C++ interop enables code authored in C# or another .NET language to access it. You write C++ code to wrap a native DLL or COM component. Unlike other .NET languages, Visual C++ has interoperability support that enables managed and unmanaged code in the same application and even in the same file. You then build the C++ code by using the **/clr** compiler switch to produce a managed assembly. Finally, you add a reference to

the assembly in your C# project and use the wrapped objects just as you would use other managed classes.

Exposing COM Components to C#

You can consume a COM component from a C# project. The general steps are as follows:

1. Locate a COM component to use and register it. Use regsvr32.exe to register or un–register a COM DLL.
2. Add to the project a reference to the COM component or type library. When you add the reference, Visual Studio uses the Tlbimp.exe (Type Library Importer), which takes a type library as input, to output a .NET interop assembly. The assembly, also named a runtime callable wrapper (RCW), contains managed classes and interfaces that wrap the COM classes and interfaces that are in the type library. Visual Studio adds to the project a reference to the generated assembly.
3. Create an instance of a class defined in the RCW. Creating an instance of that class creates an instance of the COM object.
4. Use the object just as you use other managed objects. When the object is reclaimed by garbage collection, the instance of the COM object is also released from memory.

For more information, see Exposing COM Components to the .NET Framework.

Exposing C# to COM

COM clients can consume C# types that have been correctly exposed. The basic steps to expose C# types are as follows:

1. Add interop attributes in the C# project. You can make an assembly COM visible by modifying C# project properties. For more information, see Assembly Information Dialog Box.
2. Generate a COM type library and register it for COM usage. You can modify C# project properties to automatically register the C# assembly for COM interop. Visual Studio uses the Regasm.exe (Assembly Registration Tool), using the `/tlb` command-line switch, which takes a managed assembly as input, to generate a type library. This type library describes the `public` types in the assembly and adds registry entries so that COM clients can create managed classes.

For more information, see Exposing .NET Framework Components to COM and Example COM Class.

See also

- Improving Interop Performance
- Introduction to Interoperability between COM and .NET
- Introduction to COM Interop in Visual Basic
- Marshaling between Managed and Unmanaged Code
- Interoperating with Unmanaged Code

Interoperability Overview

Article • 02/25/2023

Interoperability enables you to preserve and take advantage of existing investments in unmanaged code. Code that runs under the control of the common language runtime (CLR) is *managed code*, and code that runs outside the CLR is *unmanaged code*. COM, COM+, C++ components, ActiveX components, and Microsoft Windows API are examples of unmanaged code.

.NET enables interoperability with unmanaged code through platform invoke services, the System.Runtime.InteropServices namespace, C++ interoperability, and COM interoperability (COM interop).

Platform Invoke

Platform invoke is a service that enables managed code to call unmanaged functions implemented in dynamic link libraries (DLLs), such as the Microsoft Windows API. It locates and invokes an exported function and marshals its arguments (integers, strings, arrays, structures, and so on) across the interoperation boundary as needed.

For more information, see Consuming Unmanaged DLL Functions and How to use platform invoke to play a WAV file.

> **Note**
>
> The **Common Language Runtime** (CLR) manages access to system resources. Calling unmanaged code that is outside the CLR bypasses this security mechanism, and therefore presents a security risk. For example, unmanaged code might call resources in unmanaged code directly, bypassing CLR security mechanisms. For more information, see **Security in .NET**.

C++ Interop

You can use C++ interop, also known as It Just Works (IJW), to wrap a native C++ class. C++ interop enables code authored in C# or another .NET language to access it. You write C++ code to wrap a native DLL or COM component. Unlike other .NET languages, Visual C++ has interoperability support that enables managed and unmanaged code in the same application and even in the same file. You then build the C++ code by using the **/clr** compiler switch to produce a managed assembly. Finally, you add a reference to

the assembly in your C# project and use the wrapped objects just as you would use other managed classes.

Exposing COM Components to C#

You can consume a COM component from a C# project. The general steps are as follows:

1. Locate a COM component to use and register it. Use regsvr32.exe to register or un–register a COM DLL.
2. Add to the project a reference to the COM component or type library. When you add the reference, Visual Studio uses the Tlbimp.exe (Type Library Importer), which takes a type library as input, to output a .NET interop assembly. The assembly, also named a runtime callable wrapper (RCW), contains managed classes and interfaces that wrap the COM classes and interfaces that are in the type library. Visual Studio adds to the project a reference to the generated assembly.
3. Create an instance of a class defined in the RCW. Creating an instance of that class creates an instance of the COM object.
4. Use the object just as you use other managed objects. When the object is reclaimed by garbage collection, the instance of the COM object is also released from memory.

For more information, see Exposing COM Components to the .NET Framework.

Exposing C# to COM

COM clients can consume C# types that have been correctly exposed. The basic steps to expose C# types are as follows:

1. Add interop attributes in the C# project. You can make an assembly COM visible by modifying C# project properties. For more information, see Assembly Information Dialog Box.
2. Generate a COM type library and register it for COM usage. You can modify C# project properties to automatically register the C# assembly for COM interop. Visual Studio uses the Regasm.exe (Assembly Registration Tool), using the `/tlb` command-line switch, which takes a managed assembly as input, to generate a type library. This type library describes the `public` types in the assembly and adds registry entries so that COM clients can create managed classes.

For more information, see Exposing .NET Framework Components to COM and Example COM Class.

See also

- Improving Interop Performance
- Introduction to Interoperability between COM and .NET
- Introduction to COM Interop in Visual Basic
- Marshaling between Managed and Unmanaged Code
- Interoperating with Unmanaged Code

Comment accéder aux objets d'interopérabilité Office

Article • 09/03/2023

C# offre des fonctionnalités qui simplifient l'accès aux objets d'API Office. Les nouvelles fonctionnalités incluent les arguments nommés et les arguments facultatifs, un nouveau type appelé `dynamic` et la possibilité de passer des arguments aux paramètres de référence dans les méthodes COM comme s'il s'agissait de paramètres de valeur.

Dans cet article, vous utilisez les nouvelles fonctionnalités pour écrire le code qui crée et affiche une feuille de calcul Microsoft Office Excel. Vous écrivez le code pour ajouter un document Office Word qui contient une icône liée à la feuille de calcul Excel.

Pour effectuer cette procédure pas à pas, Microsoft Office Excel 2007 et Microsoft Office Word 2007, ou versions ultérieures, doivent être installés sur votre ordinateur.

> ⓘ **Notes**
>
> Il est possible que pour certains des éléments de l'interface utilisateur de Visual Studio, votre ordinateur affiche des noms ou des emplacements différents de ceux indiqués dans les instructions suivantes. L'édition de Visual Studio dont vous disposez et les paramètres que vous utilisez déterminent ces éléments. Pour plus d'informations, consultez **Personnalisation de l'IDE**.

> ⓘ **Important**
>
> VSTO est basé sur **.NET Framework**. Les compléments COM peuvent également être écrits avec .NET Framework. Les compléments Office ne peuvent pas être créés avec **.NET Core et .NET 5+**, les dernières versions de .NET. En effet, .NET Core et .NET 5+ ne peuvent pas fonctionner avec .NET Framework dans le même processus, et cela peut entraîner des échecs de chargement des compléments. Vous pouvez continuer d'utiliser .NET Framework pour écrire des compléments VSTO et COM pour Office. Microsoft ne mettra pas à jour VSTO ou la plateforme de compléments COM pour utiliser .NET Core ou .NET 5+. Vous pouvez utiliser .NET Core et .NET 5+, dont ASP.NET Core, pour créer le côté serveur des **compléments web Office**.

Pour créer une application console

1. Démarrez Visual Studio.
2. Dans le menu **Fichier** , pointez sur **Nouveau**, puis sélectionnez **Projet**. La boîte de dialogue **Nouveau projet** apparaît.
3. Dans le volet **Modèles installés**, développez **C#**, puis sélectionnez **Windows**.
4. Vérifiez en haut de la boîte de dialogue **Nouveau projet** que vous avez sélectionné **.NET Framework 4** (ou version ultérieure) comme version cible de .NET Framework.
5. Dans le volet **Modèles**, sélectionnez **Application console**.
6. Attribuez un nom à votre projet dans le champ **Nom**.
7. Sélectionnez **OK**.

Le nouveau projet s'affiche dans l'**Explorateur de solutions**.

Pour ajouter des références

1. Dans l'**Explorateur de solutions**, cliquez avec le bouton droit sur le nom de votre projet, puis sélectionnez **Ajouter une référence**. La boîte de dialogue **Ajouter une référence** s'affiche.
2. Dans la page **Assemblys**, sélectionnez **Microsoft.Office.Interop.Word** dans la liste **Nom du composant**, puis maintenez la touche CTRL enfoncée et sélectionnez **Microsoft.Office.Interop.Excel**. Si vous ne voyez pas les assemblys, vous devrez peut-être les installer. Consultez Guide pratique pour installer des assemblys d'interopérabilité principaux Office.
3. Sélectionnez **OK**.

Pour ajouter les directives using nécessaires

Dans l'**Explorateur de solutions**, cliquez avec le bouton droit sur le fichier *Program.cs*, puis sélectionnez **Afficher le code**. Ajoutez les directives `using` suivantes en début du fichier de code :

C#

```
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

Pour créer une liste de comptes bancaires

Copiez-collez la définition de classe suivante dans **Program.cs**, sous la classe `Program`.

C#

```
public class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

Ajoutez le code suivant à la méthode `Main` pour créer une liste `bankAccounts` contenant deux comptes.

C#

```
// Create a list of accounts.
var bankAccounts = new List<Account> {
    new Account {
                  ID = 345678,
                  Balance = 541.27
                },
    new Account {
                  ID = 1230221,
                  Balance = -127.44
                }
};
```

Pour déclarer une méthode qui exporte les informations de compte vers Excel

1. Ajoutez la méthode suivante à la classe `Program` pour définir une feuille de calcul Excel. La méthode Add a un paramètre facultatif pour spécifier un modèle particulier. Les paramètres facultatifs vous permettent d'omettre l'argument du paramètre, si vous souhaitez utiliser la valeur par défaut de ce dernier. Comme vous n'avez pas fourni d'argument, `Add` utilise le modèle par défaut et crée un classeur. L'instruction équivalente dans les versions antérieures de C# nécessite un argument d'espace réservé : `ExcelApp.Workbooks.Add(Type.Missing)`.

C#

```
static void DisplayInExcel(IEnumerable<Account> accounts)
{
    var excelApp = new Excel.Application();
    // Make the object visible.
    excelApp.Visible = true;

    // Create a new, empty workbook and add it to the collection returned
    // by property Workbooks. The new workbook becomes the active workbook.
    // Add has an optional parameter for specifying a particular template.
```

```
    // Because no argument is sent in this example, Add creates a new
workbook.
    excelApp.Workbooks.Add();

    // This example uses a single workSheet. The explicit type casting is
    // removed in a later procedure.
    Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet;
}
```

À la fin de `DisplayInExcel`, ajoutez le code suivant. Le code insère les valeurs dans les deux premières colonnes de la première ligne de la feuille de calcul.

C#

```
// Establish column headings in cells A1 and B1.
workSheet.Cells[1, "A"] = "ID Number";
workSheet.Cells[1, "B"] = "Current Balance";
```

À la fin de `DisplayInExcel`, ajoutez le code suivant. La boucle `foreach` place les informations de la liste des comptes dans les deux premières colonnes des lignes successives de la feuille de calcul.

C#

```
var row = 1;
foreach (var acct in accounts)
{
    row++;
    workSheet.Cells[row, "A"] = acct.ID;
    workSheet.Cells[row, "B"] = acct.Balance;
}
```

À la fin de `DisplayInExcel`, ajoutez le code suivant pour ajuster les largeurs de colonne au contenu.

C#

```
workSheet.Columns[1].AutoFit();
workSheet.Columns[2].AutoFit();
```

Les versions antérieures de C# nécessitent un cast explicite pour ces opérations, car `ExcelApp.Columns[1]` retourne `Object`, et `AutoFit` est une méthode Range d'Excel. Les lignes suivantes illustrent le cast.

C#

```
((Excel.Range)workSheet.Columns[1]).AutoFit();
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

C# convertit automatiquement l'`Object` retourné en `dynamic` si l'assembly est référencé par l'option du compilateur **EmbedInteropTypes** ou si la propriété Excel **Incorporer les types interop** a la valeur true. La valeur par défaut de cette propriété est true.

Pour exécuter le projet

À la fin de `Main`, ajoutez la ligne suivante.

C#

```
// Display the list in an Excel spreadsheet.
DisplayInExcel(bankAccounts);
```

Appuyez sur CTRL+F5. Une feuille de calcul Excel s'affiche avec les données des deux comptes.

Pour ajouter un document Word

Le code suivant ouvre une application Word et crée une icône pointant vers la feuille de calcul Excel. Collez la méthode `CreateIconInWordDoc`, fournie ultérieurement dans cette étape, dans la classe `Program`. `CreateIconInWordDoc` utilise des arguments nommés et facultatifs pour réduire la complexité des appels de méthode à Add et PasteSpecial. Ces appels incorporent deux autres fonctionnalités qui simplifient les appels aux méthodes COM comportant des paramètres de référence. Premièrement, vous pouvez envoyer des arguments aux paramètres de référence comme s'il s'agissait de paramètres de valeur. Autrement dit, vous pouvez envoyer les valeurs directement, sans créer une variable pour chaque paramètre de référence. Le compilateur génère des variables temporaires pour stocker les valeurs d'argument et les ignore lors du retour de l'appel. Ensuite, vous pouvez omettre le mot clé `ref` dans la liste d'arguments.

La méthode `Add` comporte quatre paramètres de référence, tous facultatifs. Vous pouvez omettre les arguments de tout ou partie des paramètres si vous voulez utiliser leur valeur par défaut.

La méthode `PasteSpecial` insère le contenu du Presse-papiers. La méthode comporte sept paramètres de référence, tous facultatifs. Le code suivant spécifie des arguments pour deux d'entre eux : `Link` pour créer un lien vers la source du contenu du Presse-

papiers et `DisplayAsIcon` pour afficher le lien sous forme d'icône. Vous pouvez utiliser des arguments nommés pour ces deux-là et omettre les autres. Bien que ces arguments constituent des paramètres de référence, vous n'avez pas à utiliser le mot clé `ref` ou à créer des variables à envoyer tant qu'arguments. Vous pouvez envoyer les valeurs directement.

C#

```
static void CreateIconInWordDoc()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;

    // The Add method has four reference parameters, all of which are
    // optional. Visual C# allows you to omit arguments for them if
    // the default values are what you want.
    wordApp.Documents.Add();

    // PasteSpecial has seven reference parameters, all of which are
    // optional. This example uses named arguments to specify values
    // for two of the parameters. Although these are reference
    // parameters, you do not need to use the ref keyword, or to create
    // variables to send in as arguments. You can send the values directly.
    wordApp.Selection.PasteSpecial( Link: true, DisplayAsIcon: true);
}
```

À la fin de `Main`, ajoutez l'instruction suivante.

C#

```
// Create a Word document that contains an icon that links to
// the spreadsheet.
CreateIconInWordDoc();
```

À la fin de `DisplayInExcel`, ajoutez l'instruction suivante. La méthode `Copy` ajoute la feuille de calcul au Presse-papiers.

C#

```
// Put the spreadsheet contents on the clipboard. The Copy method has one
// optional parameter for specifying a destination. Because no argument
// is sent, the destination is the Clipboard.
workSheet.Range["A1:B3"].Copy();
```

Appuyez sur CTRL+F5. Un document Word s'affiche avec une icône. Double-cliquez sur l'icône pour afficher la feuille de calcul au premier plan.

Pour définir la propriété Incorporer les types interop

D'autres améliorations sont possibles lorsque vous appelez un type COM qui ne nécessite pas un assembly PIA (Primary Interop Assembly) au moment de l'exécution. La suppression de la dépendance vis-à-vis des assemblys PIA aboutit à l'indépendance des versions et facilite le déploiement. Pour plus d'informations sur les avantages de la programmation sans assemblys PIA, consultez Procédure pas à pas : incorporation de types provenant d'assemblys managés.

En outre, la programmation est plus facile, car le type `dynamic` représente les types requis et retournés déclarés dans les méthodes COM. Les variables de type `dynamic` ne sont pas évaluées avant l'exécution, ce qui élimine la nécessité d'un cast explicite. Pour plus d'informations, consultez Utilisation du type dynamic.

L'incorporation des informations de type au lieu de l'utilisation des assemblys PIA est le comportement par défaut. En raison de ce comportement par défaut, plusieurs des exemples précédents sont simplifiés. Aucun cast explicite n'est requis. Par exemple, la déclaration de `worksheet` dans `DisplayInExcel` est écrite sous la forme `Excel._Worksheet workSheet = excelApp.ActiveSheet` plutôt que sous la forme `Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet`. Les appels à `AutoFit` dans la même méthode nécessiteraient également un cast explicite sans la valeur par défaut, parce que `ExcelApp.Columns[1]` retourne un `Object` et que `AutoFit` est une méthode Excel. Le code suivant illustre le cast.

C#

```
((Excel.Range)workSheet.Columns[1]).AutoFit();
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

Pour modifier la valeur par défaut et utiliser les assemblys PIA au lieu d'incorporer les informations de type, développez le nœud Références dans l'Explorateur de solutions, puis sélectionnez **Microsoft.Office.Interop.Excel** ou **Microsoft.Office.Interop.Word**. Si la fenêtre **Propriétés** n'apparaît pas, appuyez sur **F4**. Recherchez **Incorporer les types interop** dans la liste des propriétés et attribuez-lui la valeur **False**. Vous pouvez aussi compiler à l'aide de l'option du compilateur **References** au lieu de l'option **EmbedInteropTypes** à l'invite de commandes.

Pour ajouter une mise en forme supplémentaire au tableau

Remplacez les deux appels à `AutoFit` de `DisplayInExcel` avec l'instruction suivante.

C#

```
// Call to AutoFormat in Visual C# 2010.
workSheet.Range["A1", "B3"].AutoFormat(
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

La méthode AutoFormat a sept paramètres de valeur, tous facultatifs. Les arguments nommés et les arguments facultatifs vous permettent de fournir des arguments pour tout ou partie des paramètres, voire pour aucun. Dans l'instruction précédente, vous fournissez un argument pour un seul des paramètres, `Format`. Comme `Format` est le premier paramètre dans la liste des paramètres, vous n'avez pas à fournir le nom du paramètre. Cependant, l'instruction peut être plus facile à comprendre si vous incluez le nom du paramètre, comme illustré dans le code suivant.

C#

```
// Call to AutoFormat in Visual C# 2010.
workSheet.Range["A1", "B3"].AutoFormat(Format:
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

Appuyez sur CTRL+F5 pour afficher le résultat. Vous trouverez d'autres formats dans l'énumération XlRangeAutoFormat.

Exemple

L'exemple de code suivant illustre l'exemple complet.

C#

```
using System.Collections.Generic;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;

namespace OfficeProgrammingWalkthruComplete
{
    class Walkthrough
    {
        static void Main(string[] args)
        {
            // Create a list of accounts.
            var bankAccounts = new List<Account>
            {
                new Account {
                              ID = 345678,
```

```
                            Balance = 541.27
                        },
                new Account {
                            ID = 1230221,
                            Balance = -127.44
                        }
            };

            // Display the list in an Excel spreadsheet.
            DisplayInExcel(bankAccounts);

            // Create a Word document that contains an icon that links to
            // the spreadsheet.
            CreateIconInWordDoc();
        }

        static void DisplayInExcel(IEnumerable<Account> accounts)
        {
            var excelApp = new Excel.Application();
            // Make the object visible.
            excelApp.Visible = true;

            // Create a new, empty workbook and add it to the collection
returned
            // by property Workbooks. The new workbook becomes the active
workbook.
            // Add has an optional parameter for specifying a particular
template.
            // Because no argument is sent in this example, Add creates a
new workbook.
            excelApp.Workbooks.Add();

            // This example uses a single workSheet.
            Excel._Worksheet workSheet = excelApp.ActiveSheet;

            // Earlier versions of C# require explicit casting.
            //Excel._Worksheet workSheet =
(Excel.Worksheet)excelApp.ActiveSheet;

            // Establish column headings in cells A1 and B1.
            workSheet.Cells[1, "A"] = "ID Number";
            workSheet.Cells[1, "B"] = "Current Balance";

            var row = 1;
            foreach (var acct in accounts)
            {
                row++;
                workSheet.Cells[row, "A"] = acct.ID;
                workSheet.Cells[row, "B"] = acct.Balance;
            }

            workSheet.Columns[1].AutoFit();
            workSheet.Columns[2].AutoFit();

            // Call to AutoFormat in Visual C#. This statement replaces the
```

```
            // two calls to AutoFit.
            workSheet.Range["A1", "B3"].AutoFormat(
                Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);

            // Put the spreadsheet contents on the clipboard. The Copy
method has one
            // optional parameter for specifying a destination. Because no
argument
            // is sent, the destination is the Clipboard.
            workSheet.Range["A1:B3"].Copy();
        }

        static void CreateIconInWordDoc()
        {
            var wordApp = new Word.Application();
            wordApp.Visible = true;

            // The Add method has four reference parameters, all of which
are
            // optional. Visual C# allows you to omit arguments for them if
            // the default values are what you want.
            wordApp.Documents.Add();

            // PasteSpecial has seven reference parameters, all of which are
            // optional. This example uses named arguments to specify values
            // for two of the parameters. Although these are reference
            // parameters, you do not need to use the ref keyword, or to
create
            // variables to send in as arguments. You can send the values
directly.
            wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);
        }
    }

    public class Account
    {
        public int ID { get; set; }
        public double Balance { get; set; }
    }
}
```

Voir aussi

- Type.Missing
- dynamic
- Arguments nommés et facultatifs
- Comment : utiliser des arguments nommés et facultatifs dans la programmation Office

Comment utiliser des propriétés indexées dans la programmation COM Interop

Article • 09/03/2023

Les propriétés indexées fonctionnent avec d'autres fonctionnalités dans C#, comme les arguments nommés et facultatifs, un nouveau type (dynamique) et les informations de type incorporées, pour améliorer la programmation Microsoft Office.

> **ⓘ Important**
>
> VSTO est basé sur **.NET Framework**. Les compléments COM peuvent également être écrits avec .NET Framework. Les compléments Office ne peuvent pas être créés avec **.NET Core et .NET 5+**, les dernières versions de .NET. En effet, .NET Core et .NET 5+ ne peuvent pas fonctionner avec .NET Framework dans le même processus, et cela peut entraîner des échecs de chargement des compléments. Vous pouvez continuer d'utiliser .NET Framework pour écrire des compléments VSTO et COM pour Office. Microsoft ne mettra pas à jour VSTO ou la plateforme de compléments COM pour utiliser .NET Core ou .NET 5+. Vous pouvez utiliser .NET Core et .NET 5+, dont ASP.NET Core, pour créer le côté serveur des **compléments web Office**.

Dans les versions antérieures de C#, les méthodes sont accessibles comme des propriétés uniquement si la méthode `get` n'a aucun paramètre et que la méthode `set` a un seul et unique paramètre de valeur. Toutefois, toutes les propriétés COM ne respectent pas ces restrictions. Par exemple, la propriété Range[] d'Excel a un accesseur `get`, qui nécessite un paramètre pour le nom de la plage. Dans le passé, parce que vous ne pouviez pas accéder à la propriété `Range` directement, vous deviez utiliser la méthode `get_Range` à la place, comme indiqué dans l'exemple suivant.

{language}

```
// Visual C# 2008 and earlier.
var excelApp = new Excel.Application();
// . . .
Excel.Range targetRange = excelApp.get_Range("A1", Type.Missing);
```

Les propriétés indexées vous permettent d'écrire ce qui suit à la place :

{language}

```
// Visual C# 2010.
var excelApp = new Excel.Application();
// . . .
Excel.Range targetRange = excelApp.Range["A1"];
```

L'exemple précédent utilise également la fonctionnalité des arguments facultatifs, qui vous permet d'omettre `Type.Missing`.

Les propriétés indexées vous permettent d'écrire le code suivant.

{language}

```
// Visual C# 2010.
targetRange.Value = "Name";
```

Vous ne pouvez pas créer vos propres propriétés indexées. La fonctionnalité prend uniquement en charge la consommation de propriétés indexées existantes.

Exemple

L'exemple de code suivant illustre l'exemple complet. Pour plus d'informations sur la configuration d'un projet qui accède à l'API Office, consultez Comment accéder aux objets Office Interop à l'aide des fonctionnalités C#.

{language}

```
// You must add a reference to Microsoft.Office.Interop.Excel to run
// this example.
using System;
using Excel = Microsoft.Office.Interop.Excel;

namespace IndexedProperties
{
    class Program
    {
        static void Main(string[] args)
        {
            CSharp2010();
        }

        static void CSharp2010()
        {
            var excelApp = new Excel.Application();
            excelApp.Workbooks.Add();
            excelApp.Visible = true;

            Excel.Range targetRange = excelApp.Range["A1"];
```

```
            targetRange.Value = "Name";
        }

        static void CSharp2008()
        {
            var excelApp = new Excel.Application();
            excelApp.Workbooks.Add(Type.Missing);
            excelApp.Visible = true;

            Excel.Range targetRange = excelApp.get_Range("A1",
Type.Missing);
            targetRange.set_Value(Type.Missing, "Name");
            // Or
            //targetRange.Value2 = "Name";
        }
    }
}
```

Voir aussi

- Arguments nommés et facultatifs
- dynamic
- Utilisation du type dynamic

Comment utiliser l'appel de code non managé pour lire un fichier WAV

Article • 07/04/2023

L'exemple de code C# suivant explique comment utiliser des services d'appel de code non managé pour lire un fichier WAV sur le système d'exploitation Windows.

Exemple

Cet exemple de code utilise DllImportAttribute pour importer le point d'entrée de la méthode `PlaySound` de `winmm.dll` sous la forme `Form1 PlaySound()`. L'exemple utilise un formulaire Windows Form simple avec un bouton. En cliquant sur le bouton, vous ouvrez une boîte de dialogue OpenFileDialog Windows standard qui permet d'ouvrir un fichier à lire. Quand un fichier audio est sélectionné, il est lu à l'aide de la méthode `PlaySound()` de la bibliothèque *winmm.dll*. Pour plus d'informations sur cette méthode, consultez Utilisation de la fonction PlaySound avec des fichiers audio Waveform. Recherchez et sélectionnez un fichier avec une extension .wav, puis sélectionnez **Ouvrir** pour lire le fichier audio à l'aide de l'appel de code non managé. Une zone de texte affiche le chemin complet du fichier sélectionné.

C#

```
using System.Runtime.InteropServices;

namespace WinSound;

public partial class Form1 : Form
{
    private TextBox textBox1;
    private Button button1;

    public Form1()  // Constructor.
    {
        InitializeComponent();
    }

    [DllImport("winmm.DLL", EntryPoint = "PlaySound", SetLastError = true,
CharSet = CharSet.Unicode, ThrowOnUnmappableChar = true)]
    private static extern bool PlaySound(string szSound, System.IntPtr hMod,
PlaySoundFlags flags);

    [System.Flags]
    public enum PlaySoundFlags : int
    {
        SND_SYNC = 0x0000,
```

```
        SND_ASYNC = 0x0001,
        SND_NODEFAULT = 0x0002,
        SND_LOOP = 0x0008,
        SND_NOSTOP = 0x0010,
        SND_NOWAIT = 0x00002000,
        SND_FILENAME = 0x00020000,
        SND_RESOURCE = 0x00040004
    }

    private void button1_Click(object sender, System.EventArgs e)
    {
        var dialog1 = new OpenFileDialog();

        dialog1.Title = "Browse to find sound file to play";
        dialog1.InitialDirectory = @"c:\";
        //<Snippet5>
        dialog1.Filter = "Wav Files (*.wav)|*.wav";
        //</Snippet5>
        dialog1.FilterIndex = 2;
        dialog1.RestoreDirectory = true;

        if (dialog1.ShowDialog() == DialogResult.OK)
        {
            textBox1.Text = dialog1.FileName;
            PlaySound(dialog1.FileName, new System.IntPtr(),
PlaySoundFlags.SND_SYNC);
        }
    }

    private void Form1_Load(object sender, EventArgs e)
    {
        // Including this empty method in the sample because in the IDE,
        // when users click on the form, generates code that looks for a
default method
        // with this name. We add it here to prevent confusion for those
using the samples.
    }
}
```

La boîte de dialogue **Ouvrir les fichiers** est filtrée pour afficher uniquement les fichiers avec une extension .wav au moyen des paramètres de filtre.

Compilation du code

Créez un projet d'application Windows C# dans Visual Studio et nommez-le **WinSound**. Copiez le code précédent et collez-le sur le contenu du *fichier Form1.cs*. Copiez le code suivant et collez-le dans le fichier *Form1.Designer.cs*, dans le fichier la méthode `InitializeComponent()`, après tout le code existant.

```
C#
```

```
this.button1 = new System.Windows.Forms.Button();
this.textBox1 = new System.Windows.Forms.TextBox();
this.SuspendLayout();
//
// button1
//
this.button1.Location = new System.Drawing.Point(192, 40);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(88, 24);
this.button1.TabIndex = 0;
this.button1.Text = "Browse";
this.button1.Click += new System.EventHandler(this.button1_Click);
//
// textBox1
//
this.textBox1.Location = new System.Drawing.Point(8, 40);
this.textBox1.Name = "textBox1";
this.textBox1.Size = new System.Drawing.Size(168, 20);
this.textBox1.TabIndex = 1;
this.textBox1.Text = "File path";
//
// Form1
//
this.AutoScaleDimensions = new System.Drawing.SizeF(5, 13);
this.ClientSize = new System.Drawing.Size(292, 266);
this.Controls.Add(this.textBox1);
this.Controls.Add(this.button1);
this.Name = "Form1";
this.Text = "Platform Invoke WinSound C#";
this.ResumeLayout(false);
this.PerformLayout();
```

Compilez, puis exécutez le code.

Voir aussi

- Présentation détaillée de l'appel de code non managé
- Marshaling de données à l'aide de l'appel de code managé

Procédure pas à pas : programmation Office en C#

Article • 09/03/2023

C# offre des fonctionnalités qui améliorent la programmation Microsoft Office. Les fonctionnalités utiles de C# incluent des arguments nommés et facultatifs ainsi que des valeurs de retour de type `dynamic`. Dans la programmation COM, vous pouvez omettre le mot clé `ref` et accéder aux propriétés indexées.

Les deux langages autorisent l'incorporation d'informations de type, qui permet de déployer des assemblys interagissant avec les composants COM sans déployer les assemblys PIA (Primary Interop Assemblies) sur l'ordinateur de l'utilisateur. Pour plus d'informations, consultez Procédure pas à pas : incorporation de types provenant d'assemblys managés.

Cette procédure pas à pas illustre ces fonctionnalités dans le contexte de la programmation Office, mais beaucoup d'entre elles sont aussi utiles en programmation générale. Dans la procédure pas à pas, vous allez utiliser une application de complément Excel pour créer un classeur Excel. Vous créerez ensuite un document Word contenant un lien vers le classeur. Enfin, vous apprendrez à activer et désactiver la dépendance d'assembly PIA.

> **ⓘ Important**
>
> VSTO est basé sur **.NET Framework**. Les compléments COM peuvent également être écrits avec .NET Framework. Les compléments Office ne peuvent pas être créés avec **.NET Core et .NET 5+**, les dernières versions de .NET. En effet, .NET Core et .NET 5+ ne peuvent pas fonctionner avec .NET Framework dans le même processus, et cela peut entraîner des échecs de chargement des compléments. Vous pouvez continuer d'utiliser .NET Framework pour écrire des compléments VSTO et COM pour Office. Microsoft ne mettra pas à jour VSTO ou la plateforme de compléments COM pour utiliser .NET Core ou .NET 5+. Vous pouvez utiliser .NET Core et .NET 5+, dont ASP.NET Core, pour créer le côté serveur des **compléments web Office**.

Prérequis

Pour effectuer cette procédure pas à pas, Microsoft Office Excel et Microsoft Office Word doivent être installés sur votre ordinateur.

> **Notes**
>
> Il est possible que pour certains des éléments de l'interface utilisateur de Visual Studio, votre ordinateur affiche des noms ou des emplacements différents de ceux indiqués dans les instructions suivantes. L'édition de Visual Studio dont vous disposez et les paramètres que vous utilisez déterminent ces éléments. Pour plus d'informations, consultez **Personnalisation de l'IDE**.

Configurer une application Complément Excel

1. Démarrez Visual Studio.
2. Dans le menu **Fichier** , pointez sur **Nouveau**, puis sélectionnez **Projet**.
3. Dans le volet **Modèles installés**, développez **C#** et **Office**, puis sélectionnez l'année de version du produit Office.
4. Dans le volet **Modèles**, sélectionnez **Complément Excel <version>**.
5. Regardez en haut du volet **Modèles** pour vérifier que **.NET Framework 4**, ou version ultérieure, apparaît dans la zone **Framework cible**.
6. Tapez un nom pour votre projet dans la zone **Nom**, si vous le souhaitez.
7. Sélectionnez **OK**.
8. Le nouveau projet s'affiche dans l'**Explorateur de solutions**.

Ajouter des références

1. Dans l'**Explorateur de solutions**, cliquez avec le bouton droit sur le nom de votre projet, puis sélectionnez **Ajouter une référence**. La boîte de dialogue **Ajouter une référence** s'affiche.
2. Sous l'onglet **Assemblys**, sélectionnez **Microsoft.Office.Interop.Excel**, version `<version>.0.0.0` (pour plus d'informations sur les numéros de version des produits Office, consultez Versions Microsoft),dans la liste **Nom du composant**, puis maintenez la touche CTRL enfoncée et sélectionnez **Microsoft.Office.Interop.Word**, `version <version>.0.0.0`. Si vous ne voyez pas les assemblys, vous devrez peut-être les installer. Pour plus d'informations, consultez Guide pratique pour installer les assemblys PIA (Primary Interop Assembly) d'Office.
3. Sélectionnez **OK**.

Ajouter les instructions Imports ou les directives using nécessaires

Dans l'**Explorateur de solutions**, cliquez avec le bouton droit sur le fichier **ThisAddIn.cs**, puis sélectionnez **Afficher le code**. Ajoutez les directives `using` (C#) suivantes en haut du fichier de code, si elles ne sont pas déjà présentes.

C#

```
using System.Collections.Generic;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

Créer une liste de comptes bancaires

Dans l'**Explorateur de solutions**, cliquez avec le bouton droit sur le nom de votre projet, sélectionnez **Ajouter** , puis sélectionnez **Classe**. Nommez la classe Account.cs. Sélectionnez **Ajouter**. Remplacez la définition de la classe `Account` par le code suivant : Les définitions de classe utilisent les *propriétés implémentées automatiquement*.

C#

```
class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

Pour créer une liste `bankAccounts` qui contient deux comptes, ajoutez le code suivant à la méthode `ThisAddIn_Startup` dans *ThisAddIn.cs*. Les déclarations de liste utilisent les *initialiseurs de collection*.

C#

```
var bankAccounts = new List<Account>
{
    new Account
    {
        ID = 345,
        Balance = 541.27
    },
    new Account
    {
        ID = 123,
        Balance = -127.44
    }
};
```

Exporter des données vers Excel

Dans le même fichier, ajoutez la méthode suivante à la classe `ThisAddIn`. La méthode configure un classeur Excel, vers lequel elle exporte les données.

C#

```
void DisplayInExcel(IEnumerable<Account> accounts,
           Action<Account, Excel.Range> DisplayFunc)
{
    var excelApp = this.Application;
    // Add a new Excel workbook.
    excelApp.Workbooks.Add();
    excelApp.Visible = true;
    excelApp.Range["A1"].Value = "ID";
    excelApp.Range["B1"].Value = "Balance";
    excelApp.Range["A2"].Select();

    foreach (var ac in accounts)
    {
        DisplayFunc(ac, excelApp.ActiveCell);
        excelApp.ActiveCell.Offset[1, 0].Select();
    }
    // Copy the results to the Clipboard.
    excelApp.Range["A1:B3"].Copy();
}
```

- La méthode Add possède un *paramètre facultatif* pour spécifier un modèle particulier. Les paramètres facultatifs vous permettent d'omettre l'argument du paramètre, si vous souhaitez utiliser la valeur par défaut de ce dernier. Comme il n'y a aucun argument fourni dans l'exemple précédent, `Add` utilise le modèle par défaut et crée un classeur. L'instruction équivalente dans les versions antérieures de C# nécessite un argument d'espace réservé : `excelApp.Workbooks.Add(Type.Missing)`. Pour plus d'informations, consultez Arguments nommés et facultatifs.
- Les propriétés `Range` et `Offset` de l'objet Range utilisent la fonctionnalité des *propriétés indexées*. Cette fonctionnalité vous permet de consommer ces propriétés à partir des types COM à l'aide de la syntaxe C# standard suivante. Les propriétés indexées vous permettent également d'utiliser la propriété `Value` de l'objet `Range`, ce qui rend superflue l'utilisation de la propriété `Value2`. La propriété `Value` est indexée, mais l'index est optionnel. Les arguments facultatifs et les propriétés indexées fonctionnent ensemble dans l'exemple suivant.

C#

```
// Visual C# 2010 provides indexed properties for COM programming.
excelApp.Range["A1"].Value = "ID";
excelApp.ActiveCell.Offset[1, 0].Select();
```

Vous ne pouvez pas créer vos propres propriétés indexées. La fonctionnalité prend uniquement en charge la consommation de propriétés indexées existantes.

À la fin de `DisplayInExcel`, ajoutez le code suivant pour ajuster les largeurs de colonne au contenu.

C#

```
excelApp.Columns[1].AutoFit();
excelApp.Columns[2].AutoFit();
```

Ces ajouts illustrent une autre fonctionnalité de C# : le traitement des valeurs `Object` retournées par les hôtes COM tels qu'Office comme si leur type était dynamic. Les objets COM sont automatiquement traités comme des objets `dynamic` quand **Incorporer les types d'interopérabilité** est défini sur sa valeur par défaut (`True`) ou, de façon équivalente, lorsque vous référencez l'assembly avec l'option de compilateur **EmbedInteropTypes**. Pour plus d'informations sur l'incorporation des types d'interopérabilité, consultez les procédures « Pour rechercher la référence d'assembly PIA » et « Pour restaurer la dépendance d'assembly PIA » plus loin dans cet article. Pour plus d'informations sur `dynamic`, consultez dynamic ou Utilisation du type dynamic.

Appeler DisplayInExcel

Ajoutez le code suivant à la fin de la méthode `ThisAddIn_StartUp`. L'appel à `DisplayInExcel` contient deux arguments. Le premier argument est le nom de la liste des comptes traités. Le deuxième argument est une expression lambda multiligne qui définit comment traiter les données. Les valeurs `ID` et `balance` de chaque compte s'affichent dans des cellules adjacentes et la ligne s'affiche en rouge si le solde est inférieur à zéro. Pour plus d'informations, consultez Expressions lambda.

C#

```
DisplayInExcel(bankAccounts, (account, cell) =>
// This multiline lambda expression sets custom processing rules
// for the bankAccounts.
{
    cell.Value = account.ID;
    cell.Offset[0, 1].Value = account.Balance;
    if (account.Balance < 0)
```

```
    {
        cell.Interior.Color = 255;
        cell.Offset[0, 1].Interior.Color = 255;
    }
});
```

Pour exécuter le programme, appuyez sur F5. Une feuille de calcul Excel s'affiche avec les données des comptes.

Ajouter un document Word

Ajoutez le code suivant à la fin de la méthode `ThisAddIn_StartUp` pour créer un document Word qui contient un lien vers le classeur Excel.

C#

```
var wordApp = new Word.Application();
wordApp.Visible = true;
wordApp.Documents.Add();
wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);
```

Ce code illustre plusieurs fonctionnalités de C# : la possibilité d'omettre le mot clé `ref` dans la programmation COM, les arguments nommés et les arguments facultatifs. La méthode PasteSpecial comporte sept paramètres, qui sont tous des paramètres de référence facultatifs. Les arguments nommés et les arguments facultatifs vous permettent de désigner les paramètres auxquels vous souhaitez accéder par leur nom et d'envoyer des arguments à ces seuls paramètres. Dans cet exemple, les arguments spécifient la création d'un lien vers le classeur dans le Presse-papiers (paramètre `Link`) et l'affichage du lien dans le document Word sous forme d'une icône (paramètre `DisplayAsIcon`). C# vous permet également d'omettre le mot clé `ref` pour ces arguments.

Exécution de l'application

Appuyez sur F5 pour exécuter l'application. Excel démarre et affiche un tableau qui contient les informations des deux comptes de `bankAccounts`. Puis, un document Word apparaît qui contient un lien vers le tableau Excel.

Nettoyer le projet terminé

Dans Visual Studio, sélectionnez **Nettoyer la solution** dans le menu **Générer**. Sinon, le complément s'exécute chaque fois que vous ouvrez Excel sur votre ordinateur.

Rechercher la référence de PIA

1. Réexécutez l'application, mais ne sélectionnez pas **Nettoyer la solution**.
2. Sélectionnez le bouton **Démarrer**. Recherchez **Microsoft Visual Studio <version>** et ouvrez une invite de commandes développeur.
3. Tapez `ildasm` dans la fenêtre Invite de commandes développeur pour Visual Studio, puis appuyez sur Entrée. La fenêtre IL DASM s'affiche.
4. Dans le menu **Fichier** de la fenêtre IL DASM, sélectionnez **Fichier**>**Ouvrir**. Double-cliquez sur **Visual Studio <version>**, puis sur **Projets**. Ouvrez le dossier de votre projet et, dans le dossier bin/Debug, recherchez *nom de votre projet*.dll. Double-cliquez sur *nom de votre projet*.dll. Une nouvelle fenêtre affiche les attributs de votre projet, en plus des références à d'autres modules et assemblys. L'assembly inclut les espaces de noms `Microsoft.Office.Interop.Excel` et `Microsoft.Office.Interop.Word`. Par défaut, dans Visual Studio, le compilateur importe les types dont vous avez besoin à partir d'un assembly PIA référencé dans votre assembly. Pour plus d'informations, consultez Guide pratique pour afficher le contenu d'un assembly.
5. Double-cliquez sur l'icône **MANIFESTE**. Une fenêtre affiche la liste des assemblys contenant les éléments référencés par le projet. `Microsoft.Office.Interop.Excel` et `Microsoft.Office.Interop.Word` ne figurent pas dans la liste. Étant donné que vous avez importé les types dont votre projet a besoin dans votre assembly, vous n'êtes pas obligé d'installer des références à un PIA. L'importation des types dans votre assembly facilite le déploiement. Les PIA n'ont pas besoin d'être présents sur l'ordinateur de l'utilisateur. Une application ne nécessite pas le déploiement d'une version spécifique de PIA. Les applications peuvent fonctionner avec plusieurs versions d'Office, à condition que les API nécessaires existent dans toutes les versions. Le déploiement d'assemblys PIA n'étant plus nécessaire, vous pouvez créer une application dans les scénarios avancés qui fonctionne avec plusieurs versions d'Office, y compris les versions antérieures. Votre code ne peut pas utiliser d'API qui ne sont pas disponibles dans votre version d'Office. Il n'est pas toujours évident de déterminer si une API particulière était disponible dans une version antérieure. L'utilisation de versions antérieures d'Office n'est pas recommandée.
6. Fermez la fenêtre de manifeste et la fenêtre d'assembly.

Restaurer la dépendance de PIA

1. Dans l'**Explorateur de solutions**, sélectionnez le bouton **Afficher tous les fichiers**. Développez le dossier **Références** et sélectionnez **Microsoft.Office.Interop.Excel**. Appuyez sur F4 pour afficher la fenêtre **Propriétés**.
2. Dans la fenêtre **Propriétés**, remplacez la valeur **True** de la propriété **Incorporer les types d'interopérabilité** par **False**.
3. Répétez les étapes 1 et 2 de cette procédure pour `Microsoft.Office.Interop.Word`.
4. Dans C#, commentez les deux appels à `Autofit` à la fin de la méthode `DisplayInExcel`.
5. Appuyez sur F5 pour vérifier que le projet continue de s'exécuter correctement.
6. Répétez les étapes 1 à 3 de la procédure précédente pour ouvrir la fenêtre d'assembly. Notez que `Microsoft.Office.Interop.Word` et `Microsoft.Office.Interop.Excel` ne sont plus dans la liste des assemblys incorporés.
7. Double-cliquez sur l'icône **MANIFESTE** et faites défiler la liste des assemblys référencés. `Microsoft.Office.Interop.Word` et `Microsoft.Office.Interop.Excel` figurent tous deux dans la liste. Comme l'application référence les assemblys PIA Excel et Word, et que la propriété **Incorporer les types d'interopérabilité** a la valeur **False**, les deux assemblys doivent exister sur l'ordinateur de l'utilisateur final.
8. Dans Visual Studio, sélectionnez **Nettoyer la solution** dans le menu **Générer** pour nettoyer le projet achevé.

Voir aussi

- Propriétés implémentées automatiquement (C#)
- Initialiseurs d'objets et de collections
- Arguments nommés et facultatifs
- dynamic
- Utilisation du type dynamic
- Expressions lambda (C#)
- Procédure pas à pas : incorporation d'informations de type provenant d'assemblys Microsoft Office dans Visual Studio
- Procédure pas à pas : incorporation de types provenant d'assemblys managés
- Procédure pas à pas : création de votre premier complément VSTO pour Excel

Exemple de classe COM

Article • 10/05/2023

Le code suivant est un exemple de classe pouvant être exposée en tant qu'objet COM. Après avoir placé ce code dans un fichier .cs et l'avoir ajouté à votre projet, affectez la valeur **True** à la propriété **Inscrire pour COM Interop**. Pour plus d'informations, consultez Guide pratique pour inscrire un composant pour COM Interop.

L'exposition d'objets C# à COM nécessite de déclarer une interface de classe, une « interface d'événements » si nécessaire, et la classe proprement dite. Les membres de classe doivent suivre ces règles pour être visibles par COM :

- La classe doit être publique.
- Les propriétés, méthodes et événements doivent être publics.
- Les propriétés et méthodes doivent être déclarées dans l'interface de classe.
- Les événements doivent être déclarés dans l'interface d'événement.

Les autres membres publics de la classe qui ne sont pas déclarés dans ces interfaces ne sont pas visibles par COM, mais ils sont visibles par d'autres objets .NET. Pour exposer des propriétés et des méthodes à COM, vous devez les déclarer sur l'interface de classe, les marquer avec un attribut `DispId` et les implémenter dans la classe. L'ordre dans lequel vous déclarez les membres dans l'interface correspond à l'ordre utilisé pour la vtable COM. Pour exposer des événements à partir de votre classe, vous devez les déclarer sur l'interface d'événement et les marquer avec un attribut `DispId`. La classe ne doit pas implémenter cette interface.

La classe implémente l'interface de classe ; elle peut implémenter plusieurs interfaces, mais la première implémentation représente l'interface de classe par défaut. Implémentez ici les méthodes et propriétés exposées à COM. Elles doivent être publiques et correspondre aux déclarations dans l'interface de classe. Vous devez aussi déclarer ici les événements déclenchés par la classe. Elles doivent être publiques et correspondre aux déclarations dans l'interface d'événement.

Exemple

C#

```
using System.Runtime.InteropServices;

namespace project_name
{
    [Guid("EAA4976A-45C3-4BC5-BC0B-E474F4C3C83F")]
```

```
    public interface ComClass1Interface
    {
    }

    [Guid("7BD20046-DF8C-44A6-8F6B-687FAA26FA71"),
        InterfaceType(ComInterfaceType.InterfaceIsIDispatch)]
    public interface ComClass1Events
    {
    }

    [Guid("0D53A3E8-E51A-49C7-944E-E72A2064F938"),
        ClassInterface(ClassInterfaceType.None),
        ComSourceInterfaces(typeof(ComClass1Events))]
    public class ComClass1 : ComClass1Interface
    {
    }
}
```

Voir aussi

- Interopérabilité
- Générer, page du Concepteur de projets (C#)

Informations de référence sur C#

Article • 23/02/2023

Cette section fournit des ressources de référence sur les mots clés, les opérateurs, les caractères spéciaux, les directives de préprocesseur, les options du compilateur, et les erreurs et avertissements du compilateur du langage C#.

Contenu de cette section

Mots clés C#
Fournit des liens vers des informations sur la syntaxe et les mots clés C#.

Opérateurs C#
Fournit des liens vers des informations sur la syntaxe et les opérateurs C#.

Caractères spéciaux C#
Fournit des liens vers des informations sur des caractères spéciaux contextuels en C# et leur utilisation.

Directives de préprocesseur C#
Fournit des liens vers des informations sur les commandes de compilateur permettant d'incorporer du code source en C#.

Options du compilateur C#
Inclut des informations sur les options du compilateur et comment les utiliser.

Erreurs du compilateur C#
Inclut des extraits de code qui montrent la cause et la correction des erreurs du compilateur et des avertissements C#.

Spécification du langage C#
Spécification du langage C# 6.0. Il s'agit d'une ébauche de proposition pour le langage C# 6.0. Ce document sera affiné en collaboration avec le comité des normes pour C# de l'ECMA. La version 5.0 a été publiée en décembre 2017 dans le document Standard ECMA-334 5th Edition.

Les fonctionnalités ayant été implémentées dans les versions C# ultérieures à 6.0 sont représentées dans des propositions de spécification du langage. Ces documents décrivent les delta de la spécification du langage afin d'ajouter ces nouvelles fonctionnalités. Ces normes en sont au stade d'ébauche de proposition. Ces spécifications seront affinées et soumises au comité de normalisation de l'ECMA en vue d'un examen formel et de leur incorporation dans une future version de la norme C#.

Propositions de spécification C# 7.0

De nouvelles fonctionnalités ont été implémentées dans C# 7.0 : critères spéciaux, fonctions locales, déclarations de variables out, expressions throw, littéraux binaires et séparateurs numériques notamment. Ce dossier contient les spécifications de chacune de ces fonctionnalités.

Propositions de spécification C# 7.1

De nouvelles fonctionnalités ont été ajoutées à C# 7.1. D'abord, vous pouvez écrire une méthode `Main` qui renvoie `Task` ou `Task<int>`. Ceci vous permet d'ajouter le modificateur `async` à `Main`. L'expression `default` peut être utilisée sans type dans des emplacements où le type peut être déduit. De plus, les noms de membres de tuple peuvent être déduits. Enfin, les critères spéciaux peuvent être utilisés avec des génériques.

Propositions de spécification C# 7.2

C# 7.2 apporte un certain nombre de petites fonctionnalités. Vous pouvez envoyer des arguments par référence en lecture seule à l'aide du mot clé `in`. Il existe un certain nombre de modifications de bas niveau pour soutenir la sécurité au moment de la compilation pour `Span` et d'autres types associés. Vous pouvez utiliser les arguments nommés où des arguments ultérieurs seront positionnels dans certains cas. Le modificateur d'accès `private protected` vous permet de spécifier que les appelants sont limités à des types dérivés implémentés dans le même assembly. L'opérateur `?:` peut se servir d'une référence pour une variable. Vous pouvez aussi formater des nombres binaires et hexadécimaux à l'aide d'un séparateur numérique de début.

Propositions de spécification C# 7.3

C#7.3 est une nouvelle version qui comprend plusieurs petites mises à jour. Vous pouvez utiliser de nouvelles contraintes sur les paramètres de type générique. D'autres modifications facilitent l'utilisation des champs `fixed`, dont l'utilisation des répartitions stackalloc. Les variables locales déclarées avec le mot clé `ref` peuvent être réassignées pour faire référence à un nouveau stockage. Vous pouvez placer des attributs sur les propriétés implémentées automatiquement qui ciblent le champ de stockage généré par le compilateur. Les variables d'expression peuvent être utilisées dans les initialiseurs. Les tuples peuvent être comparés pour savoir s'ils sont égaux (ou inégaux). Il y a également eu des améliorations pour la résolution de surcharge.

Propositions de spécification C# 8.0

C# 8.0 est disponible avec .NET Core 3.0. Les fonctionnalités comprennent les types référence nullables, la correspondance de modèle récursif, les flux asynchrones, les plages et index, les déclarations using et using basées sur un modèle, l'attribution de fusion Null et les membres d'instance en lecture seule.

Propositions de spécification C# 9

C# 9 est disponible avec .NET 5. Les fonctionnalités incluent les enregistrements, les instructions de niveau supérieur, les améliorations de correspondance de modèle, les méthodes setter d'initialisation uniquement, les nouvelles expressions de type cible, les initialiseurs de module, l'extension de méthodes partielles, les fonctions anonymes statiques, les expressions conditionnelles de type cible, les types de retour covariant, l'extension GetEnumerator dans les boucles foreach, les paramètres d'abandon lambda, les attributs sur des fonctions locales, les entiers de taille native, les pointeurs de fonction, la suppression de l'émission de l'indicateur localsinit et les annotations des paramètres de type sans contrainte.

Propositions de spécification C# 10

C# 10 est disponible avec .NET 6. Les fonctionnalités incluent les structs d'enregistrement, les constructeurs de struct sans paramètre, les directives using globales, les espaces de noms limités aux fichiers, les modèles de propriétés étendues, les chaînes interpolées améliorées, les chaînes interpolées constantes, les améliorations lambda, l'expression d'argument de l'appelant, les directives `#line` améliorées, les attributs génériques, l'amélioration de l'analyse des affectations définies et le remplacement `AsyncMethodBuilder`.

Sections connexes

Utilisation de l'environnement de développement Visual Studio pour C#

Fournit des liens vers des des rubriques de concepts et de tâches qui décrivent l'IDE et l'éditeur.

Guide de programmation C#

Inclut des informations sur l'utilisation du langage de programmation C#.

Gestion des versions du langage C#

Article • 10/05/2023

Le compilateur C# le plus récent détermine une version du langage par défaut en fonction du ou des frameworks cibles de votre projet. Visual Studio ne fournit pas d'interface utilisateur pour modifier la valeur, mais vous pouvez la changer en modifiant le *fichier csproj*. Le choix par défaut garantit que vous utilisez la dernière version de langage compatible avec votre framework cible. Vous bénéficiez d'un accès aux dernières fonctionnalités de langage compatibles avec la cible de votre projet. Ce choix par défaut garantit également que vous n'utilisez pas de langage qui nécessite des types ou un comportement d'exécution non disponibles dans votre framework cible. Le choix d'une version de langage plus récente que la version par défaut peut entraîner des erreurs de compilation et d'exécution difficiles à diagnostiquer.

C# 11 est pris en charge uniquement sur .NET 7 et les versions plus récentes. C# 10 est pris en charge uniquement sur .NET 6 et les versions plus récentes. C# 9 est pris en charge uniquement sur .NET 5 et les versions plus récentes.

Consultez la page de compatibilité de la plateforme Visual Studio pour plus d'informations sur les versions de .NET prises en charge par les versions de Visual Studio. Consultez la page de compatibilité de la plateforme Visual Studio pour Mac pour plus d'informations sur les versions de .NET prises en charge par les versions de Visual Studio pour Mac. Consultez la page Mono dédiée à C# pour connaître la compatibilité de Mono avec les versions de C#.

Valeurs par défaut

Le compilateur détermine une valeur par défaut en fonction de ces règles :

Cible	Version	Version du langage C# par défaut
.NET	7.x	C# 11
.NET	6.x	C# 10
.NET	5.x	C# 9.0
.NET Core	3.x	C# 8.0
.NET Core	2.x	C# 7.3
.NET Standard	2.1	C# 8.0

Cible	Version	Version du langage C# par défaut
.NET Standard	2.0	C# 7.3
.NET Standard	1.x	C# 7.3
.NET Framework	all	C# 7.3

Lorsqu'il s'agit d'une préversion cible pour laquelle il existe une préversion du langage correspondante, c'est cette dernière qui est utilisée. Vous utilisez les fonctionnalités les plus récentes avec cette préversion dans n'importe quel environnement, sans que cela affecte les projets qui ciblent une version .NET Core publiée.

> **ⓘ Important**
>
> Le nouveau modèle de projet pour Visual Studio 2017 a ajouté une entrée `<LangVersion>latest</LangVersion>` aux nouveaux fichiers projet. Si vous mettez à niveau le framework cible pour ces projets, ils remplacent **le comportement par défaut**. Vous devez supprimer le fichier `<LangVersion>latest</LangVersion>` de votre projet lorsque vous mettez à jour le SDK .NET. Ensuite, votre projet utilise la version du compilateur recommandée pour votre framework cible. Vous pouvez mettre à jour le framework cible pour accéder aux fonctionnalités de langage plus récentes.

Remplacer les valeurs par défaut

Si vous devez spécifier votre version C# explicitement, vous pouvez le faire de plusieurs façons :

- Modifiez manuellement votre fichier projet.
- Définissez la version du langage pour plusieurs projets d'un sous-répertoire.
- Configurez l'option de compilateur **LangVersion**.

> **Conseil**
>
> Vous pouvez voir la version du langage dans Visual Studio dans la page des propriétés du projet. Sous l'onglet *Build*, le volet *Avancé* affiche la version sélectionnée.
>
> Pour savoir quelle version de langage vous utilisez actuellement, placez `#error version` (en respectant la casse) dans votre code. En conséquence, le compilateur

signale une erreur de compilateur CS8304 avec un message contenant la version du compilateur utilisée et la version de langage sélectionnée actuelle. Pour plus d'informations, consultez **#error (référence C#)**.

Modifier le fichier projet

Vous pouvez définir la version du langage dans votre fichier projet. Par exemple, si vous souhaitez accéder explicitement aux fonctionnalités d'évaluation, ajoutez un élément comme suit :

XML

```xml
<PropertyGroup>
   <LangVersion>preview</LangVersion>
</PropertyGroup>
```

La valeur `preview` utilise la dernière préversion disponible du langage C# que prend en charge votre compilateur.

Configurer plusieurs projets

Pour configurer plusieurs projets, vous pouvez créer un fichier **Directory.Build.props** qui contient l'élément `<LangVersion>`. En règle générale, vous faites cela dans votre répertoire de solution. Ajoutez le code suivant à un fichier **Directory.build.props** dans votre répertoire de solution :

XML

```xml
<Project>
 <PropertyGroup>
   <LangVersion>preview</LangVersion>
 </PropertyGroup>
</Project>
```

Les builds de tous les sous-répertoires du répertoire contenant ce fichier vont utiliser la préversion de C#. Pour plus d'informations, consultez Personnaliser votre build.

Informations de référence sur la version du langage C#

Le tableau suivant montre toutes les versions actuelles du langage C#. Votre compilateur peut ne pas nécessairement comprendre chaque valeur s'il est plus ancien. Si vous installez le dernier SDK .NET, vous avez accès à tous les éléments répertoriés.

Valeur	Signification
`preview`	Le compilateur accepte toute la syntaxe de langage valide de la dernière préversion.
`latest`	Le compilateur accepte la syntaxe de la dernière version publiée du compilateur (versions mineures incluses).
`latestMajor` ou `default`	Le compilateur accepte la syntaxe de la dernière version principale publiée du compilateur.
`11.0`	Le compilateur accepte uniquement la syntaxe incluse dans C# 11 ou une version antérieure.
`10.0`	Le compilateur accepte uniquement la syntaxe incluse dans C# 10 ou une version antérieure.
`9.0`	Le compilateur accepte uniquement la syntaxe incluse dans C# 9 ou une version antérieure.
`8.0`	Le compilateur accepte uniquement la syntaxe incluse dans C# 8.0 ou une version antérieure.
`7.3`	Le compilateur accepte uniquement la syntaxe incluse dans C# 7.3 ou une version antérieure.
`7.2`	Le compilateur accepte uniquement la syntaxe incluse dans C# 7.2 ou une version antérieure.
`7.1`	Le compilateur accepte uniquement la syntaxe incluse dans C# 7.1 ou une version antérieure.
`7`	Le compilateur accepte uniquement la syntaxe incluse dans C# 7.0 ou une version antérieure.
`6`	Le compilateur accepte uniquement la syntaxe incluse dans C# 6.0 ou une version antérieure.
`5`	Le compilateur accepte uniquement la syntaxe incluse dans C# 5.0 ou une version antérieure.
`4`	Le compilateur accepte uniquement la syntaxe incluse dans C# 4.0 ou une version antérieure.
`3`	Le compilateur accepte uniquement la syntaxe incluse dans C# 3.0 ou une version antérieure.

Valeur	Signification
`ISO-2` ou `2`	Le compilateur accepte uniquement la syntaxe incluse dans ISO/IEC 23270:2006 C# (2.0)
`ISO-1` ou `1`	Le compilateur accepte uniquement la syntaxe incluse dans ISO/IEC 23270:2003 C# (1.0/1.2)

Types valeur (référence C#)

Article • 07/04/2023

Les types valeur et les types référence sont les deux principales catégories de types C#. Une variable d'un type valeur contient une instance du type. Ce type s'oppose au type référence, qui contient une référence à une instance du type. Par défaut, lors de l'affectation, en passant un argument à une méthode et en retournant un résultat de méthode, les valeurs de variable sont copiées. Dans le cas de variables de type valeur, les instances de type correspondantes sont copiées. L'exemple suivant illustre ce comportement :

C#

```
using System;

public struct MutablePoint
{
    public int X;
    public int Y;

    public MutablePoint(int x, int y) => (X, Y) = (x, y);

    public override string ToString() => $"({X}, {Y})";
}

public class Program
{
    public static void Main()
    {
        var p1 = new MutablePoint(1, 2);
        var p2 = p1;
        p2.Y = 200;
        Console.WriteLine($"{nameof(p1)} after {nameof(p2)} is modified:
{p1}");
        Console.WriteLine($"{nameof(p2)}: {p2}");

        MutateAndDisplay(p2);
        Console.WriteLine($"{nameof(p2)} after passing to a method: {p2}");
    }

    private static void MutateAndDisplay(MutablePoint p)
    {
        p.X = 100;
        Console.WriteLine($"Point mutated in a method: {p}");
    }
}
// Expected output:
// p1 after p2 is modified: (1, 2)
// p2: (1, 200)
```

```
// Point mutated in a method: (100, 200)
// p2 after passing to a method: (1, 200)
```

Comme le montre l'exemple précédent, les opérations sur une variable de type valeur affectent uniquement cette instance du type valeur, stockée dans la variable.

Si un type valeur contient un membre de données d'un type référence, seule la référence à l'instance du type référence est copiée lorsqu'une instance de type valeur est copiée. La copie et l'instance de type valeur d'origine ont accès à la même instance de type référence. L'exemple suivant illustre ce comportement :

C#

```
using System;
using System.Collections.Generic;

public struct TaggedInteger
{
    public int Number;
    private List<string> tags;

    public TaggedInteger(int n)
    {
        Number = n;
        tags = new List<string>();
    }

    public void AddTag(string tag) => tags.Add(tag);

    public override string ToString() => $"{Number} [{string.Join(", ",
tags)}]";
}

public class Program
{
    public static void Main()
    {
        var n1 = new TaggedInteger(0);
        n1.AddTag("A");
        Console.WriteLine(n1);  // output: 0 [A]

        var n2 = n1;
        n2.Number = 7;
        n2.AddTag("B");

        Console.WriteLine(n1);  // output: 0 [A, B]
        Console.WriteLine(n2);  // output: 7 [A, B]
    }
}
```

> ⓘ Notes
>
> Pour rendre votre code moins susceptible d'engendrer des erreurs et plus robuste, définissez et utilisez des types de valeurs immuables. Cet article utilise des types valeur mutables uniquement à des fins de démonstration.

Sortes de types valeur et contraintes de type

Un type valeur peut être de l'un des deux types suivants :

- un type de structure, qui encapsule les données et les fonctionnalités associées
- un type d'énumération, qui est défini par un ensemble de constantes nommées et représente un choix ou une combinaison de choix

Un type valeur pouvant accepter la valeur Null `T?` représente toutes les valeurs de son type valeur `T` sous-jacent et une valeur Null supplémentaire. Vous ne pouvez pas affecter `null` à une variable d'un type valeur, sauf s'il s'agit d'un type valeur pouvant accepter la valeur Null.

Vous pouvez utiliser la struct contrainte pour spécifier qu'un paramètre de type est un type valeur non pouvant accepter la valeur Null. Les types de structure et d'énumération satisfont à la contrainte `struct`. Vous pouvez utiliser `System.Enum` dans une contrainte de classe de base (appelée contrainte d'énumération) pour spécifier qu'un paramètre de type est un type d'énumération.

Types valeurs intégrés

C# fournit les types de valeurs intégrés suivants, également appelés *types simples* :

- Types numériques intégraux
- Types numériques à virgule flottante
- bool qui représente une valeur booléenne
- char qui représente un caractère Unicode UTF-16

Tous les types simples sont des types de structure et diffèrent des autres types de structure en ce qu'ils autorisent certaines opérations supplémentaires :

- Vous pouvez utiliser des littéraux pour fournir une valeur d'un type simple. Par exemple, `'A'` est un littéral de type `char`, tandis que `2001` est un littéral de type `int`.

- Vous pouvez déclarer des constantes des types simples avec le mot clé const. Il n'est pas possible d'avoir des constantes d'autres types de structure.
- Les expressions constantes, dont les opérandes sont tous des constantes de type simple, sont évaluées au moment de la compilation.

Un tuple de valeur est un type valeur, mais pas un type simple.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Types de valeur
- Types simples
- Variables

Voir aussi

- Informations de référence sur C#
- System.ValueType
- Types référence

Types numériques intégraux (référence C#)

Article • 10/05/2023

Les *types numériques intégraux* représentent des nombres entiers. Tous les types numériques intégraux sont des types valeur. Ils sont également des types simples et peuvent être initialisés avec des littéraux. Tous les types numériques intégraux prennent en charge les opérateurs arithmétiques, logiques au niveau du bit, de comparaison et d'égalité.

Caractéristiques des types intégraux

C# prend en charge les types intégraux prédéfinis suivants :

C# type/mot clé	Plage	Taille	Type .NET
`sbyte`	-128 à 127	Entier 8 bits signé	System.SByte
`byte`	0 à 255	Entier 8 bits non signé	System.Byte
`short`	-32 768 à 32 767	Entier 16 bits signé	System.Int16
`ushort`	0 à 65 535	Entier 16 bits non signé	System.UInt16
`int`	-2 147 483 648 à 2 147 483 647	Entier 32 bits signé	System.Int32
`uint`	de 0 à 4 294 967 295	Entier 32 bits non signé	System.UInt32
`long`	-9 223 372 036 854 775 808 à 9 223 372 036 854 775 807	Entier 64 bits signé	System.Int64
`ulong`	de 0 à 18 446 744 073 709 551 615	Entier 64 bits non signé	System.UInt64
`nint`	Dépend de la plateforme (calculée au moment du runtime)	Entier 32 bits ou 64 bits signé	System.IntPtr
`nuint`	Dépend de la plateforme (calculée au moment du runtime)	Entier 32 bits ou 64 bits non signé	System.UIntPtr

Dans toutes les lignes de table à l'exception des deux dernières, chaque mot clé de type C# de la colonne la plus à gauche est un alias pour le type .NET correspondant. Le mot clé et le nom de type .NET sont interchangeables. Par exemple, les déclarations suivantes déclarent des variables du même type :

```
C#

int a = 123;
System.Int32 b = 123;
```

Les types `nint` et `nuint` dans les deux dernières lignes de la table sont des entiers de taille native. À partir de C# 9.0, vous pouvez utiliser les mots clés `nint` et `nuint` pour définir des *entiers de taille native*. Il s'agit d'entiers 32 bits lors de l'exécution dans un processus 32 bits, ou d'entiers 64 bits lors de l'exécution dans un processus 64 bits. Utilisez-les pour les scénarios d'interopérabilité, les bibliothèques de bas niveau et pour optimiser les performances dans les scénarios où les mathématiques avec des entiers sont très utilisées.

Les types entiers de taille native sont représentés en interne en tant que types .NET System.IntPtr et System.UIntPtr. À partir de C# 11, les types `nint` et `nuint` sont des alias pour les types sous-jacents.

La valeur par défaut pour chaque type intégral est zéro, `0`.

Chacun des types intégraux possède des propriétés `MinValue` et `MaxValue` qui fournissent la valeur minimale et maximale de ce type. Ces propriétés sont des constantes au moment de la compilation, à l'exception du cas des types de taille native (`nint` et `nuint`). Les propriétés `MinValue` et `MaxValue` sont calculées au moment du runtime pour les types de taille native. Les tailles de ces types dépendent des paramètres de processus.

Utilisez la structure System.Numerics.BigInteger pour représenter un entier signé sans limites supérieures ou inférieures.

Littéraux d'entier

Les littéraux entiers peuvent être de type

- *décimal* : sans préfixe
- *hexadécimal* : avec le préfixe `0x` ou `0X`
- *binaire* : avec le préfixe `0b` ou `0B`

Le code suivant illustre un exemple de chaque :

```C#
var decimalLiteral = 42;
var hexLiteral = 0x2A;
var binaryLiteral = 0b_0010_1010;
```

L'exemple précédent montre également l'utilisation de `_` comme *séparateur de chiffres*. Vous pouvez utiliser le séparateur de chiffres avec tous types de littéraux numériques.

Le type d'un littéral entier est déterminé par son suffixe comme suit :

- Quand le littéral n'a pas de suffixe, son type est le premier des types suivants dans lesquels sa valeur peut être représentée : `int`, `uint`, `long`, `ulong`.

  > ⓘ **Notes**
  >
  > Les littéraux sont interprétés comme des valeurs positives. Par exemple, le littéral `0xFF_FF_FF_FF` représente le numéro `4294967295` du type `uint`, bien qu'il ait la même représentation de bits que le nombre `-1` du type `int`. Si vous avez besoin d'une valeur d'un certain type, castez un littéral en ce type. Utilisez l'opérateur `unchecked` si une valeur littérale ne peut pas être représentée dans le type cible. Par exemple, `unchecked((int)0xFF_FF_FF_FF)` produit `-1`.

- Quand le littéral possède un suffixe `U` ou `u`, son type est le premier des types suivants dans lesquels sa valeur peut être représentée : `uint`, `ulong`.

- Quand le littéral possède un suffixe `L` ou `l`, son type est le premier des types suivants dans lesquels sa valeur peut être représentée : `long`, `ulong`.

  > ⓘ **Notes**
  >
  > Vous pouvez utiliser la lettre minuscule `l` comme suffixe. Ceci génère cependant un avertissement du compilateur, car la lettre `l` peut être facilement confondue avec le chiffre `1`. Utilisez `L` pour plus de clarté.

- Si le littéral possède un suffixe `UL`, `Ul`, `uL`, `ul`, `LU`, `Lu`, `lU` ou `lu`, son type est `ulong`.

Si la valeur représentée par un littéral entier dépasse UInt64.MaxValue, une erreur de compilation CS1021 se produit.

Si le type déterminé d'un littéral entier est `int` et que la valeur représentée par le littéral se trouve dans la plage du type de destination, la valeur peut être convertie implicitement en `sbyte`, `byte`, `short` `ushort`, `uint`, `ulong`, `nint` ou `nuint` :

C#

```
byte a = 17;
byte b = 300;   // CS0031: Constant value '300' cannot be converted to a
'byte'
```

Comme l'illustre l'exemple précédent, si la valeur du littéral n'est pas dans la plage du type de destination, une erreur de compilateur CS0031 se produit.

Vous pouvez également utiliser un cast pour convertir la valeur représentée par un littéral entier en type autre que le type déterminé du littéral :

C#

```
var signedByte = (sbyte)42;
var longVariable = (long)42;
```

Conversions

Vous pouvez convertir n'importe quel type numérique intégral en tout autre type numérique intégral. Si le type de destination peut stocker toutes les valeurs du type source, la conversion est implicite. Sinon, vous devez utiliser une expression cast pour effectuer une conversion explicite. Pour plus d'informations, consultez Conversions numériques intégrées.

Entiers dimensionnés natifs

Les types d'entiers de taille native présentent un comportement spécial, car le stockage est déterminé par la taille d'entier naturelle sur l'ordinateur cible.

- Pour obtenir la taille d'un entier natif au moment de l'exécution, vous pouvez utiliser `sizeof()`. Toutefois, le code doit être compilé dans un contexte non sécurisé. Par exemple :

  C#

```
Console.WriteLine($"size of nint = {sizeof(nint)}");
Console.WriteLine($"size of nuint = {sizeof(nuint)}");

// output when run in a 64-bit process
//size of nint = 8
//size of nuint = 8

// output when run in a 32-bit process
//size of nint = 4
//size of nuint = 4
```

Vous pouvez également obtenir la valeur équivalente à partir des propriétés statiques IntPtr.Size et UIntPtr.Size.

- Pour obtenir les valeurs minimales et maximales des entiers de taille native au moment de l'exécution, utilisez `MinValue` et `MaxValue` en tant que propriétés statiques avec les mots clés `nint` et `nuint`, comme dans l'exemple suivant :

```
C#

Console.WriteLine($"nint.MinValue = {nint.MinValue}");
Console.WriteLine($"nint.MaxValue = {nint.MaxValue}");
Console.WriteLine($"nuint.MinValue = {nuint.MinValue}");
Console.WriteLine($"nuint.MaxValue = {nuint.MaxValue}");

// output when run in a 64-bit process
//nint.MinValue = -9223372036854775808
//nint.MaxValue = 9223372036854775807
//nuint.MinValue = 0
//nuint.MaxValue = 18446744073709551615

// output when run in a 32-bit process
//nint.MinValue = -2147483648
//nint.MaxValue = 2147483647
//nuint.MinValue = 0
//nuint.MaxValue = 4294967295
```

- Vous pouvez utiliser des valeurs constantes dans les plages suivantes :
  - Pour `nint` : Int32.MinValue à Int32.MaxValue.
  - Pour `nuint` : UInt32.MinValue à UInt32.MaxValue.

- Le compilateur fournit des conversions implicites et explicites vers d'autres types numériques. Pour plus d'informations, consultez Conversions numériques intégrées.

- Il n'existe aucune syntaxe directe pour les littéraux d'entiers de taille native. Il n'existe aucun suffixe pour indiquer qu'un littéral est un entier de taille native, par

exemple `L` pour indiquer un `long`. Vous pouvez utiliser des casts implicites ou explicites d'autres valeurs entières à la place. Par exemple :

```
C#
```

```
nint a = 42
nint a = (nint)42;
```

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Types intégraux
- Littéraux d'entier
- C# 9 – Types intégraux dimensionnés natifs
- C# 11 – IntPtr numérique et `UIntPtr

Voir aussi

- Informations de référence sur C#
- Types de valeur
- Types virgule flottante
- Chaînes de format numériques standard
- Valeurs numériques dans .NET

Types numériques à virgule flottante (Référence C#)

Article • 09/05/2023

Les *types numériques à virgule flottante* représentent des nombres réels. Les types numériques à virgule flottante sont des types de valeur. Ils sont également des types simples et peuvent être initialisés avec des littéraux. Tous les types numériques à virgule flottante prennent en charge les opérateurs arithmétiques et les opérateurs de comparaison et d'égalité.

Caractéristiques des types à virgule flottante

C# prend en charge les types à virgule flottante prédéfinis suivants :

C# type/mot clé	Plage approximative	Precision	Taille	Type .NET
`float`	±1,5 x 10^{-45} à ±3,4 x 10^{38}	~6-9 chiffres	4 octets	System.Single
`double`	De ±5,0 × 10^{-324} à ±1,7 × 10^{308}	~15-17 chiffres	8 octets	System.Double
`decimal`	±1,0 x 10^{-28} to ±7,9228 x 10^{28}	28 à 29 chiffres	16 octets	System.Decimal

Dans le tableau précédent, chaque mot clé de type C# de la colonne la plus à gauche est un alias pour le type .NET correspondant. Ils sont interchangeables. Par exemple, les déclarations suivantes déclarent des variables du même type :

```
C#

double a = 12.3;
System.Double b = 12.3;
```

La valeur par défaut pour tous les types virgule flottante est zéro, `0`. Chacun des types à virgule flottante a les constantes `MinValue` et `MaxValue` qui fournissent la valeur finie minimale et maximale de ce type. Les types `float` et `double` fournissent également des constantes qui représentent des valeurs NaN (Not-a-Number) et d'infini. Par exemple, le type `double` fournit les constantes suivantes : Double.NaN, Double.NegativeInfinity et Double.PositiveInfinity.

Le type `decimal` est approprié lorsque le degré de précision requis est déterminé par le nombre de chiffres à droite du point décimal. Ces nombres sont couramment utilisés dans les applications financières, pour les montants monétaires (par exemple, 1,00 $), les taux d'intérêt (par exemple, 2,625 %) et ainsi de suite. Même les nombres qui sont précis à un seul chiffre décimal sont gérés avec plus de précision par le type `decimal` : 0,1, par exemple, peut être représenté exactement par une instance `decimal`, alors qu'il n'y a aucune instance `double` ou `float` qui représente exactement 0,1. En raison de cette différence dans les types numériques, des erreurs d'arrondi inattendues peuvent se produire dans les calculs arithmétiques lorsque vous utilisez `double` ou `float` pour des données décimales. Vous pouvez utiliser `double` au lieu de `decimal` lorsque l'optimisation des performances est plus importante que la garantie de la précision. Toutefois, toute différence de performance passerait inaperçue par toutes les applications, sauf les applications les plus gourmandes en calculs. Une autre raison possible d'éviter `decimal` est de réduire les besoins en stockage. Par exemple, ML.NET utilise `float`, car la différence entre 4 et 16 octets s'additionne pour les jeux de données très volumineux. Pour plus d'informations, consultez System.Decimal.

Vous pouvez combiner des types intégraux et des types `float` et `double` dans une expression. Dans ce cas, les types intégraux sont convertis implicitement en un des types à virgule flottante et, si nécessaire, le type `float` est implicitement converti en `double`. L'expression est évaluée de la manière suivante :

- S'il y a un type `double` dans l'expression, celle-ci prend la valeur `double` ou bool dans les comparaisons relationnelles ou les comparaisons d'égalité.
- S'il n'y a aucun type `double` dans l'expression, celle-ci prend la valeur `float` ou `bool` dans les comparaisons relationnelles ou les comparaisons d'égalité.

Vous pouvez combiner des types intégraux et le type `decimal` dans une expression. Dans ce cas, les types intégraux sont implicitement convertis en type `decimal` et l'expression est évaluée en `decimal`, ou en `bool` dans les comparaisons relationnelles et d'égalité.

Vous ne pouvez pas mélanger le type `decimal` avec les types `float` et `double` dans une expression. Dans ce cas, si vous souhaitez effectuer des opérations arithmétiques, de comparaison ou d'égalité, vous devez explicitement convertir les opérandes à partir du type `decimal`, comme le montre l'exemple suivant :

C#

```
double a = 1.0;
decimal b = 2.1m;
```

```
Console.WriteLine(a + (double)b);
Console.WriteLine((decimal)a + b);
```

Vous pouvez utiliser des chaînes au format numérique standard ou des chaînes au format numérique personnalisées pour mettre en forme une valeur à virgule flottante.

Littéraux réels

Le type d'un littéral réel est déterminé par son suffixe comme suit :

- Le littéral sans suffixe ou avec le suffixe `d` ou `D` est de type `double`
- Le littéral avec le suffixe `f` ou `F` est de type `float`
- Le littéral avec le suffixe `m` ou `M` est de type `decimal`

Le code suivant illustre un exemple de chaque :

C#

```
double d = 3D;
d = 4d;
d = 3.934_001;

float f = 3_000.5F;
f = 5.4f;

decimal myMoney = 3_000.5m;
myMoney = 400.75M;
```

L'exemple précédent montre également l'utilisation de `_` comme *séparateur de chiffres*. Vous pouvez utiliser le séparateur de chiffres avec tous types de littéraux numériques.

Vous pouvez également utiliser la notation scientifique, c'est-à-dire, spécifier une partie exposante d'un littéral réel, comme le montre l'exemple suivant :

C#

```
double d = 0.42e2;
Console.WriteLine(d);  // output 42

float f = 134.45E-2f;
Console.WriteLine(f);  // output: 1.3445

decimal m = 1.5E6m;
Console.WriteLine(m);  // output: 1500000
```

Conversions

Il n'existe qu'une seule conversion implicite entre les types numériques à virgule flottante : de `float` à `double`. Toutefois, vous pouvez convertir n'importe quel type à virgule flottante en n'importe quel autre type à virgule flottante avec le cast explicite. Pour plus d'informations, consultez Conversions numériques intégrées.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Types virgule flottante
- Type décimal
- Littéraux réels

Voir aussi

- Informations de référence sur C#
- Types de valeur
- Types intégraux
- Chaînes de format numériques standard
- Valeurs numériques dans .NET
- System.Numerics.Complex

Conversions numériques intégrées (référence C#)

Article • 07/04/2023

C# fournit un ensemble de types numériques intégraux et à virgule flottante. Il existe une conversion entre deux types numériques, implicites ou explicites. Vous devez utiliser une expression cast pour effectuer une conversion explicite.

Conversions numériques implicites

Le tableau suivant présente les conversions implicites prédéfinies entre les types numériques intégrés :

Du	À
sbyte	`short`, `int`, `long`, `float`, `double`, `decimal` ou `nint`
byte	`short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, `decimal`, `nint` ou `nuint`
short	`int`, `long`, `float`, `double`, ou `decimal`, ou `nint`
ushort	`int`, `uint`, `long`, `ulong`, `float`, `double`, ou `decimal`, `nint`, ou `nuint`
int	`long`, `float`, `double`, ou `decimal`, `nint`
uint	`long`, `ulong`, `float`, `double`, ou `decimal`, ou `nuint`
long	`float`, `double` ou `decimal`
ulong	`float`, `double` ou `decimal`
float	`double`
nint	`long`, `float`, `double` ou `decimal`
nuint	`ulong`, `float`, `double` ou `decimal`

> **Notes**
>
> Les conversions implicites de `int`, `uint`, `long`, `ulong`, `nint` ou `nuint` à `float` et de `long`, `ulong`, `nint` ou `nuint` à `double` peuvent provoquer une perte de précision, mais jamais une perte d'ordre de grandeur. Les autres conversions numériques implicites ne perdent jamais d'informations.

Notez également que

- Tout type numérique intégral est implicitement convertible en n'importe quel type numérique à virgule flottante.
- Il n'existe aucune conversion implicite vers les types `byte` et `sbyte`. Il n'existe aucune conversion implicite à partir des types `double` et `decimal`.
- Il n'existe aucune conversion implicite entre le type `decimal` et le type `float` ou `double`.
- La valeur d'une expression constante de type `int` (par exemple, une valeur représentée par un littéral intégral) peut être convertie en `sbyte`, `byte`, `short`, `ushort`, `uint`, `ulong`, `nint` ou `nuint`, à condition qu'elle se trouve dans la plage du type de destination :

  C#

  ```
  byte a = 13;
  byte b = 300;  // CS0031: Constant value '300' cannot be converted to a
  'byte'
  ```

  Comme l'illustre l'exemple précédent, si la valeur constante n'est pas dans la plage du type de destination, une erreur de compilateur CS0031 se produit.

Conversions numériques explicites

Le tableau suivant indique les conversions explicites prédéfinies entre les types numériques intégrés pour lesquels il n'existe aucune conversion implicite :

Du	À
sbyte	`byte`, `ushort`, `uint`, `ulong` ou `nuint`
byte	`sbyte`
short	`sbyte`, `byte`, `ushort`, `uint`, `ulong` ou `nuint`
ushort	`sbyte`, `byte` ou `short`
int	`sbyte`, `byte`, `short`, `ushort`, `uint`, `ulong` ou `nuint`
uint	`sbyte`, `byte`, `short`, `ushort`, `int` ou `nint`
long	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `ulong`, `nint` ou `nuint`

Du	À
ulong	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `nint` ou `nuint`
float	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `decimal`, `nint` ou `nuint`
double	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `decimal`, `nint` ou `nuint`
decimal	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, `nint` ou `nuint`
nint	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `ulong` ou `nuint`
nuint	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long` ou `nint`

> ⓘ **Notes**
>
> Une conversion numérique explicite peut entraîner une perte de données ou lever une exception, généralement **OverflowException**.

Notez également que :

- Quand vous convertissez une valeur d'un type intégral en un autre type intégral, le résultat varie selon le contexte de vérification du dépassement. Dans un contexte vérifié, la conversion réussit si la valeur source se situe dans la plage du type de destination. Sinon, une exception OverflowException est levée. Dans un contexte non vérifié, la conversion réussit toujours et procède comme suit :
  - Si le type source est plus grand que le type de destination, la valeur source est tronquée via l'abandon de ses bits « supplémentaires » les plus significatifs. Le résultat est ensuite traité en tant que valeur du type de destination.
  - Si le type source est plus petit que le type de destination, la valeur source fait l'objet d'une extension de signe ou d'une extension de zéros, pour avoir la même taille que le type de destination. L'extension de signe est utilisée si le type source est signé. L'extension de zéro est utilisée si le type source est non signé. Le résultat est ensuite traité en tant que valeur du type de destination.
  - Si le type source a la même taille que le type de destination, la valeur source est traitée comme une valeur du type de destination.
- Quand vous convertissez une valeur de type `decimal` en type intégral, la valeur source est arrondie vers zéro à la valeur intégrale la plus proche. Si la valeur intégrale obtenue se trouve en dehors de la plage du type de destination, une exception OverflowException est levée.

- Quand vous convertissez une valeur `double` ou `float` en type intégral, cette valeur est arrondie vers zéro, à la valeur intégrale la plus proche. Si la valeur intégrale résultante se situe en dehors de la plage du type de destination, le résultat varie selon le contexte de vérification du dépassement. Dans un contexte vérifié, une exception OverflowException est levée, alors que dans un contexte non vérifié, le résultat est une valeur non spécifiée du type de destination.

- Quand vous convertissez `double` en `float`, la valeur `double` est arrondie à la valeur `float` la plus proche. Si la valeur `double` est inférieure ou supérieure au type `float`, la conversion a pour résultat une valeur nulle ou un nombre infini.

- Dans le cas d'une conversion de `float` ou `double` en `decimal`, la valeur source est convertie en une représentation `decimal` et arrondie au nombre le plus proche après la 28ème décimale, si nécessaire. Selon la valeur de la valeur source, la conversion peut aboutir à l'un des résultats suivants :

  - Si la valeur source est trop petite pour être représentée en tant que `decimal`, le résultat est zéro.

  - Si la valeur source est une valeur non numérique (NaN), un nombre infini ou une valeur trop grande pour être représentée au format `decimal`, une exception OverflowException est levée.

- Quand vous convertissez `decimal` en `float` ou `double`, la valeur source est arrondie à la valeur `float` ou `double` la plus proche, respectivement.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Conversions numériques implicites
- Conversions numériques explicites

Voir aussi

- Informations de référence sur C#
- Cast et conversions de types

bool (référence C#)

Article • 10/05/2023

Le mot clé de type `bool` est un alias pour le type de structure .NET System.Boolean qui représente une valeur booléenne, qui peut être `true` ou `false`.

Pour effectuer des opérations logiques avec des valeurs du type `bool`, utilisez des opérateurs logiques booléens. Le type `bool` est le type de résultat des opérateurs de comparaison et d'égalité. Une expression `bool` défini peut être une expression conditionnelle de contrôle dans les instructions if, do, while et for et dans l'opérateur conditionnel ?:.

La valeur par défaut du type `bool` est `false`.

Littéraux

Vous pouvez utiliser les littéraux `true` et `false` pour initialiser une variable `bool` ou transmettre une valeur `bool` :

C#

```
bool check = true;
Console.WriteLine(check ? "Checked" : "Not checked");  // output: Checked

Console.WriteLine(false ? "Checked" : "Not checked");  // output: Not
checked
```

Logique booléenne à trois valeurs

Utilisez le type Nullable `bool?` si vous voulez suivre une logique à trois valeurs, par exemple pour travailler avec des bases de données qui acceptent un type booléen à trois valeurs. Dans le cas des opérandes `bool?`, les opérateurs prédéfinis `&` et `|` prennent en charge la logique à trois valeurs. Pour plus d'informations, voir la section Opérateurs logiques booléens Nullable de l'article Opérateurs logiques booléens.

Pour plus d'informations sur les types de valeur pouvant accepter la valeur Null, consultez Types de valeur pouvant accepter la valeur Null.

Conversions

C# fournit uniquement deux conversions qui impliquent le type `bool`. Il s'agit d'une conversion implicite vers le type Nullable `bool?` correspondant et d'une conversion explicite à partir du type `bool?`. Toutefois, .NET fournit des méthodes supplémentaires que vous pouvez utiliser pour effectuer une conversion vers ou depuis le type `bool`. Pour plus d'informations, consultez la section Convertir en et à partir de valeurs booléennes de la page de référence de l'API System.Boolean.

spécification du langage C#

Pour plus d'informations, consultez la section Type bool de la spécification du langage C#.

Voir aussi

- Informations de référence sur C#
- Types de valeur
- opérateurs true et false

char (référence C#)

Article • 09/05/2023

Le mot clé de type `char` est un alias du type de structure .NET System.Char qui représente un caractère Unicode UTF-16.

Type	Plage	Taille	Type .NET
`char`	U+0000 à U+FFFF	16 bits	System.Char

La valeur par défaut du type `char` est `\0`, autrement dit, U+0000.

Le type `char` prend en charge les opérateurs de comparaison, d'égalité, d'incrémentationet de décrémentation. En outre, pour les opérandes `char`, les opérateurs arithmétiques et logiques au niveau du bit effectuent une opération sur les codes des caractères correspondants et produisent le résultat du type `int`.

Le type string représente le texte sous forme de séquence de valeurs `char`.

Littéraux

Vous pouvez spécifier une valeur `char` avec :

- un littéral de caractère.
- une séquence d'échappement Unicode, qui est `\u` suivi de la représentation hexadécimale à quatre symboles d'un code de caractère.
- une séquence d'échappement hexadécimale, qui est `\x` suivi de la représentation hexadécimale d'un code de caractère.

C#

```
var chars = new[]
{
    'j',
    '\u006A',
    '\x006A',
    (char)106,
};
Console.WriteLine(string.Join(" ", chars));  // output: j j j j
```

Comme l'illustre l'exemple précédent, vous pouvez également convertir la valeur d'un code de caractère en valeur `char` correspondante.

> **Notes**
>
> Dans le cas d'une séquence d'échappement Unicode, vous devez spécifier les quatre chiffres hexadécimaux. Autrement dit, `\u006A` est une séquence d'échappement valide, tandis que `\u06A` et `\u6A` ne le sont pas.
>
> Dans le cas d'une séquence d'échappement hexadécimale, vous pouvez omettre les zéros non significatifs. Autrement dit, les séquences d'échappement `\x006A`, `\x06A` et `\x6A` sont valides et correspondent au même caractère.

Conversions

Le type `char` est implicitement convertible en types intégraux suivants : `ushort`, `int`, `uint`, `long` et `ulong`. Il est également implicitement convertible en types numériques à virgule flottante intégrés : `float`, `double` et `decimal`. Il est explicitement convertible en types intégraux `sbyte`, `byte` et `short`.

Vous ne pouvez pas convertir implicitement d'autres types en type `char`. En revanche, tout type numérique intégral ou à virgule flottante est explicitement convertible en `char`.

spécification du langage C#

Pour plus d'informations, consultez la section Types intégraux de la Spécification du langage C#.

Voir aussi

- Informations de référence sur C#
- Types de valeur
- Chaînes
- System.Text.Rune
- Encodage de caractères dans .NET

Types d'énumération (référence C#)

Article • 07/04/2023

Un *type d'énumération* (ou *type enum*) est un type valeur défini par un ensemble de constantes nommées du type numérique intégral sous-jacent. Pour définir un type d'énumération, utilisez le `enum` mot clé et spécifiez les noms des *membres enum* :

C#

```
enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}
```

Par défaut, les valeurs constantes associées des membres enum sont de type `int` ; elles commencent par zéro et augmentent de un en suivant l'ordre textuel de la définition. Vous pouvez spécifier explicitement n'importe quel autre type numérique intégral comme type sous-jacent d'un type d'énumération. Vous pouvez également spécifier explicitement les valeurs constantes associées, comme le montre l'exemple suivant :

C#

```
enum ErrorCode : ushort
{
    None = 0,
    Unknown = 1,
    ConnectionLost = 100,
    OutlierReading = 200
}
```

Vous ne pouvez pas définir une méthode à l'intérieur de la définition d'un type d'énumération. Pour ajouter des fonctionnalités à un type d'énumération, créez une méthode d'extension.

La valeur par défaut d'un type d'énumération `E` est la valeur produite par l'expression `(E)0`, même si zéro n'a pas le membre enum correspondant.

Vous utilisez un type d'énumération pour représenter un choix à partir d'un ensemble de valeurs mutuellement exclusives ou d'une combinaison de choix. Pour représenter une combinaison de choix, définissez un type d'énumération en tant qu'indicateurs binaires.

Types énumération comme indicateurs binaires

Si vous souhaitez qu'un type d'énumération représente une combinaison de choix, définissez des membres enum pour ces choix, de sorte qu'un choix individuel soit un champ de bits. Autrement dit, les valeurs associées de ces membres enum doivent être les puissances de deux. Ensuite, vous pouvez utiliser les opérateurs logiques binaires| ou & pour combiner des choix ou croiser des combinaisons de choix, respectivement. Pour indiquer qu'un type d'énumération déclare des champs binaires, appliquez-lui l'attribut Indicateurs . Comme le montre l'exemple suivant, vous pouvez également inclure certaines combinaisons classiques dans la définition d'un type d'énumération.

C#

```
[Flags]
public enum Days
{
    None      = 0b_0000_0000,  // 0
    Monday    = 0b_0000_0001,  // 1
    Tuesday   = 0b_0000_0010,  // 2
    Wednesday = 0b_0000_0100,  // 4
    Thursday  = 0b_0000_1000,  // 8
    Friday    = 0b_0001_0000,  // 16
    Saturday  = 0b_0010_0000,  // 32
    Sunday    = 0b_0100_0000,  // 64
    Weekend   = Saturday | Sunday
}

public class FlagsEnumExample
{
    public static void Main()
    {
        Days meetingDays = Days.Monday | Days.Wednesday | Days.Friday;
        Console.WriteLine(meetingDays);
        // Output:
        // Monday, Wednesday, Friday

        Days workingFromHomeDays = Days.Thursday | Days.Friday;
        Console.WriteLine($"Join a meeting by phone on {meetingDays &
workingFromHomeDays}");
        // Output:
        // Join a meeting by phone on Friday

        bool isMeetingOnTuesday = (meetingDays & Days.Tuesday) ==
Days.Tuesday;
        Console.WriteLine($"Is there a meeting on Tuesday:
{isMeetingOnTuesday}");
        // Output:
        // Is there a meeting on Tuesday: False

        var a = (Days)37;
        Console.WriteLine(a);
```

```
        // Output:
        // Monday, Wednesday, Saturday
    }
}
```

Pour plus d'informations et d'exemples, consultez la System.FlagsAttribute page de référence d'API et les membres non exclusifs et la section Attributs indicateurs de la page de référence d'API System.Enum .

Le système. Type enum et contrainte d'énumération

Le type System.Enum est la classe de base abstraite de tous les types d'énumération. Il fournit plusieurs méthodes pour obtenir des informations sur un type d'énumération et ses valeurs. Pour plus d'informations et des exemples, consultez la page de référence API System.Enum.

Vous pouvez utiliser `System.Enum` dans une contrainte de classe de base (appelée contrainte d'énumération) pour spécifier qu'un paramètre de type est un type d'énumération. Tout type d'énumération respecte également la contrainte `struct`, qui est utilisée pour spécifier qu'un paramètre de type est un type valeur non-nullable.

Conversions

Pour tout type d'énumération, il existe des conversions explicites entre le type d'énumération et son type intégral sous-jacent. Si vous castez une valeur enum vers son type sous-jacent, le résultat est la valeur intégrale associée d'un membre enum.

C#

```
public enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}

public class EnumConversionExample
{
    public static void Main()
    {
        Season a = Season.Autumn;
        Console.WriteLine($"Integral value of {a} is {(int)a}");  // output:
Integral value of Autumn is 2
```

```
        var b = (Season)1;
        Console.WriteLine(b);  // output: Summer

        var c = (Season)4;
        Console.WriteLine(c);  // output: 4
    }
}
```

Utilisez la méthode Enum.IsDefined pour déterminer si un type d'énumération contient un membre enum avec la valeur associée certaine.

Pour tout type d'énumération, il existe respectivement, des conversions de boxing et d'unboxing vers et à partir du typeSystem.Enum.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Énumérations
- Valeurs et opérations des énumérations
- Opérateurs logiques d'énumération
- Opérateurs de comparaison d'énumération
- Conversions d'énumération explicites
- Conversions d'énumération implicites

Voir aussi

- Informations de référence sur C#
- Chaînes de format d'énumération
- Lignes directrices de conception - Conception d'énumération
- Lignes directrices de conception - Conventions d'affectation de noms enum
- switch, expression
- instruction switch

Types de structure (référence C#)

Article • 13/07/2023

Un *type de structure* (ou *type de struct*) est un type valeur qui peut encapsuler des données et des fonctionnalités associées. Vous utilisez le mot clé `struct` pour définir un type de structure :

C#

```
public struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; }
    public double Y { get; }

    public override string ToString() => $"({X}, {Y})";
}
```

Pour plus d'informations sur les types `ref struct` et `readonly ref struct`, consultez l'article sur les types de structure ref.

Les types de structure ont une *sémantique de valeur*. Autrement dit, une variable d'un type de structure contient une instance du type. Par défaut, les valeurs de variable sont copiées lors de l'affectation, en transformant un argument en une méthode et en retournant un résultat de méthode. Pour les variables de type structure, une instance du type est copiée. Pour plus d'informations, consultez Types valeur.

En règle générale, vous utilisez des types de structure pour concevoir de petits types centrés sur les données qui fournissent peu ou pas de comportement. Par exemple, .NET utilise des types de structure pour représenter un nombre (entier et réel), une valeur booléenne, un caractère Unicode, une instance temporelle. Si vous vous concentrez sur le comportement d'un type, envisagez de définir une classe. Les types de classe ont une *sémantique de référence*. Autrement dit, une variable d'un type de classe contient une référence à une instance du type, et non à l'instance elle-même.

Étant donné que les types de structure ont une sémantique de valeur, nous vous recommandons de définir des types de structure *immuables*.

`readonly` struct

Vous utilisez le modificateur `readonly` pour déclarer qu'un type de structure est immuable. Tous les membres de données d'une struct `readonly` doivent être en lecture seule comme suit :

- Toute déclaration de champ doit avoir le modificateur readonly
- Toute propriété, y compris les propriétés implémentées automatiquement, doit être en lecture seule. Dans C# 9.0 et les versions ultérieures, une propriété peut avoir un accesseur init.

Cela garantit qu'aucun membre d'une struct `readonly` modifie l'état de la struct. Cela signifie que d'autres membres d'instance, à l'exception des constructeurs, sont implicitement readonly.

> ⓘ **Notes**
>
> Dans une struct `readonly`, un membre de données d'un type de référence mutable peut toujours muter son propre état. Par exemple, vous ne pouvez pas remplacer une instance `List<T>`, mais vous pouvez y ajouter de nouveaux éléments.

Le code suivant définit une struct `readonly` avec des setters de propriétés init uniquement, disponibles en C# 9.0 et versions ultérieures :

C#

```
public readonly struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; init; }
    public double Y { get; init; }

    public override string ToString() => $"({X}, {Y})";
}
```

Membres d'instance `readonly`

Vous pouvez également utiliser le modificateur `readonly` pour déclarer qu'un membre d'instance ne modifie pas l'état d'une struct. Si vous ne pouvez pas déclarer le type de structure entier en tant que `readonly`, utilisez le `readonly` modificateur pour marquer les membres d'instance qui ne modifient pas l'état de la struct.

Dans un membre d'instance `readonly`, vous ne pouvez pas affecter les champs d'instance de la structure. Toutefois, un membre `readonly` peut appeler un membre non `readonly`. Dans ce cas, le compilateur crée une copie de l'instance de structure et appelle le membre non `readonly` sur cette copie. Par conséquent, l'instance de structure d'origine n'est pas modifiée.

En règle générale, vous appliquez le modificateur `readonly` aux types de membres d'instance suivants :

- méthodes :

  C#

  ```
  public readonly double Sum()
  {
      return X + Y;
  }
  ```

  Vous pouvez également appliquer le modificateur `readonly` aux méthodes qui remplacent les méthodes déclarées dans System.Object :

  C#

  ```
  public readonly override string ToString() => $"({X}, {Y})";
  ```

- propriétés et indexeurs :

  C#

  ```
  private int counter;
  public int Counter
  {
      readonly get => counter;
      set => counter = value;
  }
  ```

  Si vous devez appliquer le modificateur `readonly` aux deux accesseurs d'une propriété ou d'un indexeur, appliquez-le dans la déclaration de la propriété ou de l'indexeur.

> ⓘ **Notes**
>
> Le compilateur déclare un accesseur `get` d'une **propriété implémentée automatiquement** comme `readonly`, quel que soit la présence du modificateur `readonly` dans une déclaration de propriété.

Dans C# 9.0 et versions ultérieures, vous pouvez appliquer le modificateur `readonly` à une propriété ou un indexeur avec un accesseur `init` :

```
C#

public readonly double X { get; init; }
```

Vous pouvez appliquer le modificateur `readonly` aux champs statiques d'un type de structure, mais pas à d'autres membres statiques, tels que des propriétés ou des méthodes.

Le compilateur peut utiliser le modificateur `readonly` pour les optimisations des performances. Pour plus d'informations, consultez Éviter les allocations.

Mutation non destructrice

À compter de C# 10, vous pouvez utiliser l'expression with pour produire une copie d'une instance de type structure avec les propriétés et champs spécifiés modifiés. Vous utilisez la syntaxe d'initialiseur d'objet pour spécifier les membres à modifier et leurs nouvelles valeurs, comme l'illustre l'exemple suivant :

```
C#

public readonly struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; init; }
    public double Y { get; init; }

    public override string ToString() => $"({X}, {Y})";
}

public static void Main()
```

```
{
    var p1 = new Coords(0, 0);
    Console.WriteLine(p1);  // output: (0, 0)

    var p2 = p1 with { X = 3 };
    Console.WriteLine(p2);  // output: (3, 0)

    var p3 = p1 with { X = 1, Y = 4 };
    Console.WriteLine(p3);  // output: (1, 4)
}
```

struct `record`

À compter de C# 10, vous pouvez définir des types de structure d'enregistrement. Les types d'enregistrements fournissent des fonctionnalités intégrées pour encapsuler des données. Vous pouvez définir les deux types `record struct` et `readonly record struct`. Une struct d'enregistrement ne peut pas être un ref struct. Pour plus d'informations et d'exemples, consultez Enregistrements.

Tableaux inlined

À compter de C# 12, vous pouvez déclarer *des tableaux inlined* en tant `struct` que type :

C#

```
[System.Runtime.CompilerServices.InlineArray(10)]
public struct CharBuffer
{
    private char _firstElement;
}
```

Un tableau inlined est une structure qui contient un bloc contigu d'éléments N du même type. Il s'agit d'un équivalent en code sécurisé de la déclaration de mémoire tampon fixe disponible uniquement dans le code non sécurisé. Un tableau inlined est un `struct` avec les caractéristiques suivantes :

- Il contient un seul champ.
- Le struct ne spécifie pas de disposition explicite.

En outre, le compilateur valide System.Runtime.CompilerServices.InlineArrayAttributel'attribut :

- La longueur doit être supérieure à zéro (`> 0`).

- Le type de cible doit être un struct.

Dans la plupart des cas, on peut accéder à un tableau en ligne comme à un tableau, à la fois pour lire et écrire des valeurs. En outre, vous pouvez utiliser les opérateurs de plage et d'index.

Les restrictions concernant le type de champ unique sont minimes. Il ne peut s'agir d'un type de pointeur, mais il peut s'agir de n'importe quel type de référence ou de n'importe quel type de valeur. Vous pouvez utiliser des tableaux inlined avec presque n'importe quelle structure de données C#.

Les tableaux inlined sont une fonctionnalité de langage avancée. Ils sont destinés aux scénarios hautes performances où un bloc d'éléments inlined et contigu est plus rapide que d'autres structures de données alternatives. Vous pouvez en savoir plus sur les tableaux inlined à partir du speclet de fonctionnalité

Initialisation de struct et valeurs par défaut

Une variable d'un type `struct` contient directement les données pour ce `struct`. Cela crée une distinction entre une `struct` non initialisée qui a sa valeur par défaut, et une `struct` initialisée qui stocke les valeurs définies en la construisant. Par exemple, prenons le code suivant :

C#

```
public readonly struct Measurement
{
    public Measurement()
    {
        Value = double.NaN;
        Description = "Undefined";
    }

    public Measurement(double value, string description)
    {
        Value = value;
        Description = description;
    }

    public double Value { get; init; }
    public string Description { get; init; }

    public override string ToString() => $"{Value} ({Description})";
}

public static void Main()
{
```

```
    var m1 = new Measurement();
    Console.WriteLine(m1);  // output: NaN (Undefined)

    var m2 = default(Measurement);
    Console.WriteLine(m2);  // output: 0 ()

    var ms = new Measurement[2];
    Console.WriteLine(string.Join(", ", ms));  // output: 0 (), 0 ()
}
```

Comme l'illustre l'exemple précédent, l'expression de valeur par défaut ignore un constructeur sans paramètre et produit la valeur par défaut du type de structure. L'instanciation de tableau de type structure ignore également un constructeur sans paramètre et produit un tableau rempli avec les valeurs par défaut d'un type de structure.

La situation la plus courante où vous verrez que les valeurs par défaut se trouvent dans des tableaux ou dans d'autres collections où le stockage interne inclut des blocs de variables. L'exemple suivant crée un tableau de 30 structures `TemperatureRange`, chacune ayant la valeur par défaut :

C#

```
// All elements have default values of 0:
TemperatureRange[] lastMonth = new TemperatureRange[30];
```

Tous les champs membres d'une struct doivent être *définitivement attribués* lors de sa création, car les types `struct` stockent directement leurs données. La valeur `default` d'un struct a *définitivement défini* tous les champs sur 0. Tous les champs doivent être affectés définitivement lorsqu'un constructeur est appelé. Vous initialisez des champs à l'aide des mécanismes suivants :

- Vous pouvez ajouter des *initialiseurs de champ* à n'importe quel champ ou propriété implémentée automatiquement.
- Vous pouvez initialiser tous les champs ou propriétés automatiques dans le corps du constructeur.

À compter de C# 11, si vous n'initialisez pas tous les champs dans une struct, le compilateur ajoute du code au constructeur qui initialise ces champs à la valeur par défaut. Le compilateur effectue son analyse habituelle de l'affectation définie. Tous les champs accessibles avant d'être attribués, ou qui ne sont pas affectés définitivement lorsque le constructeur termine l'exécution sont affectés à leurs valeurs par défaut avant l'exécution du corps du constructeur. Si `this` est accessible avant que tous les champs

ne soient attribués, la struct est initialisé à la valeur par défaut avant l'exécution du corps du constructeur.

C#

```
public readonly struct Measurement
{
    public Measurement(double value)
    {
        Value = value;
    }

    public Measurement(double value, string description)
    {
        Value = value;
        Description = description;
    }

    public Measurement(string description)
    {
        Description = description;
    }

    public double Value { get; init; }
    public string Description { get; init; } = "Ordinary measurement";

    public override string ToString() => $"{Value} ({Description})";
}

public static void Main()
{
    var m1 = new Measurement(5);
    Console.WriteLine(m1);  // output: 5 (Ordinary measurement)

    var m2 = new Measurement();
    Console.WriteLine(m2);  // output: 0 ()

    var m3 = default(Measurement);
    Console.WriteLine(m3);  // output: 0 ()
}
```

Chaque `struct` a un constructeur `public` sans paramètre. Si vous écrivez un constructeur sans paramètre, il doit être public. Si une struct déclare des initialiseurs de champ, elle doit déclarer explicitement un constructeur. Ce constructeur n'a pas besoin d'être sans paramètre. Si une struct déclare un initialiseur de champ mais aucun constructeur, le compilateur signale une erreur. Tout constructeur explicitement déclaré (avec des paramètres ou sans paramètre) exécute tous les initialiseurs de champ pour cette struct. Tous les champs sans initialiseur de champ ou affectation dans un constructeur sont définis sur la valeur par défaut. Pour plus d'informations, consultez la proposition de caractéristique Constructeurs de struct sans paramètre.

À partir de C# 12, les types `struct` peuvent définir un constructeur principal dans le cadre de leur déclaration. Les constructeurs primaires fournissent une syntaxe concise pour les paramètres du constructeur qui peuvent être utilisés dans tout le corps, `struct`, dans n'importe quelle déclaration de membre pour cette struct.

Si tous les champs d'instance d'un type de structure sont accessibles, vous pouvez également l'instancier sans l'opérateur `new`. Dans ce cas, vous devez initialiser tous les champs d'instance avant la première utilisation de l'instance. L'exemple suivant montre comment effectuer cette opération :

C#

```
public static class StructWithoutNew
{
    public struct Coords
    {
        public double x;
        public double y;
    }

    public static void Main()
    {
        Coords p;
        p.x = 3;
        p.y = 4;
        Console.WriteLine($"({p.x}, {p.y})");  // output: (3, 4)
    }
}
```

Dans le cas des types de valeur intégrés, utilisez les littéraux correspondants pour spécifier une valeur du type.

Limitations avec la conception d'un type de structure

Les structs ont la plupart des fonctionnalités d'un type de classe. Il existe certaines exceptions, et notamment des exceptions qui ont été supprimées dans des versions plus récentes :

- Un type de structure ne peut pas hériter d'une autre classe ou d'un autre type de structure, et il ne peut pas être la base d'une classe. Toutefois, un type de structure peut implémenter des interfaces.
- Vous ne pouvez pas déclarer de finaliseur au sein d'un type de structure.

- Avant C# 11, un constructeur d'un type de structure doit initialiser tous les champs d'instance du type.
- Avant C# 10, vous ne pouvez pas déclarer de constructeur sans paramètre.
- Avant C# 10, vous ne pouvez pas initialiser un champ d'instance ou une propriété au moment de sa déclaration.

Passage de variables de type structure par référence

Lorsque vous transformez une variable de type structure en une méthode en tant qu'argument ou que vous renvoyez une valeur de type structure à partir d'une méthode, l'instance entière d'un type de structure est copiée. La transmission par valeur peut affecter les performances de votre code dans des scénarios hautes performances qui impliquent des types de structure volumineux. Vous pouvez éviter la copie de valeurs en transformant une variable de type structure par référence. Utilisez les modificateurs de paramètres de méthode ref, out ou in pour indiquer qu'un argument doit être transformé par référence. Utilisez les retours par référence pour retourner un résultat de méthode par référence. Pour plus d'informations, consultez Éviter les allocations.

contrainte de struct

Vous utilisez également le mot clé `struct` dans la contrainte struct pour spécifier qu'un paramètre de type est un type valeur non nullable. Les types de structure et d'énumération répondent à la contrainte `struct`.

Conversions

Pour n'importe quel type de structure (à l'exception des types ref struct), il existe des conversions boxing et unboxing vers et depuis les types System.ValueType et System.Object. Il existe également des conversions boxing et unboxing entre un type de structure et toute interface qu'il implémente.

spécification du langage C#

Pour plus d'informations, consultez la section Structs de la Spécification du langage C#.

Pour plus d'informations sur les fonctionnalités `struct`, consultez les notes de proposition des fonctionnalités suivantes :

- C# 7.2 - Structs en lecture seule
- C# 8 - Membres de l'instance en lecture seule
- C# 10 - Constructeurs de struct sans paramètre
- C# 10 - Autoriser l'expression with sur les structs
- C# 10 - Structs d'enregistrement
- C# 11 - Structs par défaut automatiques

Voir aussi

- Informations de référence sur C#
- Système de types C#
- Recommandations en matière de conception - Choix entre la classe et la struct
- Recommandations en matière de conception - Conception de struct

Types Tuple (référence C#)

Article • 29/05/2023

La fonctionnalité *tuples* fournit une syntaxe concise pour regrouper plusieurs éléments de données dans une structure de données légère. L'exemple suivant montre comment déclarer une variable tuple, l'initialiser et accéder à ses membres de données :

C#

```
(double, int) t1 = (4.5, 3);
Console.WriteLine($"Tuple with elements {t1.Item1} and {t1.Item2}.");
// Output:
// Tuple with elements 4.5 and 3.

(double Sum, int Count) t2 = (4.5, 3);
Console.WriteLine($"Sum of {t2.Count} elements is {t2.Sum}.");
// Output:
// Sum of 3 elements is 4.5.
```

Comme l'illustre l'exemple précédent, pour définir un type tuple, vous spécifiez des types de tous ses membres de données et, éventuellement, les noms de champ. Vous ne pouvez pas définir de méthodes dans un type tuple, mais vous pouvez utiliser les méthodes fournies par .NET, comme l'illustre l'exemple suivant :

C#

```
(double, int) t = (4.5, 3);
Console.WriteLine(t.ToString());
Console.WriteLine($"Hash code of {t} is {t.GetHashCode()}.");
// Output:
// (4.5, 3)
// Hash code of (4.5, 3) is 718460086.
```

Les types Tuple prennent en charge les opérateurs d'égalité `==` et `!=`. Pour plus d'informations, consultez la section Égalité Tuple.

Les types tuple sont des types valeur ; les éléments tuples sont des champs publics. Cela fait des tuples des types de valeurs *mutables*.

Vous pouvez définir des tuples avec un grand nombre arbitraire d'éléments :

C#

```
var t =
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12, 13, 14, 15, 16, 17, 18,
```

```
19, 20, 21, 22, 23, 24, 25, 26);
Console.WriteLine(t.Item26);  // output: 26
```

Cas d'usage de tuples

L'un des cas d'usage les plus courants de tuples est en tant que type de retour de méthode. Autrement dit, au lieu de définir des paramètres de méthodeout, vous pouvez regrouper des résultats de méthode dans un type de retour tuple, comme l'illustre l'exemple suivant :

C#

```
var xs = new[] { 4, 7, 9 };
var limits = FindMinMax(xs);
Console.WriteLine($"Limits of [{string.Join(" ", xs)}] are {limits.min} and
{limits.max}");
// Output:
// Limits of [4 7 9] are 4 and 9

var ys = new[] { -9, 0, 67, 100 };
var (minimum, maximum) = FindMinMax(ys);
Console.WriteLine($"Limits of [{string.Join(" ", ys)}] are {minimum} and
{maximum}");
// Output:
// Limits of [-9 0 67 100] are -9 and 100

(int min, int max) FindMinMax(int[] input)
{
    if (input is null || input.Length == 0)
    {
        throw new ArgumentException("Cannot find minimum and maximum of a
null or empty array.");
    }

    // Initialize min to MaxValue so every value in the input
    // is less than this initial value.
    var min = int.MaxValue;
    // Initialize max to MinValue so every value in the input
    // is greater than this initial value.
    var max = int.MinValue;
    foreach (var i in input)
    {
        if (i < min)
        {
            min = i;
        }
        if (i > max)
        {
            max = i;
        }
```

```
    }
    return (min, max);
}
```

Comme l'illustre l'exemple précédent, vous pouvez utiliser l'instance de tuple retournée directement ou la déconstruire dans des variables distinctes.

Vous pouvez également utiliser des types tuples au lieu de types anonymes ; par exemple, dans les requêtes LINQ. Pour plus d'informations, consultez Choix entre les types anonymes et tuples.

En règle générale, vous utilisez des tuples pour regrouper des éléments de données faiblement associés. Dans les API publiques, envisagez de définir une classe ou un type de structure.

Noms de champs Tuple

Vous spécifiez explicitement les noms des champs tuples dans une expression d'initialisation de tuple ou dans la définition d'un type tuple, comme l'illustre l'exemple suivant :

C#

```
var t = (Sum: 4.5, Count: 3);
Console.WriteLine($"Sum of {t.Count} elements is {t.Sum}.");

(double Sum, int Count) d = (4.5, 3);
Console.WriteLine($"Sum of {d.Count} elements is {d.Sum}.");
```

Si vous ne spécifiez pas de nom de champ, il peut être déduit du nom de la variable correspondante dans une expression d'initialisation de tuple, comme l'illustre l'exemple suivant :

C#

```
var sum = 4.5;
var count = 3;
var t = (sum, count);
Console.WriteLine($"Sum of {t.count} elements is {t.sum}.");
```

C'est ce que l'on appelle des initialiseurs de projection tuple. Le nom d'une variable n'est pas projeté sur un nom de champ tuple dans les cas suivants :

- Le nom du candidat est un nom de membre d'un type tuple, par exemple, `Item3`, `ToString` ou `Rest`.
- Le nom du candidat est un doublon d'un autre nom de champ tuple, explicite ou implicite.

Dans les cas précédents, vous spécifiez explicitement le nom d'un champ ou accédez à un champ par son nom par défaut.

Les noms par défaut des champs tuple sont `Item1`, `Item2`, `Item3`, et ainsi de suite. Vous pouvez toujours utiliser le nom par défaut d'un champ, même lorsqu'un nom de champ est spécifié explicitement ou déduit, comme l'illustre l'exemple suivant :

C#

```
var a = 1;
var t = (a, b: 2, 3);
Console.WriteLine($"The 1st element is {t.Item1} (same as {t.a}).");
Console.WriteLine($"The 2nd element is {t.Item2} (same as {t.b}).");
Console.WriteLine($"The 3rd element is {t.Item3}.");
// Output:
// The 1st element is 1 (same as 1).
// The 2nd element is 2 (same as 2).
// The 3rd element is 3.
```

L'affectation de tuple et les comparaisons d'égalité tuples ne prennent pas en compte les noms de champs.

Au moment de la compilation, le compilateur remplace les noms de champs qui ne sont pas par défaut par les noms par défaut correspondants. Par conséquent, les noms de champs explicitement spécifiés ou déduits ne sont pas disponibles au moment de l'exécution.

> **Conseil**
>
> Activez la règle de style de code .NET **IDE0037** pour définir une préférence sur les noms de champs tuples déduits ou explicites.

À compter de C# 12, vous pouvez spécifier un alias pour un type tuple avec une using directive. L'exemple suivant ajoute un alias `global using` pour un type tuple avec deux valeurs entières pour une valeur `Min` et `Max` autorisée :

C#

```
global using BandPass = (int Min, int Max);
```

Après avoir déclaré l'alias, vous pouvez utiliser le nom `BandPass` comme alias pour ce type de tuple :

```
C#

BandPass bracket = (40, 100);
Console.WriteLine($"The bandpass filter is {bracket.Min} to {bracket.Max}");
```

Un alias n'introduit pas de nouveau *type*, mais crée uniquement un synonyme pour un type existant. Vous pouvez déconstruire un tuple déclaré avec l'alias `BandPass` comme vous pouvez le faire avec son type de tuple sous-jacent :

```
C#

(int a , int b) = bracket;
Console.WriteLine($"The bracket is {a} to {b}");
```

Comme pour l'attribution ou la déconstruction de tuple, les noms de membres tuple n'ont pas besoin de correspondre ; les types le font.

De même, un deuxième alias avec le même arité et les mêmes types de membres peut être utilisé de manière interchangeable avec l'alias d'origine. Vous pouvez déclarer un deuxième alias :

```
C#

using Range = (int Minimum, int Maximum);
```

Vous pouvez affecter un tuple `Range` à un tuple `BandPass`. Comme pour toutes les affectations de tuples, les noms de champs ne doivent pas nécessairement correspondre, uniquement les types et l'arité.

```
C#

Range r = bracket;
Console.WriteLine($"The range is {r.Minimum} to {r.Maximum}");
```

Un alias pour un type tuple fournit plus d'informations sémantiques lorsque vous utilisez des tuples. Il n'introduit pas de nouveau type. Pour assurer la cohérence des types, vous devez plutôt déclarer une position record.

Affectation de tuple et déconstruction

Le langage C# prend en charge l'affectation entre les types tuples qui répondent aux deux conditions suivantes :

- Les deux types tuple ont le même nombre d'éléments
- Pour chaque position de tuple, le type de l'élément tuple droit est identique ou implicitement convertible au type de l'élément tuple gauche correspondant

Les valeurs d'élément Tuple sont attribuées en suivant l'ordre des éléments tuple. Les noms des champs tuples sont ignorés et non attribués, comme l'illustre l'exemple suivant :

C#

```
(int, double) t1 = (17, 3.14);
(double First, double Second) t2 = (0.0, 1.0);
t2 = t1;
Console.WriteLine($"{nameof(t2)}: {t2.First} and {t2.Second}");
// Output:
// t2: 17 and 3.14

(double A, double B) t3 = (2.0, 3.0);
t3 = t2;
Console.WriteLine($"{nameof(t3)}: {t3.A} and {t3.B}");
// Output:
// t3: 17 and 3.14
```

Vous pouvez également utiliser l'opérateur d'affectation `=` pour *déconstruire* une instance tuple dans des variables distinctes. Vous pouvez le faire de plusieurs façons :

- Utilisez le mot clé `var` en dehors des parenthèses pour déclarer des variables implicitement typées et laisser le compilateur déduire leurs types :

  C#

  ```
  var t = ("post office", 3.6);
  var (destination, distance) = t;
  Console.WriteLine($"Distance to {destination} is {distance}
  kilometers.");
  // Output:
  // Distance to post office is 3.6 kilometers.
  ```

- Déclarez explicitement le type de chaque variable entre parenthèses :

  C#

  ```
  var t = ("post office", 3.6);
  (string destination, double distance) = t;
  Console.WriteLine($"Distance to {destination} is {distance}
  ```

```
kilometers.");
// Output:
// Distance to post office is 3.6 kilometers.
```

- Déclarez certains types explicitement et d'autres types implicitement (avec `var`) entre parenthèses :

```
C#

var t = ("post office", 3.6);
(var destination, double distance) = t;
Console.WriteLine($"Distance to {destination} is {distance}
kilometers.");
// Output:
// Distance to post office is 3.6 kilometers.
```

- Utilisez des variables existantes :

```
C#

var destination = string.Empty;
var distance = 0.0;

var t = ("post office", 3.6);
(destination, distance) = t;
Console.WriteLine($"Distance to {destination} is {distance}
kilometers.");
// Output:
// Distance to post office is 3.6 kilometers.
```

La destination d'une expression de déconstruction peut inclure à la fois des variables existantes et des variables déclarées dans la déclaration de déconstruction.

Vous pouvez également combiner la déconstruction avec la correspondance de modèle pour inspecter les caractéristiques des champs d'un tuple. L'exemple suivant boucle plusieurs entiers et imprime ceux qui sont divisibles par 3. Il déconstruit le résultat du tuple de Int32.DivRem et correspond à un `Remainder` de 0 :

```
C#

for (int i = 4; i < 20;  i++)
{
    if (Math.DivRem(i, 3) is ( Quotient: var q, Remainder: 0 ))
    {
        Console.WriteLine($"{i} is divisible by 3, with quotient {q}");
    }
}
```

Pour plus d'informations sur la déconstruction des tuples et autres types, consultez Déconstruction des tuples et autres types.

Égalité des tuples

Les types tuple prennent en charge les opérateurs `==` et `!=`. Ces opérateurs comparent les membres de l'opérande gauche aux membres correspondants de l'opérande droit en suivant l'ordre des éléments tuples.

C#

```
(int a, byte b) left = (5, 10);
(long a, int b) right = (5, 10);
Console.WriteLine(left == right);  // output: True
Console.WriteLine(left != right);  // output: False

var t1 = (A: 5, B: 10);
var t2 = (B: 5, A: 10);
Console.WriteLine(t1 == t2);  // output: True
Console.WriteLine(t1 != t2);  // output: False
```

Comme l'illustre l'exemple précédent, les opérations `==` et `!=` ne prennent pas en compte les noms de champs tuple.

Deux tuples sont comparables lorsque les deux conditions suivantes sont remplies :

- Les deux tuples ont le même nombre d'éléments. Par exemple, `t1 != t2` ne compile pas si `t1` et `t2` possèdent un nombre d'éléments différent.
- Pour chaque position de tuple, les éléments correspondants des opérandes tuples gauche et droit sont comparables aux opérateurs `==` et `!=`. Par exemple, `(1, (2, 3)) == ((1, 2), 3)` ne compile pas car `1` n'est pas comparable à `(1, 2)`.

Les opérateurs `==` et `!=` comparent les tuples en court-circuit. Autrement dit, une opération s'arrête dès qu'elle rencontre une paire d'éléments inégaux ou atteint les extrémités des tuples. Toutefois, avant toute comparaison, *tous* les éléments de tuple sont évalués, comme l'illustre l'exemple suivant :

C#

```
Console.WriteLine((Display(1), Display(2)) == (Display(3), Display(4)));

int Display(int s)
{
    Console.WriteLine(s);
    return s;
```

```
}
// Output:
// 1
// 2
// 3
// 4
// False
```

Les tuples en tant que paramètres sortants

En règle générale, vous refactorisez une méthode qui a des paramètres out dans une méthode qui retourne un tuple. Toutefois, il existe des cas dans lesquels un paramètre `out` peut être d'un type tuple. L'exemple suivant montre comment utiliser des tuples en tant que paramètres `out` :

C#

```
var limitsLookup = new Dictionary<int, (int Min, int Max)>()
{
    [2] = (4, 10),
    [4] = (10, 20),
    [6] = (0, 23)
};

if (limitsLookup.TryGetValue(4, out (int Min, int Max) limits))
{
    Console.WriteLine($"Found limits: min is {limits.Min}, max is
{limits.Max}");
}
// Output:
// Found limits: min is 10, max is 20
```

Tuples vs `System.Tuple`

Les tuples C#, qui sont soutenus par des types System.ValueTuple, sont différents des tuples représentés par les types System.Tuple. Les principales différences sont les suivantes :

- Les types `System.ValueTuple` sont des types valeur. Les types `System.Tuple` sont des types de référence.
- Les types `System.ValueTuple` sont mutables. Les types `System.Tuple` sont immuables.
- Les membres de données de type `System.ValueTuple` sont des champs. Les membres de données de type `System.Tuple` sont des propriétés.

spécification du langage C#

Pour plus d'informations, consultez les pages suivantes :

- Types de tuples
- Opérateurs d’égalité de tuple

Voir aussi

- Informations de référence sur C#
- Types de valeur
- Choisir entre les types anonymes et tuples
- System.ValueTuple

Types valeur pouvant accepter la valeur Null (référence C#)

Article • 07/04/2023

Un *type valeur nullable* `T?` représente toutes les valeurs de son type valeur `T` sous-jacent et une valeur null supplémentaire. Par exemple, vous pouvez affecter l'une des trois valeurs suivantes à une `bool?` variable : `true`, `false` ou `null`. Un type `T` de valeur sous-jacent ne peut pas être un type de valeur nullable lui-même.

Tout type de valeur nullable est une instance de la structure générique System.Nullable<T>. Vous pouvez faire référence à un type valeur nullable avec un type `T` sous-jacent dans l'un des formulaires interchangeables suivants : `Nullable<T>` ou `T?`.

Généralement, vous utilisez un type valeur nullable lorsque vous avez besoin de représenter la valeur non définie d'un type valeur sous-jacent. Par exemple, une variable booléenne ou `bool`, ne peut être que `true` ou `false`. Toutefois, dans certaines applications, une valeur de variable peut être non définie ou manquante. Par exemple, un champ de base de données peut contenir `true` ou `false`, ou il peut ne contenir aucune valeur, c'est-à-dire `NULL`. Vous pouvez utiliser le type `bool?` dans ce scénario.

Déclaration et affectation

Comme un type valeur est implicitement convertible en type valeur nullable correspondant, vous pouvez assigner une valeur à une variable d'un type valeur nullable comme vous le feriez pour son type valeur sous-jacent. Vous pouvez également affecter la valeur `null`. Par exemple :

C#

```
double? pi = 3.14;
char? letter = 'a';

int m2 = 10;
int? m = m2;

bool? flag = null;

// An array of a nullable value type:
int?[] arr = new int?[10];
```

La valeur par défaut d'un type valeur nullable représente `null`, c'est-à-dire qu'il s'agit d'une instance dont la propriété Nullable<T>.HasValue retourne `false`.

Examen d'une instance d'un type valeur nullable

Vous pouvez utiliser l'opérateur is avec un modèle de type pour examiner une instance d'un type valeur nullable pour `null` et récupérer une valeur d'un type sous-jacent :

```
C#
```

```
int? a = 42;
if (a is int valueOfA)
{
    Console.WriteLine($"a is {valueOfA}");
}
else
{
    Console.WriteLine("a does not have a value");
}
// Output:
// a is 42
```

Vous pouvez toujours utiliser les propriétés en lecture seule suivantes pour examiner et obtenir une valeur d'une variable de type valeur nullable :

- Nullable<T>.HasValue indique si une instance d'un type valeur nullable a une valeur de son type sous-jacent.
- Nullable<T>.Value obtient la valeur d'un type sous-jacent si HasValue est `true`. Si HasValue est `false`, la propriété Value lève une InvalidOperationException.

L'exemple suivant utilise la propriété `HasValue` pour tester si la variable contient une valeur avant de l'afficher :

```
C#
```

```
int? b = 10;
if (b.HasValue)
{
    Console.WriteLine($"b is {b.Value}");
}
else
{
    Console.WriteLine("b does not have a value");
}
```

```
// Output:
// b is 10
```

Vous pouvez également comparer une variable de type nullable avec `null` au lieu d'utiliser la propriété `HasValue`, comme le montre l'exemple suivant :

C#

```
int? c = 7;
if (c != null)
{
    Console.WriteLine($"c is {c.Value}");
}
else
{
    Console.WriteLine("c does not have a value");
}
// Output:
// c is 7
```

Conversion d'un type valeur nullable en type sous-jacent

Si vous souhaitez affecter une valeur d'un type valeur nullable à une variable de type valeur non nullable, vous devrez peut-être spécifier la valeur à attribuer à la place de `null`. Utilisez l'opérateur de coalescence nulle?? pour ce faire (vous pouvez également utiliser la méthode Nullable<T>.GetValueOrDefault(T) dans le même but) :

C#

```
int? a = 28;
int b = a ?? -1;
Console.WriteLine($"b is {b}");  // output: b is 28

int? c = null;
int d = c ?? -1;
Console.WriteLine($"d is {d}");  // output: d is -1
```

Si vous souhaitez utiliser la valeur par défaut du type valeur sous-jacent à la place de `null`, utilisez la méthode Nullable<T>.GetValueOrDefault().

Vous pouvez également convertir explicitement un type valeur nullable en type non nullable, comme le montre l'exemple suivant :

C#

```
int? n = null;

//int m1 = n;    // Doesn't compile
int n2 = (int)n; // Compiles, but throws an exception if n is null
```

Au moment de l'exécution, si la valeur d'un type nullable est `null`, le cast explicite lève une`InvalidOperationException`.

Un type valeur non-nullable `T` est implicitement converti en type valeur nullable correspondant `T?`.

Opérateurs levés

Les opérateurs unaires et binaires prédéfinis ou tous les opérateurs surchargés pris en charge par un type valeur `T` sont également pris en charge par le type de valeur nullable `T?` correspondant. Ces opérateurs, également appelés *opérateurs levés* produisent `null` si l'un des opérandes ou les deux sont `null` ; sinon, l'opérateur utilise les valeurs contenues de ses opérandes pour calculer le résultat. Par exemple :

C#

```
int? a = 10;
int? b = null;
int? c = 10;

a++;        // a is 11
a = a * c;  // a is 110
a = a + b;  // a is null
```

> ⓘ **Notes**
>
> Pour le type `bool?`, les opérateurs `&` et `|` prédéfinis ne suivent pas les règles décrites dans cette section : le résultat d'une évaluation peut être non null, même si l'un des opérandes est `null`. Pour plus d'informations, voir la section **Opérateurs logiques booléens Nullable** de l'article **Opérateurs logiques booléens**.

Pour les opérateurs de comparaison `<`, `>`, `<=` et `>=`, si l'un ou les deux opérandes sont `null`, le résultat est `false` ; sinon, les valeurs contenues des opérandes sont comparées. Ne partez pas du principe que parce qu'une comparaison spécifique (par exemple `<=`) retourne `false`, la comparaison opposée (`>`) retourne `true`. L'exemple suivant montre que 10 n'est

- ni supérieur ou égal à `null`
- ni inférieur à `null`

C#

```
int? a = 10;
Console.WriteLine($"{a} >= null is {a >= null}");
Console.WriteLine($"{a} < null is {a < null}");
Console.WriteLine($"{a} == null is {a == null}");
// Output:
// 10 >= null is False
// 10 < null is False
// 10 == null is False

int? b = null;
int? c = null;
Console.WriteLine($"null >= null is {b >= c}");
Console.WriteLine($"null == null is {b == c}");
// Output:
// null >= null is False
// null == null is True
```

Pour l'opérateur d'égalité `==`, si les deux opérandes sont `null`, le résultat est `true`, si une seule des opérandes est `null`, le résultat est `false` ; sinon, les valeurs contenues des opérandes sont comparées.

Pour l'opérateur d'égalité `!=`, si les deux opérandes sont `null`, le résultat est `false`, si un seul des opérandes est `null`, le résultat est `true` ; sinon, les valeurs contenues des opérandes sont comparées.

S'il existe une conversion définie par l'utilisateur entre deux types de valeurs, la même conversion peut également être utilisée entre les types de valeurs nullables correspondants.

Boxing et unboxing

Une instance d'un type valeur nullable `T?` est boxée comme suit :

- Si HasValue retourne `false`, la référence null est générée.
- Si HasValue retourne `true`, la valeur correspondante du type valeur sous-jacent `T` est boxed, pas l'instance de Nullable<T>.

Vous pouvez effectuer l'unboxing du type valeur `T` vers le type `T?` correspondant, comme le montre l'exemple suivant :

C#

```
int a = 41;
object aBoxed = a;
int? aNullable = (int?)aBoxed;
Console.WriteLine($"Value of aNullable: {aNullable}");

object aNullableBoxed = aNullable;
if (aNullableBoxed is int valueOfA)
{
    Console.WriteLine($"aNullableBoxed is boxed int: {valueOfA}");
}
// Output:
// Value of aNullable: 41
// aNullableBoxed is boxed int: 41
```

Guide pratique pour identifier un type valeur nullable

L'exemple suivant montre comment déterminer si une instance System.Type représente un type valeur nullable construit, autrement dit, le type System.Nullable<T> avec un paramètre de type spécifié `T` :

C#

```
Console.WriteLine($"int? is {(IsNullable(typeof(int?)) ? "nullable" : "non
nullable")} value type");
Console.WriteLine($"int is {(IsNullable(typeof(int)) ? "nullable" : "non-
nullable")} value type");

bool IsNullable(Type type) => Nullable.GetUnderlyingType(type) != null;

// Output:
// int? is nullable value type
// int is non-nullable value type
```

Comme l'illustre l'exemple, vous utilisez l'opérateur typeof pour créer une instance System.Type.

Si vous souhaitez déterminer si une instance est d'un type valeur nullable, n'utilisez pas la méthode Object.GetType pour obtenir une instance Type à tester avec le code précédent. Quand vous appelez la méthode Object.GetType sur une instance d'un type valeur nullable, l'instance est boxed à Object. Comme le boxing d'une instance non nulle d'un type valeur nullable équivaut au boxing d'une valeur du type sous-jacent, GetType retourne une instance Type qui représente le type sous-jacent d'un type valeur nullable :

C#

```
int? a = 17;
Type typeOfA = a.GetType();
Console.WriteLine(typeOfA.FullName);
// Output:
// System.Int32
```

Par ailleurs, n'utilisez pas l'opérateur is pour déterminer si une instance est d'un type valeur nullable. Comme le montre l'exemple suivant, vous ne pouvez pas distinguer les types d'une instance et son instance de type sous-jacent à l'aide de l'opérateur `is` :

C#

```
int? a = 14;
if (a is int)
{
    Console.WriteLine("int? instance is compatible with int");
}

int b = 17;
if (b is int?)
{
    Console.WriteLine("int instance is compatible with int?");
}
// Output:
// int? instance is compatible with int
// int instance is compatible with int?
```

Au lieu de cela, utilisez le Nullable.GetUnderlyingType du premier exemple et typeof pour vérifier si une instance est de type valeur nullable.

> ⓘ **Notes**
>
> Les méthodes décrites dans cette section ne s'appliquent pas aux **types de référence nullables**.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Types Nullable
- Opérateurs levés
- Conversions nullables implicites

- Conversions nullables explicites
- Opérateurs de conversion levés

Voir aussi

- Informations de référence sur C#
- Que signifie réellement « lifted » ?
- System.Nullable<T>
- System.Nullable
- Nullable.GetUnderlyingType
- Types références Nullables

Types référence (référence C#)

Article • 07/04/2023

Il existe deux genres de types en C# : les types référence et les types valeur. Les variables des types référence font référence à leurs données (objets), tandis que les variables des types valeur contiennent directement leurs données. Avec les types référence, deux variables peuvent faire référence au même objet ; par conséquent, les opérations sur une variable peuvent affecter le même objet référencé par l'autre variable. Avec les types valeur, chaque variable dispose de sa propre copie des données et les opérations sur une variable ne peuvent pas affecter une autre (sauf pour les variables de paramètre `in`, `ref` et `out` ; voir le modificateur de paramètre in, ref et out).

Les mots clés suivants sont utilisés pour déclarer des types référence :

- class
- interface
- delegate
- enregistrement

C# fournit également les types référence intégrés suivants :

- dynamic
- object
- string

Voir aussi

- Informations de référence sur C#
- Mots clés C#
- Types de pointeur
- Types de valeur

Types référence intégrés (Référence C#)

Article • 10/05/2023

C# a de nombreux types référence intégrés. Ils ont des mots clés ou des opérateurs qui sont synonymes pour un type dans la bibliothèque .NET.

Type d'objet.

Le type `object` est un alias de System.Object dans .NET. Dans le système de type unifié de C#, tous les types (les types référence et valeur, prédéfinis ou définis par l'utilisateur) héritent directement ou indirectement du type System.Object. Vous pouvez assigner des valeurs de tout type aux variables de type `object`. Toute variable `object` peut être attribuée à sa valeur par défaut à l'aide du littéral `null`. Quand une variable d'un type valeur est convertie en type objet, elle est dite *boxed*. Quand une variable de type `object` est convertie en type valeur, elle est dite *unboxed*. Pour plus d'informations, consultez Conversion boxing et unboxing.

Type de chaîne

Le type `string` représente une séquence de zéro, un ou plusieurs caractères Unicode. `string` est un alias de System.String dans .NET.

Bien que `string` soit un type référence, les opérateurs d'égalité== (!= et) sont définis pour comparer les valeurs des objets `string`, pas les références. L'égalité basée sur les valeurs permet de tester l'égalité de chaînes de façon plus intuitive. Par exemple :

C#

```
string a = "hello";
string b = "h";
// Append to contents of 'b'
b += "ello";
Console.WriteLine(a == b);
Console.WriteLine(object.ReferenceEquals(a, b));
```

Cet exemple précédent affiche « True », puis « False », car le contenu des chaînes est équivalent, mais `a` et `b` ne font pas référence à la même instance de chaîne.

L'opérateur + concatène les chaînes :

C#

```
string a = "good " + "morning";
```

Le code précédent crée un objet String qui contient « good morning ».

Les chaînes sont *immuables* : le contenu d'un objet de chaîne ne peut pas être modifié une fois l'objet créé. Par exemple, lorsque vous écrivez ce code, le compilateur crée en fait un nouvel objet String pour stocker la nouvelle séquence de caractères, et ce nouvel objet est attribué à `b`. La mémoire allouée pour `b` (quand elle contenait la chaîne « h ») est alors éligible pour le garbage collection.

C#

```
string b = "h";
b += "ello";
```

L'`[]`opérateurpeut être utilisé pour un accès en lecture seule aux différents caractères d'une chaîne. Les valeurs index valides commencent à `0` et doivent être inférieures à la longueur de la chaîne :

C#

```
string str = "test";
char x = str[2];  // x = 's';
```

De la même façon, l'opérateur `[]` permet également d'itérer sur chaque caractère dans une chaîne :

C#

```
string str = "test";

for (int i = 0; i < str.Length; i++)
{
  Console.Write(str[i] + " ");
}
// Output: t e s t
```

Littéraux de chaîne

Les littéraux de chaîne sont de type `string` et peuvent être écrits sous trois formes, brut, entre guillemets et verbatim.

Les littéraux de chaîne brute sont disponibles à partir de C# 11. Les littéraux de chaîne brute peuvent contenir du texte arbitraire sans nécessiter de séquences d'échappement. Les littéraux de chaîne brute peuvent inclure des espaces blancs et de nouvelles lignes, des guillemets incorporés et d'autres caractères spéciaux. Les littéraux de chaîne brutes sont placés entre trois guillemets doubles (""") au minimum :

C#

```
"""
This is a multi-line
    string literal with the second line indented.
"""
```

Vous pouvez même inclure une séquence de trois (ou plus) guillemets doubles. Si votre texte nécessite une séquence incorporée de guillemets, vous commencez et terminez le littéral de chaîne brute avec plus de guillemets, selon les besoins :

C#

```
"""""
This raw string literal has four """", count them: """" four!
embedded quote characters in a sequence. That's why it starts and ends
with five double quotes.

You could extend this example with as many embedded quotes as needed for
your text.
"""""
```

Les littéraux de chaîne brute ont généralement les séquences de guillemets de début et de fin sur des lignes distinctes du texte incorporé. Les littéraux de chaîne brute multiligne prennent en charge les chaînes qui sont elles-mêmes des chaînes entre guillemets :

C#

```
var message = """
"This is a very important message."
""";
Console.WriteLine(message);
// output: "This is a very important message."
```

Lorsque les guillemets de début et de fin se trouvent sur des lignes distinctes, les nouvelles lignes qui suivent le guillemet d'ouverture et qui précèdent le guillemet de fin ne sont pas incluses dans le contenu final. La séquence de guillemets de fermeture dicte

la colonne la plus à gauche pour le littéral de chaîne. Vous pouvez mettre en retrait un littéral de chaîne brute pour qu'il corresponde au format de code global :

```
C#
```

```
var message = """
    "This is a very important message."
    """;
Console.WriteLine(message);
// output: "This is a very important message."
// The leftmost whitespace is not part of the raw string literal
```

Les colonnes à droite de la séquence de guillemets de fin sont conservées. Ce comportement active les chaînes brutes pour les formats de données tels que JSON, YAML ou XML, comme illustré dans l'exemple suivant :

```
C#
```

```
var json= """
    {
        "prop": 0
    }
    """;
```

Le compilateur émet une erreur si l'une des lignes de texte s'étend à gauche de la séquence de guillemets de fermeture. Les séquences de guillemets d'ouverture et de fermeture peuvent se trouver sur la même ligne, à condition que le littéral de chaîne ne commence ni ne se termine par un caractère de guillemet :

```
C#
```

```
var shortText = """He said "hello!" this morning.""";
```

Vous pouvez combiner des littéraux de chaîne bruts avec l'interpolation de chaîne pour inclure des guillemets et des accolades dans la chaîne de sortie.

Les littéraux de chaîne entre guillemets sont placés entre guillemets doubles (") :

```
C#
```

```
"good morning"  // a string literal
```

Les littéraux de chaîne peuvent contenir tout littéral de caractère. Les séquences d'échappement sont incluses. L'exemple suivant utilise la séquence d'échappement `\\` pour la barre oblique inverse, `\u0066` pour la lettre f et `\n` pour un saut de ligne.

C#

```
string a = "\\\u0066\n F";
Console.WriteLine(a);
// Output:
// \f
//  F
```

> ⓘ Notes
>
> Le code d'échappement `\udddd` (où `dddd` est un nombre à quatre chiffres) représente le caractère Unicode U+`dddd`. Les codes d'échappement Unicode à huit chiffres sont également reconnus : `\Uddddddddd`.

Les littéraux de chaîne textuelle commencent par `@` et sont placés entre guillemets doubles. Par exemple :

C#

```
@"good morning"  // a string literal
```

L'avantage des chaînes textuelles est que les séquences d'échappement *ne sont pas* traitées, ce qui facilite l'écriture. Par exemple, le texte suivant correspond à un nom de fichier Windows complet :

C#

```
@"c:\Docs\Source\a.txt"  // rather than "c:\\Docs\\Source\\a.txt"
```

Pour inclure un guillemet double dans une chaîne @-quoted, doublez-le :

C#

```
@"""Ahoy!"" cried the captain." // "Ahoy!" cried the captain.
```

Littéraux de chaîne UTF-8

Les chaînes dans .NET sont stockées à l'aide de l'encodage UTF-16. UTF-8 est la norme pour les protocoles Web et d'autres bibliothèques importantes. À compter de C# 11, vous pouvez ajouter le suffixe `u8` à un littéral de chaîne pour spécifier l'encodage UTF-8. Les littéraux UTF-8 sont stockés en tant qu'objets `ReadOnlySpan<byte>`. Le type naturel

d'un littéral de chaîne UTF-8 est `ReadOnlySpan<byte>`. L'utilisation d'un littéral de chaîne UTF-8 crée une déclaration plus claire que la déclaration de l'équivalent System.ReadOnlySpan<T>, comme indiqué dans le code suivant :

```
C#

ReadOnlySpan<byte> AuthWithTrailingSpace = new byte[] { 0x41, 0x55, 0x54,
0x48, 0x20 };
ReadOnlySpan<byte> AuthStringLiteral = "AUTH "u8;
```

Pour stocker un littéral de chaîne UTF-8 en tant que tableau, vous devez utiliser ReadOnlySpan<T>.ToArray() pour copier les octets contenant le littéral dans le tableau mutable :

```
C#

byte[] AuthStringLiteral = "AUTH "u8.ToArray();
```

Les littéraux de chaîne UTF-8 ne sont pas des constantes au moment de la compilation ; ce sont des constantes d'exécution. Par conséquent, ils ne peuvent pas être utilisés comme valeur par défaut pour un paramètre facultatif. Les littéraux de chaîne UTF-8 ne peuvent pas être combinés avec l'interpolation de chaîne. Vous ne pouvez pas utiliser le jeton `$` et le suffixe `u8` sur la même expression de chaîne.

Type de délégué

La déclaration d'un type délégué est semblable à une signature de méthode. Elle a une valeur de retour et un nombre quelconque de paramètres de type quelconque :

```
C#

public delegate void MessageDelegate(string message);
public delegate int AnotherDelegate(MyType m, long num);
```

Dans .NET, les types `System.Action` et `System.Func` fournissent des définitions génériques pour de nombreux délégués courants. Vous n'avez probablement pas besoin de définir de nouveaux types de délégué personnalisés. À la place, vous pouvez créer des instanciations des types génériques fournis.

Un `delegate` est un type référence qui peut être utilisé pour encapsuler une méthode anonyme ou nommée. Les délégués sont comparables aux pointeurs fonction en C++, mais ils offrent l'avantage d'être sûrs et de type sécurisé. Pour les applications de

délégués, consultez Délégués et Délégués génériques. Les délégués sont la base des événements. Un délégué peut être instancié en l'associant à une méthode nommée ou anonyme.

Le délégué doit être instancié avec une méthode ou une expression lambda qui a un type de retour compatible et des paramètres d'entrée. Pour plus d'informations sur le degré de variance autorisé dans la signature de méthode, consultez Variance dans les délégués. Pour une utilisation avec des méthodes anonymes, le délégué et le code à lui associer sont déclarés ensemble.

La combinaison ou la suppression de délégués échoue avec une exception d'exécution lorsque les types de délégués impliqués au moment de l'exécution sont différents en raison de la conversion de variantes. L'exemple suivant illustre une situation qui peut échouer :

C#

```
Action<string> stringAction = str => {};
Action<object> objectAction = obj => {};

// Valid due to implicit reference conversion of
// objectAction to Action<string>, but may fail
// at run time.
Action<string> combination = stringAction + objectAction;
```

Vous pouvez créer un délégué avec le type d'exécution correct en créant un nouvel objet délégué. L'exemple suivant montre comment cette solution de contournement peut être appliquée à l'exemple précédent.

C#

```
Action<string> stringAction = str => {};
Action<object> objectAction = obj => {};

// Creates a new delegate instance with a runtime type of Action<string>.
Action<string> wrappedObjectAction = new Action<string>(objectAction);

// The two Action<string> delegate instances can now be combined.
Action<string> combination = stringAction + wrappedObjectAction;
```

À compter de C# 9, vous pouvez déclarer *des pointeurs de fonction*, qui utilisent une syntaxe similaire. Un pointeur de fonction utilise l'instruction `calli` au lieu d'instancier un type délégué et d'appeler la méthode virtuelle `Invoke`.

Type dynamique

Le type `dynamic` indique que l'utilisation de la variable et des références à ses membres contourne la vérification du type au moment de la compilation. Au lieu de cela, ces opérations sont résolues au moment de l'exécution. Le type `dynamic` simplifie l'accès aux API COM telles que les API Office Automation, aux API dynamiques telles que les bibliothèques IronPython et au modèle DOM (Document Object Model) HTML.

Le type `dynamic` se comporte comme le type `object` dans la plupart des cas. En particulier, toute expression non null peut être convertie en type `dynamic`. Le type `dynamic` diffère de `object` en cela que les opérations qui contiennent des expressions de type `dynamic` ne sont pas résolues et leur type n'est pas vérifié par le compilateur. Le compilateur empaquète des informations sur l'opération, qui sont ensuite utilisées pour évaluer l'opération au moment de l'exécution. Dans le cadre du processus, les variables de type `dynamic` sont compilées dans des variables de type `object`. Ainsi, le type `dynamic` existe seulement au moment de la compilation, et non au moment de l'exécution.

L'exemple suivant compare une variable de type `dynamic` à une variable de type `object`. Pour vérifier le type de chaque variable au moment de la compilation, placez le pointeur de la souris sur `dyn` ou `obj` dans les instructions `WriteLine`. Copiez le code suivant dans un éditeur où IntelliSense est disponible. IntelliSense affiche **dynamic** pour `dyn` et **object** pour `obj`.

C#

```
class Program
{
    static void Main(string[] args)
    {
        dynamic dyn = 1;
        object obj = 1;

        // Rest the mouse pointer over dyn and obj to see their
        // types at compile time.
        System.Console.WriteLine(dyn.GetType());
        System.Console.WriteLine(obj.GetType());
    }
}
```

Les instructions WriteLine affichent les types d'exécution de `dyn` et `obj`. À ce stade, tous deux ont le même type, entier. La sortie suivante est produite :

Console

```
System.Int32
System.Int32
```

Pour voir la différence entre `dyn` et `obj` au moment de la compilation, ajoutez les deux lignes suivantes entre les déclarations et les instructions `WriteLine` de l'exemple précédent.

C#

```
dyn = dyn + 3;
obj = obj + 3;
```

Une erreur du compilateur est signalée pour la tentative d'ajout d'un entier et un d'objet dans l'expression `obj + 3`. Toutefois, aucune erreur n'est signalée pour `dyn + 3`. L'expression qui contient `dyn` n'est pas vérifié au moment de la compilation, car le type de `dyn` est `dynamic`.

L'exemple suivant utilise `dynamic` dans plusieurs déclarations. La méthode `Main` compare également la vérification de type au moment de la compilation avec la vérification de type au moment de l'exécution.

C#

```
using System;

namespace DynamicExamples
{
    class Program
    {
        static void Main(string[] args)
        {
            ExampleClass ec = new ExampleClass();
            Console.WriteLine(ec.ExampleMethod(10));
            Console.WriteLine(ec.ExampleMethod("value"));

            // The following line causes a compiler error because
ExampleMethod
            // takes only one argument.
            //Console.WriteLine(ec.ExampleMethod(10, 4));

            dynamic dynamic_ec = new ExampleClass();
            Console.WriteLine(dynamic_ec.ExampleMethod(10));

            // Because dynamic_ec is dynamic, the following call to
ExampleMethod
            // with two arguments does not produce an error at compile time.
            // However, it does cause a run-time error.
            //Console.WriteLine(dynamic_ec.ExampleMethod(10, 4));
        }
    }
```

```
    class ExampleClass
    {
        static dynamic _field;
        dynamic Prop { get; set; }

        public dynamic ExampleMethod(dynamic d)
        {
            dynamic local = "Local variable";
            int two = 2;

            if (d is int)
            {
                return local;
            }
            else
            {
                return two;
            }
        }
    }
}
// Results:
// Local variable
// 2
// Local variable
```

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- §8.2.3 Le type d'objet
- §8.2.4 Le type dynamique
- §8.2.5 Le type de chaîne
- §8.2.8 Les types délégués
- C# 11 - Les littéraux de chaîne bruts
- C# 11 - Les littéraux de chaîne bruts

Voir aussi

- Référence C#
- Mots clés C#
- Événements
- Utilisation du type dynamic
- Bonnes pratiques pour l'utilisation de chaînes
- Opérations de chaînes de base

- Création de chaînes
- Opérateurs de conversion et de test de type
- Guide pratique : caster de manière sécurisée avec les critères spéciaux, ainsi que les opérateurs is et as
- Procédure pas à pas : création et utilisation d'objets dynamiques
- System.Object
- System.String
- System.Dynamic.DynamicObject

Enregistrements (référence C#)

Article • 25/05/2023

À compter de C# 9, vous utilisez le modificateur `record` pour définir un type de référence qui fournit des fonctionnalités intégrées pour l'encapsulation des données. C# 10 permet la syntaxe `record class` en tant que synonyme, afin de clarifier un type référence et `record struct` de définir un type valeur avec des fonctionnalités similaires.

Lorsque vous déclarez un constructeur principal sur un enregistrement, le compilateur génère des propriétés publiques pour les paramètres du constructeur principal. Les paramètres du constructeur principal d'un enregistrement sont appelés *paramètres positionnels*. Le compilateur crée des *propriétés positionnelles* qui reflètent le constructeur principal ou les paramètres positionnels. Le compilateur ne synthétise pas les propriétés des paramètres de constructeur principal sur les types qui n'ont pas le modificateur `record`.

Les deux exemples suivants illustrent les types référence `record` (ou `record class`) :

C#

```
public record Person(string FirstName, string LastName);
```

C#

```
public record Person
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
};
```

Les deux exemples suivants illustrent `record struct` types valeur :

C#

```
public readonly record struct Point(double X, double Y, double Z);
```

C#

```
public record struct Point
{
    public double X { get; init; }
    public double Y { get; init; }
```

```
    public double Z { get; init; }
}
```

Vous pouvez également créer des enregistrements avec des propriétés et des champs mutables :

C#

```
public record Person
{
    public required string FirstName { get; set; }
    public required string LastName { get; set; }
};
```

Les structs d'enregistrement peuvent également être mutables, les structs d'enregistrement positionnel, comme les structs d'enregistrement sans paramètres positionnels :

C#

```
public record struct DataMeasurement(DateTime TakenAt, double Measurement);
```

C#

```
public record struct Point
{
    public double X { get; set; }
    public double Y { get; set; }
    public double Z { get; set; }
}
```

Bien que les enregistrements puissent être mutables, ils sont principalement destinés à prendre en charge les modèles de données immuables. Le type d'enregistrement procure les fonctionnalités suivantes :

- Syntaxe concise pour la création d'un type de référence avec des propriétés immuables
- Comportement intégré utile pour un type de référence centré sur les données :
  - Égalité des valeurs
  - Syntaxe concise pour les mutations non destructrices
  - Mise en forme intégrée pour l'affichage
- Prise en charge des hiérarchies d'héritage

Les exemples précédents montrent certaines distinctions entre les enregistrements qui sont des types référence et les enregistrements qui sont des types valeur :

- Un `record` ou un `record class` déclare un type référence. Le mot clé `class` est facultatif, mais peut apporter de la clarté aux lecteurs. Un `record struct` déclare une classe de valeur.
- Les propriétés positionnelles sont *immuables* dans un `record class` et un `readonly record struct`. Elles sont *mutables* dans un `record struct`.

Le reste de cet article traite des types `record class` et `record struct`. Les différences sont détaillées dans chaque section. Vous devez choisir entre un `record class` et un `record struct`, ce qui est similaire à un choix entre un `class` et un `struct`. Le terme *enregistrement* est utilisé pour décrire le comportement qui s'applique à tous les types d'enregistrements. `record struct` ou `record class` est utilisé pour décrire un comportement qui s'applique uniquement aux types de struct ou de classe, respectivement. Le type `record` a été introduit dans C# 9 ; les types `record struct` ont été introduits dans C# 10.

Syntaxe positionnelle pour la définition de propriété

Vous pouvez utiliser des paramètres positionnels pour déclarer des propriétés d'un enregistrement et initialiser les valeurs de propriété lorsque vous créez une instance :

C#

```
public record Person(string FirstName, string LastName);

public static void Main()
{
    Person person = new("Nancy", "Davolio");
    Console.WriteLine(person);
    // output: Person { FirstName = Nancy, LastName = Davolio }
}
```

Lorsque vous utilisez la syntaxe positionnelle pour la définition de propriété, le compilateur crée :

- Une propriété publique implémentée automatiquement pour chaque paramètre positionnel fourni dans la déclaration d'enregistrement.
  - Pour les types `record` et les types `readonly record struct` : une propriété init uniquement .
  - Pour les types `record struct` : une propriété en lecture/écriture.
- Un constructeur principal dont les paramètres correspondent aux paramètres positionnels de la déclaration d'enregistrement.

- Pour les types de struct d'enregistrement, constructeur sans paramètre qui définit chaque champ à sa valeur par défaut.
- Une méthode `Deconstruct` avec un paramètre `out` pour chaque paramètre positionnel fourni dans la déclaration d'enregistrement. La méthode déconstruit les propriétés définies à l'aide de la syntaxe positionnelle ; elle ignore les propriétés définies à l'aide de la syntaxe de propriété standard.

Vous pouvez ajouter des attributs à l'un de ces éléments créés par le compilateur à partir de la définition d'enregistrement. Vous pouvez ajouter une *cible* à n'importe quel attribut que vous appliquez aux propriétés de l'enregistrement positionnel. L'exemple suivant applique le System.Text.Json.Serialization.JsonPropertyNameAttribute à chaque propriété de l'enregistrement `Person`. La cible `property:` indique que l'attribut est appliqué à la propriété générée par le compilateur. D'autres valeurs sont `field:`, afin d'appliquer l'attribut au champ et `param:`, afin d'appliquer l'attribut au paramètre.

C#

```
/// <summary>
/// Person record type
/// </summary>
/// <param name="FirstName">First Name</param>
/// <param name="LastName">Last Name</param>
/// <remarks>
/// The person type is a positional record containing the
/// properties for the first and last name. Those properties
/// map to the JSON elements "firstName" and "lastName" when
/// serialized or deserialized.
/// </remarks>
public record Person([property: JsonPropertyName("firstName")] string
FirstName,
    [property: JsonPropertyName("lastName")] string LastName);
```

L'exemple précédent montre également comment créer des commentaires de documentation XML pour l'enregistrement. Vous pouvez ajouter la balise `<param>` pour ajouter de la documentation pour les paramètres du constructeur principal.

Si la définition de propriété implémentée automatiquement n'est pas celle que vous souhaitez, vous pouvez définir votre propre propriété du même nom. Par exemple, vous pourriez souhaiter modifier l'accessibilité ou la mutabilité, ou fournir une implémentation pour l'accesseur `get` ou `set`. Si vous déclarez la propriété dans votre source, vous devez l'initialiser à partir du paramètre positionnel de l'enregistrement. Si votre propriété est implémentée automatiquement, vous devez initialiser la propriété. Si vous ajoutez un champ de stockage dans votre source, vous devez initialiser le champ de stockage. Le déconstructeur généré utilise votre définition de propriété. Par exemple,

l'exemple suivant déclare les propriétés `FirstName` et `LastName` d'un enregistrement positionnel `public`, mais limite le paramètre positionnel `Id` à `internal`. Vous pouvez utiliser cette syntaxe pour les enregistrements et les types de struct d'enregistrement.

C#

```
public record Person(string FirstName, string LastName, string Id)
{
    internal string Id { get; init; } = Id;
}

public static void Main()
{
    Person person = new("Nancy", "Davolio", "12345");
    Console.WriteLine(person.FirstName); //output: Nancy

}
```

Un type d'enregistrement n'a pas besoin de déclarer de propriétés positionnelles. Vous pouvez déclarer un enregistrement sans aucune propriété positionnelle, et vous pouvez déclarer d'autres champs et propriétés, comme dans l'exemple suivant :

C#

```
public record Person(string FirstName, string LastName)
{
    public string[] PhoneNumbers { get; init; } = Array.Empty<string>();
};
```

Si vous définissez des propriétés à l'aide de la syntaxe de propriété standard, mais que vous omettez le modificateur d'accès, les propriétés sont implicitement `private`.

Immuabilité

Un *enregistrement positionnel* et un *struct d'enregistrement positionnel en lecture seule* déclarent des propriétés init uniquement. Un *struct d'enregistrement positionnel* déclare des propriétés en lecture-écriture. Vous pouvez remplacer l'une de ces valeurs par défaut, comme indiqué dans la section précédente.

L'immuabilité peut être utile si vous avez besoin d'un type centré sur les données pour être thread-safe ou si vous dépendez d'un code de hachage restant le même dans une table de hachage. Toutefois, l'immuabilité n'est pas appropriée pour tous les scénarios de données. Entity Framework Core, par exemple, ne prend pas en charge la mise à jour avec des types d'entités immuables.

Les propriétés init uniquement, qu'elles soient créées à partir de paramètres positionnels (`record class` et `readonly record struct`) ou en spécifiant `init` des accesseurs, ont *une immuabilité superficielle*. Après l'initialisation, vous ne pouvez pas modifier la valeur des propriétés de type valeur ou la référence des propriétés de type référence. Toutefois, les données auxquelles fait référence une propriété de type référence peuvent être modifiées. L'exemple suivant montre que le contenu d'une propriété immuable de type référence (un tableau dans ce cas) est mutable :

C#

```
public record Person(string FirstName, string LastName, string[]
PhoneNumbers);

public static void Main()
{
    Person person = new("Nancy", "Davolio", new string[1] { "555-1234" });
    Console.WriteLine(person.PhoneNumbers[0]); // output: 555-1234

    person.PhoneNumbers[0] = "555-6789";
    Console.WriteLine(person.PhoneNumbers[0]); // output: 555-6789
}
```

Les fonctionnalités propres aux types d'enregistrements sont implémentées par des méthodes synthétisées par le compilateur, et aucune de ces méthodes ne compromet l'immuabilité en modifiant l'état de l'objet. Sauf indication contraire, les méthodes synthétisées sont générées pour les déclarations `record`, `record struct` et `readonly record struct`.

Égalité des valeurs

Si vous ne remplacez pas les méthodes d'égalité, le type que vous déclarez détermine la façon dont l'égalité est définie :

- Pour les types `class`, deux objets sont égaux s'ils font référence au même objet en mémoire.
- Pour les types `struct`, deux objets sont égaux s'ils sont du même type et stockent les mêmes valeurs.
- Pour les types avec le modificateur `record`, (`record class`, `record struct` et `readonly record struct`), deux objets sont égaux s'ils sont du même type et stockent les mêmes valeurs.

La définition de l'égalité pour un `record struct` est la même que pour un `struct`. La différence est que pour un `struct`, l'implémentation se trouve dans

ValueType.Equals(Object) et s'appuie sur la réflexion. Pour les enregistrements, l'implémentation est synthétisée par le compilateur et utilise les membres de données déclarés.

L'égalité des références est requise pour certains modèles de données. Par exemple, Entity Framework Core dépend de l'égalité des références pour s'assurer qu'il n'utilise qu'une seule instance d'un type d'entité pour ce qui est conceptuellement une seule entité. Pour cette raison, les enregistrements et les structs d'enregistrement ne sont pas appropriés pour une utilisation en tant que types d'entités dans Entity Framework Core.

L'exemple suivant illustre l'égalité de valeur des types d'enregistrements :

C#

```
public record Person(string FirstName, string LastName, string[]
PhoneNumbers);

public static void Main()
{
    var phoneNumbers = new string[2];
    Person person1 = new("Nancy", "Davolio", phoneNumbers);
    Person person2 = new("Nancy", "Davolio", phoneNumbers);
    Console.WriteLine(person1 == person2); // output: True

    person1.PhoneNumbers[0] = "555-1234";
    Console.WriteLine(person1 == person2); // output: True

    Console.WriteLine(ReferenceEquals(person1, person2)); // output: False
}
```

Pour implémenter l'égalité des valeurs, le compilateur synthétise plusieurs méthodes, notamment :

- Un remplacement de Object.Equals(Object). Il s'agit d'une erreur si le remplacement est déclaré explicitement.

  Cette méthode est utilisée comme base pour la Object.Equals(Object, Object) méthode statique lorsque les deux paramètres ne sont pas null.

- `virtual`, ou `sealed`, `Equals(R? other)` où `R` est le type d'enregistrement. Cette méthode implémente IEquatable<T>. Cette méthode peut être déclarée explicitement.

- Si le type d'enregistrement est dérivé d'un type d'enregistrement de base `Base`, `Equals(Base? other)`. Il s'agit d'une erreur si le remplacement est déclaré

explicitement. Si vous fournissez votre propre implémentation de `Equals(R? other)`, fournissez également une implémentation de `GetHashCode`.

- Un remplacement de Object.GetHashCode(). Cette méthode peut être déclarée explicitement.
- Remplace les opérateurs `==` et `!=`. Il s'agit d'une erreur si les opérateurs sont déclarés explicitement.
- Si le type d'enregistrement est dérivé d'un type d'enregistrement de base `protected override Type EqualityContract { get; };`. Cette propriété peut être déclarée explicitement. Pour plus d'informations, consultez Égalité dans les hiérarchies d'héritage.

Si un type d'enregistrement a une méthode qui correspond à la signature d'une méthode synthétisée autorisée à être déclarée explicitement, le compilateur ne synthétise pas cette méthode.

Mutation non destructrice

Si vous devez copier une instance avec certaines modifications, vous pouvez utiliser une expression `with` pour obtenir une *mutation non destructrice*. Une expression `with` crée une nouvelle instance d'enregistrement qui est une copie d'une instance d'enregistrement existante, avec les propriétés et les champs spécifiés modifiés. Vous utilisez la syntaxe d'initialiseur d'objet pour spécifier les valeurs à modifier, comme indiqué dans l'exemple suivant :

C#

```
public record Person(string FirstName, string LastName)
{
    public string[] PhoneNumbers { get; init; }
}

public static void Main()
{
    Person person1 = new("Nancy", "Davolio") { PhoneNumbers = new string[1]
};
    Console.WriteLine(person1);
    // output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers
= System.String[] }

    Person person2 = person1 with { FirstName = "John" };
    Console.WriteLine(person2);
    // output: Person { FirstName = John, LastName = Davolio, PhoneNumbers =
System.String[] }
```

```
    Console.WriteLine(person1 == person2); // output: False

    person2 = person1 with { PhoneNumbers = new string[1] };
    Console.WriteLine(person2);
    // output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers
= System.String[] }
    Console.WriteLine(person1 == person2); // output: False

    person2 = person1 with { };
    Console.WriteLine(person1 == person2); // output: True
}
```

L'expression `with` peut définir des propriétés positionnelles ou des propriétés créées à l'aide de la syntaxe de propriété standard. Les propriétés explicitement déclarées doivent avoir un accesseur `init` ou `set` pour être modifiées dans une expression `with`.

Le résultat d'une expression `with` est une *copie superficielle*, ce qui signifie que pour une propriété de référence, seule la référence à une instance est copiée. L'enregistrement d'origine et la copie se terminent par une référence à la même instance.

Pour implémenter cette fonctionnalité pour les types `record class`, le compilateur synthétise une méthode clone et un constructeur de copie. La méthode clone virtuelle retourne un nouvel enregistrement initialisé par le constructeur de copie. Lorsque vous utilisez une expression `with`, le compilateur crée du code qui appelle la méthode clone, puis définit les propriétés spécifiées dans l'expression `with`.

Si vous avez besoin d'un comportement de copie différent, vous pouvez écrire votre propre constructeur de copie dans un `record class`. Si vous effectuez cette opération, le compilateur n'en synthétisera pas une. Réalisez votre constructeur `private` si l'enregistrement est `sealed`, sinon réalisez-le pour qu'il soit `protected`. Le compilateur ne synthétise pas un constructeur de copie pour les types `record struct`. Vous pouvez en écrire un, mais le compilateur ne génère pas d'appels pour les expressions `with`. Les valeurs de `record struct` sont copiées lors de l'affectation.

Vous ne pouvez pas remplacer la méthode clone et vous ne pouvez pas créer un membre nommé `Clone` dans n'importe quel type d'enregistrement. Le nom réel de la méthode clone est généré par le compilateur.

Mise en forme intégrée pour l'affichage

Les types d'enregistrements ont une méthode ToString générée par le compilateur qui affiche les noms et les valeurs des propriétés publiques et des champs. La méthode `ToString` retourne une chaîne au format suivant :

> <nom du type d'enregistrement> { <nom de la propriété > = <valeur>, <nom de la propriété > = <valeur>, ...}

La chaîne imprimée pour `<value>` est la chaîne retournée par le ToString() pour le type de la propriété. Dans l'exemple suivant, `ChildNames` est un System.Array, où `ToString` retourne `System.String[]` :

```
Person { FirstName = Nancy, LastName = Davolio, ChildNames = System.String[]
}
```

Pour implémenter cette fonctionnalité, dans les types `record class`, le compilateur synthétise une méthode virtuelle `PrintMembers` et un remplacement ToString. Dans les types `record struct`, ce membre est `private`. Le remplacement `ToString` crée un objet StringBuilder avec le nom de type suivi d'un crochet ouvrant. Il appelle `PrintMembers` pour ajouter des noms de propriétés et des valeurs, puis ajoute le crochet fermant. L'exemple suivant montre un code similaire à ce que contient le remplacement synthétisé :

C#

```
public override string ToString()
{
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.Append("Teacher"); // type name
    stringBuilder.Append(" { ");
    if (PrintMembers(stringBuilder))
    {
        stringBuilder.Append(" ");
    }
    stringBuilder.Append("}");
    return stringBuilder.ToString();
}
```

Vous pouvez fournir votre propre implémentation de `PrintMembers` ou le remplacement `ToString`. Des exemples sont fournis ultérieurement dans cet article, dans la section PrintMembersMise en forme des enregistrements dérivés. En C# 10 et dans les versions ultérieures, votre implémentation de `ToString` peut inclure le modificateur `sealed`, ce qui empêche le compilateur de synthétiser une implémentation `ToString` pour tous les enregistrements dérivés. Vous pouvez créer une représentation de chaîne cohérente dans une hiérarchie de types `record`. (Les enregistrements dérivés auront toujours une méthode `PrintMembers` générée pour toutes les propriétés dérivées.)

Héritage

Cette section s'applique uniquement aux types `record class`.

Un enregistrement peut hériter d'un autre enregistrement. Toutefois, un enregistrement ne peut pas hériter d'une classe et une classe ne peut pas hériter d'un enregistrement.

Paramètres positionnels dans les types d'enregistrements dérivés

L'enregistrement dérivé déclare les paramètres positionnels pour tous les paramètres du constructeur principal de l'enregistrement de base. L'enregistrement de base déclare et initialise ces propriétés. L'enregistrement dérivé ne les masque pas, mais crée et initialise uniquement des propriétés pour les paramètres qui ne sont pas déclarés dans son enregistrement de base.

L'exemple suivant illustre l'héritage avec la syntaxe de propriété positionnelle :

C#

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio, Grade = 3 }
}
```

Égalité dans les hiérarchies d'héritage

Cette section s'applique aux types `record class`, mais pas aux types `record struct`. Pour que deux variables d'enregistrement soient égales, le type d'exécution doit être égal. Les types des variables contenantes peuvent être différents. La comparaison de l'égalité héritée est illustrée dans l'exemple de code suivant :

C#

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public static void Main()
```

```
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Person student = new Student("Nancy", "Davolio", 3);
    Console.WriteLine(teacher == student); // output: False

    Student student2 = new Student("Nancy", "Davolio", 3);
    Console.WriteLine(student2 == student); // output: True
}
```

Dans l'exemple, toutes les variables sont déclarées sous la forme `Person`, même lorsque l'instance est un type dérivé de `Student` ou `Teacher`. Les instances ont les mêmes propriétés et les mêmes valeurs de propriété. Mais `student == teacher` retourne `False` bien que les deux soient des variables types `Person`, et `student == student2` retourne `True` bien que l'une soit une variable `Person` et l'autre soit une variable `Student`. Le test d'égalité dépend du type d'exécution de l'objet réel, et non du type déclaré de la variable.

Pour implémenter ce comportement, le compilateur synthétise une propriété `EqualityContract` qui retourne un objet Type qui correspond au type de l'enregistrement. Le `EqualityContract` permet aux méthodes d'égalité de comparer le type d'exécution des objets lorsque l'égalité est vérifiée. Si le type de base d'un enregistrement est `object`, cette propriété est `virtual`. Si le type de base est un autre type d'enregistrement, cette propriété est un remplacement. Si le type d'enregistrement est `sealed`, cette propriété est en fait `sealed`, car le type est `sealed`.

Lorsque le code compare deux instances d'un type dérivé, les méthodes d'égalité synthétisées vérifient l'égalité de tous les membres de données des types de base et dérivés. La méthode synthétisée `GetHashCode` utilise la méthode `GetHashCode` de tous les membres de données déclarés dans le type de base et le type d'enregistrement dérivé. Les membres de données d'un `record` incluent tous les champs déclarés et le champ de stockage synthétisé du compilateur pour toutes les propriétés implémentées automatiquement.

Expressions `with` dans les enregistrements dérivés

Le résultat d'une expression `with` a le même type d'exécution que l'opérande de l'expression. Toutes les propriétés du type d'exécution sont copiées, mais vous pouvez uniquement définir les propriétés du type au moment de la compilation, comme le montre l'exemple suivant :

```
C#
```

```
public record Point(int X, int Y)
{
    public int Zbase { get; set; }
};
public record NamedPoint(string Name, int X, int Y) : Point(X, Y)
{
    public int Zderived { get; set; }
};

public static void Main()
{
    Point p1 = new NamedPoint("A", 1, 2) { Zbase = 3, Zderived = 4 };

    Point p2 = p1 with { X = 5, Y = 6, Zbase = 7 }; // Can't set Name or
Zderived
    Console.WriteLine(p2 is NamedPoint);  // output: True
    Console.WriteLine(p2);
    // output: NamedPoint { X = 5, Y = 6, Zbase = 7, Name = A, Zderived = 4
}

    Point p3 = (NamedPoint)p1 with { Name = "B", X = 5, Y = 6, Zbase = 7,
Zderived = 8 };
    Console.WriteLine(p3);
    // output: NamedPoint { X = 5, Y = 6, Zbase = 7, Name = B, Zderived = 8
}
}
```

Mise en forme `PrintMembers` dans les enregistrements dérivés

La méthode synthétisée `PrintMembers` d'un type d'enregistrement dérivé appelle l'implémentation de base. Il en résulte que toutes les propriétés publiques et les champs des types dérivés et de base sont inclus dans la sortie `ToString`, comme illustré dans l'exemple suivant :

C#

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);

public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio, Grade = 3 }
}
```

Vous pouvez fournir votre propre implémentation de la méthode `PrintMembers`. Dans ce cas, utilisez la signature suivante :

- Pour un enregistrement `sealed` qui dérive de `object` (ne déclare pas d'enregistrement de base) : `private bool PrintMembers(StringBuilder builder)` ;
- Pour un enregistrement `sealed` qui dérive d'un autre enregistrement (notez que le type englobant est `sealed`, donc la méthode est effectivement `sealed`) : `protected override bool PrintMembers(StringBuilder builder)` ;
- Pour un enregistrement qui n'est pas `sealed` et dérive de l'objet : `protected virtual bool PrintMembers(StringBuilder builder);`
- Pour un enregistrement qui n'est pas `sealed` et qui dérive d'un autre enregistrement : `protected override bool PrintMembers(StringBuilder builder);`

Voici un exemple de code qui remplace les méthodes synthétisées `PrintMembers`, l'une pour un type d'enregistrement qui dérive d'un objet et l'autre pour un type d'enregistrement qui dérive d'un autre enregistrement :

C#

```
public abstract record Person(string FirstName, string LastName, string[]
PhoneNumbers)
{
    protected virtual bool PrintMembers(StringBuilder stringBuilder)
    {
        stringBuilder.Append($"FirstName = {FirstName}, LastName =
{LastName}, ");
        stringBuilder.Append($"PhoneNumber1 = {PhoneNumbers[0]},
PhoneNumber2 = {PhoneNumbers[1]}");
        return true;
    }
}

public record Teacher(string FirstName, string LastName, string[]
PhoneNumbers, int Grade)
    : Person(FirstName, LastName, PhoneNumbers)
{
    protected override bool PrintMembers(StringBuilder stringBuilder)
    {
        if (base.PrintMembers(stringBuilder))
        {
            stringBuilder.Append(", ");
        };
        stringBuilder.Append($"Grade = {Grade}");
        return true;
    }
};
```

```
public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", new string[2] { "555-
1234", "555-6789" }, 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio, PhoneNumber1
= 555-1234, PhoneNumber2 = 555-6789, Grade = 3 }
}
```

> ⓘ Notes
>
> En C# 10 et versions ultérieures, le compilateur synthétise `PrintMembers` dans des enregistrements dérivés même lorsqu'un enregistrement de base a scellé la méthode `ToString`. Vous pouvez également créer votre propre implémentation de `PrintMembers`.

Comportement du déconstructeur dans les enregistrements dérivés

La méthode `Deconstruct` d'un enregistrement dérivé retourne les valeurs de toutes les propriétés positionnelles du type au moment de la compilation. Si le type de variable est un enregistrement de base, seules les propriétés de l'enregistrement de base sont déconstruites, sauf si l'objet est casté vers le type dérivé. L'exemple suivant illustre l'appel d'un déconstructeur sur un enregistrement dérivé.

C#

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);

public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    var (firstName, lastName) = teacher; // Doesn't deconstruct Grade
    Console.WriteLine($"{firstName}, {lastName}");// output: Nancy, Davolio

    var (fName, lName, grade) = (Teacher)teacher;
    Console.WriteLine($"{fName}, {lName}, {grade}");// output: Nancy,
Davolio, 3
}
```

Contraintes génériques

Le mot clé `record` est un modificateur pour un type `class` ou `struct`. L'ajout du modificateur `record` inclut le comportement décrit précédemment dans cet article. Il n'existe aucune contrainte générique qui nécessite qu'un type soit un enregistrement. Un `record class` répond à la contrainte `class`. Un `record struct` répond à la contrainte `struct`. Pour plus d'informations, consultez Contraintes sur les paramètres de type.

spécification du langage C#

Pour plus d'informations, consultez la section Classes de la Spécification du langage C#.

Pour plus d'informations sur les fonctionnalités introduites dans C# 9 et les versions ultérieures, consultez ces notes de proposition de fonctionnalités :

- Enregistrements
- Setter init uniquement
- Retours covariants

Voir aussi

- Informations de référence sur C#
- Instructions de conception - Choisir entre la classe et le struct
- Instructions de conception - Conception de struct
- Système de types C#
- with, expression

class (référence C#)

Article • 07/04/2023

Les classes sont déclarées à l'aide du mot clé `class`, comme l'illustre l'exemple suivant :

C#

```
class TestClass
{
    // Methods, properties, fields, events, delegates
    // and nested classes go here.
}
```

Notes

Le langage C# ne permet qu'un seul héritage. Cela signifie qu'une classe peut uniquement hériter de l'implémentation d'une seule classe de base. Toutefois, une classe peut implémenter plusieurs interfaces. Le tableau suivant répertorie des exemples d'héritage de classe et d'implémentation d'interface :

Héritage	Exemple
Aucun	`class ClassA { }`
Unique	`class DerivedClass : BaseClass { }`
Aucun, implémente deux interfaces	`class ImplClass : IFace1, IFace2 { }`
Unique, implémente une seule interface	`class ImplDerivedClass : BaseClass, IFace1 { }`

Les classes que vous déclarez directement dans un espace de noms, non imbriquées dans d'autres classes, peuvent être public ou internal. Par défaut, les classes sont `internal`.

Les membres de classe, notamment les classes imbriquées, peuvent être public, protected internal, protected, internal, private ou private protected. Par défaut, ils sont `private`.

Pour plus d'informations, consultez Modificateurs d'accès.

Vous pouvez déclarer des classes génériques qui ont des paramètres de type. Pour plus d'informations, consultez Classes génériques.

Une classe peut contenir les déclarations des membres suivants :

- Constructeurs
- Constantes
- Fields
- Finaliseurs
- Méthodes
- Propriétés
- Indexeurs
- Opérateurs
- Événements
- Délégués
- Classes
- Interfaces
- Types de structure
- Types d'énumération

Exemple

L'exemple suivant explique comment déclarer des champs, des constructeurs et des méthodes de classe. Il illustre également l'instanciation d'un objet et l'impression des données d'une instance. Dans cet exemple, deux classes sont déclarées. La première, `Child`, contient deux champs privés (`name` et `age`), deux constructeurs publics et une méthode publique. La deuxième, `StringTest`, contient `Main`.

C#

```
class Child
{
    private int age;
    private string name;

    // Default constructor:
    public Child()
    {
        name = "N/A";
    }
```

```
    // Constructor:
    public Child(string name, int age)
    {
        this.name = name;
        this.age = age;
    }

    // Printing method:
    public void PrintChild()
    {
        Console.WriteLine("{0}, {1} years old.", name, age);
    }
}

class StringTest
{
    static void Main()
    {
        // Create objects by using the new operator:
        Child child1 = new Child("Craig", 11);
        Child child2 = new Child("Sally", 10);

        // Create an object using the default constructor:
        Child child3 = new Child();

        // Display results:
        Console.Write("Child #1: ");
        child1.PrintChild();
        Console.Write("Child #2: ");
        child2.PrintChild();
        Console.Write("Child #3: ");
        child3.PrintChild();
    }
}
/* Output:
    Child #1: Craig, 11 years old.
    Child #2: Sally, 10 years old.
    Child #3: N/A, 0 years old.
*/
```

Commentaires

Notez que, dans l'exemple précédant, les champs privés (`name` et `age`) ne sont accessibles que par le biais de la méthode publique de la classe `Child`. Par exemple, vous ne pouvez pas imprimer le nom de l'enfant à partir de la méthode `Main` en utilisant une instruction comme celle-ci :

```
C#
```

```
Console.Write(child1.name);   // Error
```

L'accès aux membres privés de `Child` à partir de `Main` est uniquement possible si `Main` est un membre de la classe.

Du fait que les types déclarés dans une classe sans modificateur d'accès sont par défaut `private`, les données membres dans cet exemple sont toujours `private` si le mot clé est supprimé.

Notez enfin que pour l'objet créé à l'aide du constructeur sans paramètre (`child3`), le champ `age` est initialisé par défaut à la valeur zéro.

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Types référence

interface (Informations de référence sur C#)

Article • 10/05/2023

Une interface définit un contrat. Toute class ou struct qui implémente ce contrat doit fournir une implémentation des membres définis dans l'interface. Une interface peut définir une implémentation par défaut pour les membres. Elle peut également définir des membres static afin de fournir une implémentation unique pour les fonctionnalités courantes. À partir de C# 11, une interface peut définir des membres `static abstract` ou `static virtual` pour déclarer qu'un type d'implémentation doit fournir les membres déclarés. En règle générale, les méthodes `static virtual` déclarent qu'une implémentation doit définir un ensemble d'opérateurs surchargés.

Dans l'exemple suivant, la classe `ImplementationClass` doit implémenter une méthode nommée `SampleMethod` qui n'a aucun paramètre et qui retourne `void`.

Pour plus d'informations et d'exemples, consultez Interfaces.

Exemple d'interface

C#

```
interface ISampleInterface
{
    void SampleMethod();
}

class ImplementationClass : ISampleInterface
{
    // Explicit interface member implementation:
    void ISampleInterface.SampleMethod()
    {
        // Method implementation.
    }

    static void Main()
    {
        // Declare an interface instance.
        ISampleInterface obj = new ImplementationClass();

        // Call the member.
        obj.SampleMethod();
    }
}
```

Une interface peut être membre d'un espace de noms ou d'une classe. Une déclaration d'interface peut contenir des déclarations (signatures sans implémentation) des membres suivants :

- Méthodes
- Propriétés
- Indexeurs
- Événements

Membres d'interface par défaut

Ces déclarations de membre précédentes ne contiennent généralement pas de corps. Un membre d'interface peut déclarer un corps. Les corps membres d'une interface sont l'*implémentation par défaut*. Les membres dotés de corps permettent à l'interface de fournir une implémentation « par défaut » pour les classes et les structs qui ne fournissent pas d'implémentation de remplacement. Une interface peut inclure :

- Constantes
- Opérateurs
- Constructeur statique.
- Types imbriqués
- Champs statiques, méthodes, propriétés, indexeurs et événements
- Déclarations de membre utilisant la syntaxe d'implémentation d'interface explicite.
- Modificateurs d'accès explicites (l'accès par défaut est public).

Membres virtuels et abstraits statiques

À partir de C# 11, une interface peut déclarer des membres `static abstract` et `static virtual` pour tous les types de membres, à l'exception des champs. Les interfaces peuvent déclarer que les types d'implémentation doivent définir des opérateurs ou d'autres membres statiques. Cette fonctionnalité permet aux algorithmes génériques de spécifier le comportement de type nombre. Vous pouvez voir des exemples dans les types numériques dans le runtime .NET, par exemple System.Numerics.INumber<TSelf>. Ces interfaces définissent des opérateurs mathématiques courants implémentés par de nombreux types numériques. Le compilateur doit résoudre les appels aux méthodes `static virtual` et `static abstract` au moment de la compilation. Les méthodes `static virtual` et `static abstract` déclarées dans les interfaces n'ont pas de mécanisme de répartition d'exécution analogue aux méthodes `virtual` ou `abstract` déclarées dans les classes. Au lieu de cela, le compilateur utilise des informations de type disponibles au moment de la compilation. Par conséquent, les méthodes `static virtual` sont presque

exclusivement déclarées dans les interfaces génériques. En outre, la plupart des interfaces qui déclarent les méthodes `static virtual` ou `static abstract` notifient que l'un des paramètres de types doit implémenter l'interface déclarée. Par exemple, l'interface `INumber<T>` déclare que `T` doit implémenter `INumber<T>`. Le compilateur utilise l'argument de type pour résoudre les appels aux méthodes et opérateurs déclarés dans la déclaration d'interface. Par exemple, le type `int` implémente `INumber<int>`. Lorsque le paramètre de type `T` désigne l'argument de type `int`, les membres `static` déclarés dans `int` sont appelés. Sinon, si `double` est l'argument de type, les membres `static` déclarés dans le type `double` sont appelés.

> **ⓘ Important**
>
> La répartition des méthodes `static abstract` et `static virtual` déclarées dans les interfaces est résolue à l'aide du type de temps de compilation d'une expression. Si le type d'exécution d'une expression est dérivé d'un type de temps de compilation différent, les méthodes statiques sur le type de base (heure de compilation) sont appelées.

Vous pouvez essayer cette fonctionnalité en travaillant avec le tutoriel sur les membres abstraits statiques dans les interfaces.

Héritage de l'interface

Les interfaces peuvent ne pas contenir d'état d'instance. Bien que les champs statiques soient désormais autorisés, les champs d'instance ne sont pas autorisés dans les interfaces. Les propriétés automatiques d'instance ne sont pas prises en charge dans les interfaces, car elles déclareraient implicitement un champ masqué. Cette règle a un effet subtil sur les déclarations de propriété. Dans une déclaration d'interface, le code suivant ne déclare pas de propriété implémentée automatiquement comme il le fait dans une `class` ou une `struct`. À la place, il déclare une propriété qui n'a pas d'implémentation par défaut, mais doit être implémentée dans n'importe quel type qui implémente l'interface :

C#

```
public interface INamed
{
  public string Name {get; set;}
}
```

Une interface peut hériter d'une ou de plusieurs interfaces de base. Lorsqu'une interface remplace une méthode implémentée dans une interface de base, elle doit utiliser la syntaxe d'implémentation d'interface explicite.

Lorsqu'une liste de types de base contient une classe de base et des interfaces, la classe de base doit figurer en premier dans la liste.

Une classe qui implémente une interface peut implémenter explicitement les membres de cette interface. Un membre implémenté explicitement n'est pas accessible via une instance de classe, mais uniquement via une instance de l'interface. En outre, les membres de l'interface par défaut sont accessibles uniquement via une instance de l'interface.

Pour plus d'informations sur l'implémentation d'interface explicite, consultez Implémentation d'interface explicite.

Exemple d'implémentation d'interface

L'exemple suivant montre une implémentation d'interface. Dans cet exemple, l'interface contient la déclaration de propriété, et la classe contient l'implémentation. Toutes les instances d'une classe qui implémentent `IPoint` ont les propriétés entières `x` et `y`.

C#

```
interface IPoint
{
    // Property signatures:
    int X { get; set; }

    int Y { get; set; }

    double Distance { get; }
}

class Point : IPoint
{
    // Constructor:
    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }

    // Property implementation:
    public int X { get; set; }

    public int Y { get; set; }
```

```
    // Property implementation
    public double Distance =>
       Math.Sqrt(X * X + Y * Y);
}

class MainClass
{
    static void PrintPoint(IPoint p)
    {
        Console.WriteLine("x={0}, y={1}", p.X, p.Y);
    }

    static void Main()
    {
        IPoint p = new Point(2, 3);
        Console.Write("My Point: ");
        PrintPoint(p);
    }
}
// Output: My Point: x=2, y=3
```

spécification du langage C#

Pour plus d'informations, consultez la section Interfaces de la spécification du langage C#, la spécification de fonctionnalité pour C# 8 - Membres d'interface par défaut et la spécification de fonctionnalité pour C# 11 - Membres abstraits statiques dans les interfaces

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Types référence
- Interfaces
- Utilisation de propriétés
- Utilisation d'indexeurs

Types référence nullables (Référence C#)

Article • 10/05/2023

> **Notes**
>
> Cet article traite des types référence nullables. Vous pouvez également déclarer des **types valeur nullables**.

Les types référence nullables sont disponibles dans du code qui a opté pour un *contexte prenant en charge les types nullables*. Les types référence nullables, les avertissements d'analyse statique null et l'opérateur null-forgiving sont des fonctionnalités de langage facultatives. Ils sont tous désactivés par défaut. Un *contexte nullable* est contrôlé au niveau du projet à l'aide des paramètres de build ou dans le code à l'aide de pragmas.

> **Important**
>
> Tous les modèles de projet commençant par .NET 6 (C# 10) activent le *contexte nullable* pour le projet. Les projets créés avec des modèles antérieurs n'incluent pas cet élément, et ces fonctionnalités sont désactivées, sauf si vous les activez dans le fichier projet ou si vous utilisez des pragmas.

Dans un contexte prenant en charge les types nullables :

- Une variable d'un type référence `T` doit être initialisée avec une valeur non-null et peut ne jamais être affectée d'une valeur pouvant être `null`.
- Une variable d'un type référence `T?` peut être initialisée avec `null` ou se voir affecter `null`, mais doit être comparée à `null` avant d'être déréférencée.
- Une variable `m` de type `T?` est considérée comme non-null lorsque vous appliquez l'opérateur null-forgiving, comme dans `m!`.

Les distinctions entre un type référence non nullable `T` et un type référence nullable `T?` sont appliquées par l'interprétation des règles précédentes par le compilateur. Une variable de type `T` et une variable de type `T?` sont représentées par le même type .NET. L'exemple suivant déclare une chaîne non nullable et une chaîne nullable, puis utilise l'opérateur null-forgiving pour affecter une valeur à une chaîne non nullable :

C#

```
string notNull = "Hello";
string? nullable = default;
```

```
notNull = nullable!; // null forgiveness
```

Les variables `notNull` et `nullable` sont toutes les deux représentées par le type String. Étant donné que les types non nullables et nullables sont tous les deux stockés comme un même type, il existe plusieurs emplacements où l'utilisation d'un type référence nullable n'est pas autorisée. En général, un type référence nullable ne peut pas être utilisé comme classe de base ou interface implémentée. Un type référence nullable ne peut pas être utilisé dans une expression de création d'objet ni de test de type. Un type référence nullable ne peut pas être le type d'une expression d'accès au membre. Les exemples suivants illustrent ces constructions :

C#

```
public MyClass : System.Object? // not allowed
{
}

var nullEmpty = System.String?.Empty; // Not allowed
var maybeObject = new object?(); // Not allowed
try
{
    if (thing is string? nullableString) // not allowed
        Console.WriteLine(nullableString);
} catch (Exception? e) // Not Allowed
{
    Console.WriteLine("error");
}
```

Références nullables et analyse statique

Les exemples de la section précédente illustrent la nature des types référence nullables. Les types référence nullables ne sont pas de nouveaux types de classes, mais plutôt des annotations sur des types référence existants. Le compilateur utilise ces annotations pour vous aider à trouver des erreurs de référence null potentielles dans votre code. Il n'existe aucune différence de runtime entre un type référence non nullable et un type référence nullable. Le compilateur n'ajoute aucune vérification de runtime pour les types référence non nullables. Les avantages se trouvent dans l'analyse au moment de la compilation. Le compilateur génère des avertissements qui vous aident à rechercher et à corriger les erreurs null potentielles dans votre code. Vous déclarez votre intention et le compilateur vous avertit lorsque votre code contrevient à cette intention.

Dans un contexte prenant en charge les types nullables, le compilateur effectue une analyse statique sur des variables de n'importe quel type référence, nullable et non nullable. Le compilateur effectue le suivi de l'*état null* de chaque variable de référence

comme *not-null* ou *maybe-null*. L'état par défaut d'une référence non nullable est *not-null*. L'état par défaut d'une référence nullable est *maybe-null*.

Les types référence non nullables doivent toujours pouvoir être déréférencés en toute sécurité, car leur *état null* est *not-null*. Pour appliquer cette règle, le compilateur émet des avertissements si un type référence non nullable n'est pas initialisé en valeur non-null. Les variables locales doivent être affectées à l'endroit où elles sont déclarées. Chaque champ doit se voir attribuer une valeur *not-null*, dans un initialiseur de champ ou dans chaque constructeur. Le compilateur émet des avertissements quand une référence non nullable est affectée à une référence dont l'état est *maybe-null*. En règle générale, une référence non nullable est *not-null* et aucun avertissement n'est émis quand ces variables sont déréférencées.

> ⓘ **Notes**
>
> Si vous affectez une expression *maybe-null* à un type de référence non nullable, le compilateur génère un avertissement. Le compilateur génère ensuite des avertissements pour cette variable jusqu'à ce qu'elle soit affectée à une expression *not-null*.

Les types référence nullables peuvent être initialisés ou affectés à `null`. Par conséquent, l'analyse statique doit déterminer qu'une variable est *not-null* avant d'être déréférencée. S'il est déterminé qu'une référence nullable est *maybe-null*, l'affectation à une variable de référence non nullable génère un avertissement du compilateur. La classe suivante montre des exemples de ces avertissements :

C#

```
public class ProductDescription
{
    private string shortDescription;
    private string? detailedDescription;

    public ProductDescription() // Warning! shortDescription not
initialized.
    {
    }

    public ProductDescription(string productDescription) =>
        this.shortDescription = productDescription;

    public void SetDescriptions(string productDescription, string?
details=null)
    {
        shortDescription = productDescription;
        detailedDescription = details;
```

```
    }

    public string GetDescription()
    {
        if (detailedDescription.Length == 0) // Warning! dereference
possible null
        {
            return shortDescription;
        }
        else
        {
            return $"{shortDescription}\n{detailedDescription}";
        }
    }

    public string FullDescription()
    {
        if (detailedDescription == null)
        {
            return shortDescription;
        }
        else if (detailedDescription.Length > 0) // OK, detailedDescription
can't be null.
        {
            return $"{shortDescription}\n{detailedDescription}";
        }
        return shortDescription;
    }
}
```

L'extrait de code suivant montre où le compilateur émet des avertissements lors de l'utilisation de cette classe :

C#

```
string shortDescription = default; // Warning! non-nullable set to null;
var product = new ProductDescription(shortDescription); // Warning! static
analysis knows shortDescription maybe null.

string description = "widget";
var item = new ProductDescription(description);

item.SetDescriptions(description, "These widgets will do everything.");
```

Les exemples précédents montrent comment l'analyse statique du compilateur détermine l'*état null* des variables de référence. Le compilateur applique des règles de langage pour les contrôles de valeur null et les affectations afin de guider son analyse. Le compilateur ne peut pas faire d'hypothèses sur la sémantique des méthodes ni des propriétés. Si vous appelez des méthodes qui effectuent des contrôles de valeur null, le compilateur ne peut pas savoir que ces méthodes affectent l'*état null* d'une variable. Il

existe des attributs que vous pouvez ajouter à vos API pour informer le compilateur sur la sémantique des arguments et des valeurs de retour. Ces attributs ont été appliqués à de nombreuses API courantes dans les bibliothèques .NET Core. Par exemple, IsNullOrEmpty a été mis à jour et le compilateur interprète correctement cette méthode comme un contrôle de valeur null. Pour plus d'informations sur les attributs qui s'appliquent à l'analyse statique d'*état null*, consultez l'article sur les attributs nullables.

Définition du contexte nullable

Il existe deux façons de contrôler le contexte nullable. Au niveau du projet, vous pouvez ajouter le paramètre de projet `<Nullable>enable</Nullable>`. Dans un fichier source C# individuel, vous pouvez ajouter le pragma `#nullable enable` pour activer le contexte nullable. Consultez l'article sur la définition d'une stratégie nullable. Avant .NET 6, les nouveaux projets utilisent la valeur par défaut, `<Nullable>disable</Nullable>`. À partir de .NET 6, les nouveaux projets incluent l'élément `<Nullable>enable</Nullable>` dans le fichier projet.

spécification du langage C#

Pour plus d'informations, consultez les propositions suivantes pour la spécification du langage C# :

- Types références Nullables
- Brouillon de spécification des types référence nullables

Voir aussi

- Informations de référence sur C#
- Types valeur Nullable

void (référence C#)

Article • 07/04/2023

Vous utilisez `void` comme type de retour d'une méthode (ou d'une fonction locale) pour spécifier que la méthode ne retourne pas de valeur.

C#

```
public static void Display(IEnumerable<int> numbers)
{
    if (numbers is null)
    {
        return;
    }

    Console.WriteLine(string.Join(" ", numbers));
}
```

Vous pouvez également utiliser `void` comme type de référence pour déclarer un pointeur vers un type inconnu. Pour plus d'informations, consultez Types pointeur.

Vous ne pouvez pas utiliser `void` comme type d'une variable.

Voir aussi

- Informations de référence sur C#
- System.Void

Instructions de déclaration

Article • 21/06/2023

Une instruction de déclaration déclare une nouvelle variable locale, constante locale ou variable de référence locale. Pour déclarer une variable locale, spécifiez son type et indiquez son nom. Vous pouvez déclarer plusieurs variables du même type dans une même instruction, comme le montre l'exemple suivant :

```
C#

string greeting;
int a, b, c;
List<double> xs;
```

Dans une instruction de déclaration, vous pouvez également initialiser une variable avec sa valeur initiale :

```
C#

string greeting = "Hello";
int a = 3, b = 2, c = a + b;
List<double> xs = new();
```

Les exemples précédents spécifient explicitement le type d'une variable. Vous pouvez également laisser le compilateur déduire le type d'une variable à partir de son expression d'initialisation. Pour ce faire, utilisez le mot clé `var` au lieu du nom d'un type. Pour plus d'informations, consultez la section Variables locales implicitement typées.

Pour déclarer une constante locale, utilisez le const mot clé, comme le montre l'exemple suivant :

```
C#

const string Greeting = "Hello";
const double MinLimit = -10.0, MaxLimit = -MinLimit;
```

Lorsque vous déclarez une constante locale, vous devez également l'initialiser.

Pour plus d'informations sur les variables de référence locales, consultez la section Variables de référence.

Variables locales implicitement typées

Lorsque vous déclarez une variable locale, vous pouvez laisser le compilateur déduire le type de la variable à partir de l'expression d'initialisation. Pour ce faire, utilisez le mot clé `var` au lieu du nom d'un type :

C#

```
var greeting = "Hello";
Console.WriteLine(greeting.GetType());  // output: System.String

var a = 32;
Console.WriteLine(a.GetType());  // output: System.Int32

var xs = new List<double>();
Console.WriteLine(xs.GetType());  // output:
System.Collections.Generic.List`1[System.Double]
```

Comme le montre l'exemple précédent, les variables locales implicitement typées sont fortement typées.

> ⓘ Notes
>
> Lorsque vous utilisez `var` dans le **contexte pouvant accepter la valeur Null** activé et que le type d'une expression d'initialisation est un type référence, le compilateur déduit toujours un type de référence **pouvant accepter la valeur Null** même si le type d'une expression d'initialisation n'est pas compatible avec la valeur Null.

Une utilisation courante de `var` est l'expression d'invocation d'un constructeur. L'utilisation de `var` vous évite de répéter un nom de type dans une déclaration de variable et une instanciation d'objet, comme le montre l'exemple suivant :

C#

```
var xs = new List<int>();
```

À compter de C# 9.0, vous pouvez utiliser une expression new typée cible en guise d'alternative :

C#

```
List<int> xs = new();
List<int>? ys = new();
```

Lorsque vous utilisez des types anonymes, vous devez utiliser des variables locales implicitement typées. L'exemple suivant montre une expression de requête qui utilise un type anonyme pour contenir le nom et le numéro de téléphone d'un client :

```
C#
```

```
var fromPhoenix = from cust in customers
                  where cust.City == "Phoenix"
                  select new { cust.Name, cust.Phone };

foreach (var customer in fromPhoenix)
{
    Console.WriteLine($"Name={customer.Name}, Phone={customer.Phone}");
}
```

Dans l'exemple précédent, vous ne pouvez pas spécifier explicitement le type de la variable `fromPhoenix`. Le type est IEnumerable<T>, mais, dans ce cas, `T` est un type anonyme et vous ne pouvez pas fournir son nom. C'est pourquoi vous devez utiliser `var`. Pour la même raison, vous devez utiliser `var` lorsque vous déclarez la variable d'itération `customer` dans l'instruction `foreach`.

Pour plus d'informations sur les variables locales implicitement typées, consultez Variables locales implicitement typées.

Dans les critères spéciaux, le mot clé `var` est utilisé dans un modèle var.

Variables de référence

Lorsque vous déclarez une variable locale et que vous ajoutez le mot clé `ref` avant le type de la variable, vous déclarez une *variable de référence*, ou une variable locale `ref` :

```
C#
```

```
ref int alias = ref variable;
```

Une variable de référence est une variable qui fait référence à une autre variable, appelée « *référent* ». Autrement dit, une variable de référence est un *alias* à son référent. Lorsque vous affectez une valeur à une variable de référence, cette valeur est assignée au référent. Lorsque vous lisez la valeur d'une variable de référence, la valeur du référent est retournée. L'exemple suivant illustre ce comportement :

```
C#
```

```
int a = 1;
ref int alias = ref a;
Console.WriteLine($"(a, alias) is ({a}, {alias})");  // output: (a, alias)
is (1, 1)

a = 2;
Console.WriteLine($"(a, alias) is ({a}, {alias})");  // output: (a, alias)
is (2, 2)

alias = 3;
Console.WriteLine($"(a, alias) is ({a}, {alias})");  // output: (a, alias)
is (3, 3)
```

Utilisez l'opérateur d'assignation ref `= ref` pour changer le référent d'une variable de référence, comme le montre l'exemple suivant :

C#

```
void Display(int[] s) => Console.WriteLine(string.Join(" ", s));

int[] xs = { 0, 0, 0 };
Display(xs);

ref int element = ref xs[0];
element = 1;
Display(xs);

element = ref xs[^1];
element = 3;
Display(xs);
// Output:
// 0 0 0
// 1 0 0
// 1 0 3
```

Dans l'exemple précédent, la variable de référence `element` est initialisée en tant qu'alias au premier élément de tableau. Ensuite, elle est réassignée par référence (`ref`) pour faire référence au dernier élément de tableau.

Vous pouvez définir une variable locale `ref readonly`. Vous ne pouvez pas assigner une valeur à une variable `ref readonly`. Toutefois, vous pouvez réaffecter par référence (`ref`) une telle variable de référence, comme le montre l'exemple suivant :

C#

```
int[] xs = { 1, 2, 3 };

ref readonly int element = ref xs[0];
```

```
// element = 100;  error CS0131: The left-hand side of an assignment must be
a variable, property or indexer
Console.WriteLine(element);  // output: 1

element = ref xs[^1];
Console.WriteLine(element);  // output: 3
```

Vous pouvez affecter un retour de référence à une variable de référence, comme le montre l'exemple suivant :

C#

```
using System;

public class NumberStore
{
    private readonly int[] numbers = { 1, 30, 7, 1557, 381, 63, 1027, 2550,
511, 1023 };

    public ref int GetReferenceToMax()
    {
        ref int max = ref numbers[0];
        for (int i = 1; i < numbers.Length; i++)
        {
            if (numbers[i] > max)
            {
                max = ref numbers[i];
            }
        }
        return ref max;
    }

    public override string ToString() => string.Join(" ", numbers);
}

public static class ReferenceReturnExample
{
    public static void Run()
    {
        var store = new NumberStore();
        Console.WriteLine($"Original sequence: {store.ToString()}");

        ref int max = ref store.GetReferenceToMax();
        max = 0;
        Console.WriteLine($"Updated sequence:  {store.ToString()}");
        // Output:
        // Original sequence: 1 30 7 1557 381 63 1027 2550 511 1023
        // Updated sequence:  1 30 7 1557 381 63 1027 0 511 1023
    }
}
```

Dans l'exemple précédent, la méthode `GetReferenceToMax` est une méthode *returns-by-ref*. Elle ne retourne pas la valeur maximale elle-même, mais un retour de référence qui est un alias à l'élément de tableau qui contient la valeur maximale. La méthode `Run` affecte un retour de référence à la variable de référence `max`. Ensuite, en affectant une valeur à `max`, elle met à jour le stockage interne de l'instance `store`. Vous pouvez également définir une méthode `ref readonly`. Les appelants d'une méthode `ref readonly` ne peuvent pas affecter une valeur à son retour de référence.

La variable d'itération de l'instruction `foreach` peut être une variable de référence. Pour plus d'informations, consultez la section foreach de l'article Instructions d'itération.

Dans les scénarios critiques pour les performances, l'utilisation de variables et de retours de référence peut améliorer les performances en évitant les opérations de copie potentiellement coûteuses.

Le compilateur garantit qu'une variable de référence ne vit pas plus longtemps que son référent et reste valide pendant toute sa durée de vie. Pour plus d'informations, consultez la section Contextes de sécurité de référence de la Spécification du langage C#.

Pour plus d'informations sur les champs `ref`, consultez la section Champs ref de l'article Types de structure ref.

scoped ref

Le mot clé contextuel `scoped` limite la durée de vie d'une valeur. Le modificateur `scoped` limite la durée de vie de *ref-safe-to-escape* ou *safe-to-escape*, respectivement, à la méthode actuelle. En fait, l'ajout du modificateur `scoped` déclare que votre code n'allongera pas la durée de vie de la variable.

Vous pouvez appliquer `scoped` à un paramètre ou à une variable locale. Le modificateur `scoped` peut être appliqué aux paramètres et aux variables locales quand le type est un ref struct. Sinon, le modificateur `scoped` peut être appliqué uniquement aux variables de référence locales. Cela inclut les variables locales déclarées avec le modificateur `ref` et les paramètres déclarés avec les modificateurs ou `in`, `ref` ou `out`.

Le modificateur `scoped` est implicitement ajouté à `this` dans les méthodes déclarées dans un `struct`, des paramètres `out` et des paramètres `ref` quand le type est un `ref struct`.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Instructions de déclaration
- Variables et retours de référence

Pour plus d'informations sur le modificateur `scoped`, consultez la note de proposition Améliorations des structures de bas niveau.

Voir aussi

- Informations de référence sur C#
- Initialiseurs d'objets et de collections
- ref, mot clé
- Réduire les allocations de mémoire à l'aide de nouvelles fonctionnalités C#
- Préférences « var » (règles de style IDE0007 et IDE0008)

Types intégrés (référence C#)

Article • 10/05/2023

Le tableau suivant répertorie les types valeur C# intégrés par catégorie :

Type/mot clé C#	Type .NET
bool	System.Boolean
byte	System.Byte
sbyte	System.SByte
char	System.Char
decimal	System.Decimal
double	System.Double
float	System.Single
int	System.Int32
uint	System.UInt32
nint	System.IntPtr
nuint	System.UIntPtr
long	System.Int64
ulong	System.UInt64
short	System.Int16
ushort	System.UInt16

Le tableau suivant répertorie les types référence C# intégrés par catégorie :

Type/mot clé C#	Type .NET
object	System.Object
string	System.String
dynamic	System.Object

Dans le tableau précédent, chaque mot clé de type C# de la colonne gauche (hormis dynamique) est un alias pour le type .NET correspondant. Ils sont interchangeables. Par

exemple, les déclarations suivantes déclarent des variables du même type :

```
C#
```

```
int a = 123;
System.Int32 b = 123;
```

Le mot clé void représente l'absence d'un type. Vous l'utilisez comme type de retour d'une méthode qui ne retourne pas de valeur.

Voir aussi

- Utilisez des mots clés de langage plutôt que des noms de type infrastructure (règle de style IDE0049)
- Informations de référence sur C#
- Valeurs par défaut des types C#

Types non managés (référence C#)

Article • 23/06/2023

Un type est un **type non managé** s'il s'agit de l'un des types suivants :

- `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `nint`, `nuint`, `char`, `float`, `double`, `decimal` ou `bool`
- Tout type enum
- Tout type pointer
- Tout type struct défini par l'utilisateur qui contient des champs de types non managés uniquement.

Vous pouvez utiliser la contrainte unmanaged pour spécifier qu'un paramètre de type est un type non managé non pointeur, non nullable.

Un type de struct *construit* qui contient des champs de types non managés uniquement n'est pas géré, comme le montre l'exemple suivant :

C#

```
using System;

public struct Coords<T>
{
    public T X;
    public T Y;
}

public class UnmanagedTypes
{
    public static void Main()
    {
        DisplaySize<Coords<int>>();
        DisplaySize<Coords<double>>();
    }

    private unsafe static void DisplaySize<T>() where T : unmanaged
    {
        Console.WriteLine($"{typeof(T)} is unmanaged and its size is
{sizeof(T)} bytes");
    }
}
// Output:
// Coords`1[System.Int32] is unmanaged and its size is 8 bytes
// Coords`1[System.Double] is unmanaged and its size is 16 bytes
```

Un struct générique peut être la source des types construits non managés et managés. L'exemple précédent définit un struct `Coords<T>` générique et présente les exemples de types construits non managés. L'exemple d'un type managé est `Coords<object>`. Il est géré, car il possède les champs du type `object`, qui est géré. Si vous souhaitez que *tous* les types construits soient des types non managés, utilisez la contrainte `unmanaged` dans la définition d'un struct générique :

C#

```
public struct Coords<T> where T : unmanaged
{
    public T X;
    public T Y;
}
```

spécification du langage C#

Pour plus d'informations, consultez la section Types de pointeur de la Spécification du langage C#.

Voir aussi

- Informations de référence sur C#
- Types de pointeur
- Types liés à la mémoire et à l'étendue
- Opérateur sizeof
- stackalloc

Valeurs par défaut des types C# (référence C#)

Article • 07/04/2023

Le tableau suivant présente les valeurs par défaut des types C# :

Type	Valeur par défaut
Tout type référence	`null`
Tout type numérique intégral intégré	0 (zéro)
Tout type numérique à virgule flottante intégré	0 (zéro)
bool	`false`
char	`'\0'` (U+0000)
enum	Valeur produite par l'expression `(E)0`, où `E` est l'identificateur de l'enum.
struct	Valeur produite en affectant à tous les champs de type valeur leur valeur par défaut et à tous les champs de type référence la valeur `null`.
Tout type valeur Nullable	Instance pour laquelle la propriété HasValue a la valeur `false` et la propriété Value n'est pas définie. Cette valeur par défaut est également connue sous le nom de valeur *null* du type valeur pouvant accepter la valeur Null.

Expressions de valeur par défaut

Utilisez l'opérateur default pour produire la valeur par défaut d'un type, comme illustré dans l'exemple suivant :

C#

```
int a = default(int);
```

Vous pouvez utiliser le littéral default pour initialiser une variable avec la valeur par défaut de son type :

```
C#

int a = default;
```

Constructeur sans paramètre d'un type valeur

Pour un type valeur, le constructeur sans paramètre *implicite* produit également la valeur par défaut du type, comme le montre l'exemple suivant :

```
C#

var n = new System.Numerics.Complex();
Console.WriteLine(n);  // output: (0, 0)
```

Au moment de l'exécution, si l'instance System.Type représente un type valeur, vous pouvez utiliser la méthode Activator.CreateInstance(Type) pour appeler le constructeur sans paramètre afin d'obtenir la valeur par défaut du type.

> ⓘ **Notes**
>
> Dans C# 10 et versions ultérieures, un **type de structure** (qui est un type valeur) peut avoir un **constructeur sans paramètre explicite** qui peut produire une valeur autre que par défaut du type. Par conséquent, nous vous recommandons d'utiliser l'opérateur `default` ou le littéral `default` pour produire la valeur par défaut d'un type.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Valeurs par défaut
- Constructeurs par défaut
- C# 10 - Constructeurs de struct sans paramètre
- C# 11 - Structs par défaut automatiques

Voir aussi

- Informations de référence sur C#
- Constructeurs

Mots clés C#

Article • 25/02/2023

Les mots clés sont des identificateurs réservés prédéfinis, qui ont des significations spécifiques pour le compilateur. Ils ne peuvent pas être utilisés comme identificateurs dans votre programme, sauf s'ils incluent `@` comme préfixe. Par exemple, `@if` est un identificateur valide, mais pas `if`, car `if` est un mot clé.

Le premier tableau de cette rubrique répertorie les mots clés qui sont des identificateurs réservés quelle que soit la partie d'un programme C#. Le deuxième tableau de cette rubrique liste les mots clés contextuels en C#. Les mots clés contextuels ont une signification spéciale dans un contexte de programme limité ; ils peuvent être utilisés comme identificateurs en dehors de ce contexte. En règle générale, lorsque de nouveaux mots clés sont ajoutés au langage C#, ils sont ajoutés comme mots clés contextuels afin d'éviter l'interruption des programmes écrits dans les versions antérieures.

abstract
as
base
bool
break
byte
case
catch
char
checked
class
const
continue
decimal
default
delegate
do
double
else
enum

event
explicit

extern
false
finally
fixed
float
for
foreach
goto
if
implicite
in
int
interface
internal
is
lock
long

namespace
nouveau
null
object
operator
out
override
params
private
protected
public
readonly
ref
return
sbyte
sealed
short
sizeof
stackalloc

static
string
struct

switch
this
throw
true
try
typeof
uint
ulong
non cochée
unsafe
ushort
using
virtual
void
volatile
while

Mots clés contextuels

Un mot clé contextuel sert à donner une signification spécifique dans le code, sans pour autant être un mot réservé en C#. Certains mots clés contextuels, tels que `partial` et `where`, ont des significations spéciales dans deux contextes ou plus.

add
and
alias
ascending
args
async
await
by
descending
dynamic
equals
from

get
global
groupe
init

into
join
let
managée (convention d'appel de pointeur de fonction)
nameof
nint
not

notnull
nuint
actif
or
orderby
partial (type)
partial (méthode)
enregistrement
remove
select

set
non managée (convention d'appel de pointeur de fonction)
non managée (contrainte de type générique)
value
var
when (condition de filtre)
where (contrainte de type générique)
where (clause de requête)
with
yield

Voir aussi

- Informations de référence sur C#

Modificateurs d’accès (référence C#)

Article • 07/04/2023

Les modificateurs d’accès sont des mots clés utilisés pour spécifier l’accessibilité déclarée d’un membre ou d’un type. Cette section présente les cinq modificateurs d’accès :

- `public`
- `protected`
- `internal`
- `private`
- `file`

Les modificateurs d’accès permettent de spécifier les sept niveaux d’accessibilité suivants :

- public: l’accès n’est pas restreint.
- protected : L’accès est limité à la classe conteneur ou aux types dérivés de la classe conteneur.
- internal : L’accès est limité à l’assembly actuel.
- protected internal : l’accès est limité à l’assembly actuel ou aux types dérivés de la classe conteneur.
- private : L’accès est limité au type conteneur.
- private protected : l’accès est limité à la classe conteneur ou aux types dérivés de la classe conteneur dans l’assembly actuel.
- file: le type déclaré est visible uniquement dans le fichier source actuel. Les types de fichiers délimités sont généralement utilisés pour les générateurs de sources.

Cette section introduit également les concepts suivants :

- Niveaux d’accessibilité : utilisation des quatre modificateurs d’accès pour déclarer six niveaux d’accessibilité.
- Domaine d’accessibilité : spécifie où, dans les sections du programme, un membre peut être référencé.
- Limitations sur l’utilisation des niveaux d’accessibilité : résumé des restrictions sur l’utilisation des niveaux d’accessibilité déclarés.

Voir aussi

- Ajouter des modificateurs d’accessibilité (règle de style IDE0040)
- Référence C#

- Guide de programmation C#
- Mots clés C#
- Modificateurs d’accès
- Mots clés d'accès
- Modificateurs

Niveaux d'accessibilité (référence C#)

Article • 17/08/2023

Utilisez les modificateurs d'accès `public`, `protected`, `internal` ou `private` pour spécifier l'un des niveaux d'accessibilité déclarés ci-dessous pour les membres.

Accessibilité déclarée	Signification
public	L'accès n'est pas limité.
protected	L'accès est limité à la classe conteneur ou aux types dérivés de la classe conteneur.
internal	L'accès est limité à l'assembly actuel.
protected internal	L'accès est limité à l'assembly actuel ou aux types dérivés de la classe conteneur.
private	L'accès est limité au type conteneur.
private protected	L'accès est limité à la classe conteneur ou aux types dérivés de la classe conteneur dans l'assembly actuel.

Vous ne pouvez spécifier qu'un seul modificateur d'accès pour un membre ou un type, sauf si vous utilisez les combinaisons `protected internal` ou `private protected`.

Les modificateurs d'accès ne sont pas autorisés sur les espaces de noms. Les espaces de noms ne présentent aucune limitation d'accès.

Selon le contexte dans lequel une déclaration de membre est effectuée, seules certaines accessibilités déclarées sont autorisées. Si aucun modificateur d'accès n'est spécifié dans une déclaration de membre, une accessibilité par défaut est utilisée.

Les types de premier niveau, qui ne sont pas imbriqués dans d'autres types, peuvent uniquement avoir une accessibilité `internal` ou `public`. L'accessibilité par défaut de ces types est `internal`.

Les types imbriqués, qui sont membres d'autres types, peuvent avoir les accessibilités déclarées indiquées dans le tableau suivant.

Membres de	Accessibilité par défaut du membre	Accessibilité déclarée du membre autorisée
`enum`	`public`	None

Membres de	Accessibilité par défaut du membre	Accessibilité déclarée du membre autorisée

`class`	`private`	`public`
		`protected`
		`internal`
		`private`
		`protected internal`
		`private protected`

`interface`	`public`	`public`
		`protected`
		`internal`
		`private` *
		`protected internal`
		`private protected`

`struct`	`private`	`public`
		`internal`
		`private`

* Un membre `interface` avec l'accessibilité `private` doit avoir une implémentation par défaut.

> ⓘ **Notes**
>
> Si une classe ou un struct est modifié avec le modificateur de mot clé **record**, les mêmes modificateurs d'accès sont autorisés.
> De plus, avec le modificateur **record** l'accessibilité des membres par défaut est toujours `private` pour la classe et le struct.

L'accessibilité d'un type imbriqué dépend de son domaine d'accessibilité, qui est déterminé à la fois par l'accessibilité déclarée du membre et par le domaine

d’accessibilité du type conteneur immédiat. Toutefois, le domaine d'accessibilité d'un type imbriqué ne peut pas dépasser celui du type conteneur.

Spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Modificateurs d’accès
- Domaine d’accessibilité
- Restrictions concernant l’utilisation des niveaux d’accessibilité
- Modificateurs d’accès
- public
- private
- protected
- internal

Domaine d'accessibilité (référence C#)

Article • 07/04/2023

Le domaine d'accessibilité d'un membre indique dans quelles sections du programme un membre peut être référencé. Si le membre est imbriqué dans un autre type, son domaine d'accessibilité est déterminé à la fois par le niveau d'accessibilité du membre et par le domaine d'accessibilité du type conteneur immédiat.

Le domaine d'accessibilité d'un type de niveau supérieur est au moins le texte du programme du projet dans lequel il est déclaré. Cela signifie que le domaine inclut tous les fichiers sources de ce projet. Le domaine d'accessibilité d'un type imbriqué est au moins le texte de programme du type dans lequel il est déclaré. Autrement dit, le domaine est le corps du type, qui inclut tous les types imbriqués. Le domaine d'accessibilité d'un type imbriqué ne dépasse jamais celui du type conteneur. Ces concepts sont illustrés dans l'exemple suivant.

Exemple

Cet exemple comporte un type de niveau supérieur, `T1`, et deux classes imbriquées, `M1` et `M2`. Les classes contiennent des champs qui ont des accessibilités déclarées différentes. Dans la méthode `Main`, un commentaire suit chaque instruction pour indiquer le domaine d'accessibilité de chaque membre. Notez que les instructions qui tentent de référencer les membres inaccessibles sont commentées. Si vous souhaitez consulter les erreurs du compilateur provoquées par le référencement d'un membre inaccessible, supprimez les commentaires un par un.

C#

```
public class T1
{
    public static int publicInt;
    internal static int internalInt;
    private static int privateInt = 0;

    static T1()
    {
        // T1 can access public or internal members
        // in a public or private (or internal) nested class.
        M1.publicInt = 1;
        M1.internalInt = 2;
        M2.publicInt = 3;
        M2.internalInt = 4;

        // Cannot access the private member privateInt
```

```
        // in either class:
        // M1.privateInt = 2; //CS0122
    }

    public class M1
    {
        public static int publicInt;
        internal static int internalInt;
        private static int privateInt = 0;
    }

    private class M2
    {
        public static int publicInt = 0;
        internal static int internalInt = 0;
        private static int privateInt = 0;
    }
}

class MainClass
{
    static void Main()
    {
        // Access is unlimited.
        T1.publicInt = 1;

        // Accessible only in current assembly.
        T1.internalInt = 2;

        // Error CS0122: inaccessible outside T1.
        // T1.privateInt = 3;

        // Access is unlimited.
        T1.M1.publicInt = 1;

        // Accessible only in current assembly.
        T1.M1.internalInt = 2;

        // Error CS0122: inaccessible outside M1.
        //    T1.M1.privateInt = 3;

        // Error CS0122: inaccessible outside T1.
        //    T1.M2.publicInt = 1;

        // Error CS0122: inaccessible outside T1.
        //    T1.M2.internalInt = 2;

        // Error CS0122: inaccessible outside M2.
        //    T1.M2.privateInt = 3;

        // Keep the console open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
```

Spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Modificateurs d’accès
- Niveaux d'accessibilité
- Restrictions concernant l’utilisation des niveaux d’accessibilité
- Modificateurs d’accès
- public
- private
- protected
- internal

Limitations sur l'utilisation des niveaux d'accessibilité (informations de référence sur C#)

Article • 07/04/2023

Lorsque vous spécifiez un type dans une déclaration, vérifiez si le niveau d'accessibilité du type dépend du niveau d'accessibilité d'un membre ou d'un autre type. Par exemple, la classe de base directe doit être au moins aussi accessible que la classe dérivée. Les déclarations suivantes entraînent une erreur du compilateur, car la classe de base `BaseClass` est moins accessible que `MyClass` :

C#

```
class BaseClass {...}
public class MyClass: BaseClass {...} // Error
```

Le tableau suivant résume les limitations sur les niveaux d'accessibilité déclarés.

Context	Notes
Classes	La classe de base directe d'un type de classe doit être au moins aussi accessible que le type de classe lui-même.
Interfaces	Les interfaces de base explicites d'un type d'interface doivent être au moins aussi accessibles que le type d'interface lui-même.
Délégués	Le type de retour et les types de paramètres d'un type délégué doivent être au moins aussi accessibles que le type délégué lui-même.
Constantes	Le type d'une constante doit être au moins aussi accessible que la constante elle-même.
Fields	Le type d'un champ doit être au moins aussi accessible que le champ lui-même.
Méthodes	Le type de retour et les types de paramètres d'une méthode doivent être au moins aussi accessibles que la méthode elle-même.
Propriétés	Le type d'une propriété doit être au moins aussi accessible que la propriété elle-même.
Événements	Le type d'un événement doit être au moins aussi accessible que l'événement lui-même.
Indexeurs	Le type et les types de paramètres d'un indexeur doivent être au moins aussi accessibles que l'indexeur lui-même.

Context	Notes
Opérateurs	Le type de retour et les types de paramètres d'un opérateur doivent être au moins aussi accessibles que l'opérateur lui-même.
Constructeurs	Les types de paramètres d'un constructeur doivent être au moins aussi accessibles que le constructeur lui-même.

Exemple

L'exemple suivant contient des déclarations erronées de différents types. Le commentaire qui suit chaque déclaration indique l'erreur du compilateur à attendre.

C#

```
// Restrictions on Using Accessibility Levels
// CS0052 expected as well as CS0053, CS0056, and CS0057
// To make the program work, change access level of both class B
// and MyPrivateMethod() to public.

using System;

// A delegate:
delegate int MyDelegate();

class B
{
    // A private method:
    static int MyPrivateMethod()
    {
        return 0;
    }
}

public class A
{
    // Error: The type B is less accessible than the field A.myField.
    public B myField = new B();

    // Error: The type B is less accessible
    // than the constant A.myConst.
    public readonly B myConst = new B();

    public B MyMethod()
    {
        // Error: The type B is less accessible
        // than the method A.MyMethod.
        return new B();
    }

    // Error: The type B is less accessible than the property A.MyProp
```

```
    public B MyProp
    {
        set
        {
        }
    }

    MyDelegate d = new MyDelegate(B.MyPrivateMethod);
    // Even when B is declared public, you still get the error:
    // "The parameter B.MyPrivateMethod is not accessible due to
    // protection level."

    public static B operator +(A m1, B m2)
    {
        // Error: The type B is less accessible
        // than the operator A.operator +(A,B)
        return new B();
    }

    static void Main()
    {
        Console.Write("Compiled successfully");
    }
}
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Modificateurs d’accès
- Domaine d’accessibilité
- Niveaux d'accessibilité
- Modificateurs d’accès
- public
- private
- protected
- internal

internal (référence C#)

Article • 09/05/2023

Le mot clé `internal` est un modificateur d'accès pour les types et les membres de type.

> Cette page traite de l'accès `internal`. Le mot clé `internal` fait également partie du modificateur d'accès protected internal.

Les types et les membres internes (internal) sont accessibles uniquement dans les fichiers d'un même assembly, comme dans l'exemple suivant :

```
C#
```

```
public class BaseClass
{
    // Only accessible within the same assembly.
    internal static int x = 0;
}
```

Pour obtenir une comparaison de `internal` et des autres modificateurs d'accès, consultez Niveaux d'accessibilité et Modificateurs d'accès.

Pour plus d'informations sur les assemblys, consultez Aassemblys dans .NET.

L'accès interne est fréquemment utilisé lors du développement basé sur les composants, car il permet à un groupe de composants de collaborer de façon privée sans être exposés au reste du code de l'application. Par exemple, un framework de création d'interfaces graphiques utilisateur peut fournir les classes `Control` et `Form` qui coopèrent en utilisant des membres ayant un accès interne. Étant donné que ces membres sont internes, ils ne sont pas exposés au code qui utilise le framework.

Le fait de référencer un type ou un membre avec accès interne en dehors de l'assembly dans lequel il a été défini constitue une erreur.

Exemple 1

Cet exemple contient deux fichiers : `Assembly1.cs` et `Assembly1_a.cs`. Le premier fichier contient la classe de base interne `BaseClass`. Dans le deuxième fichier, une tentative d'instanciation de `BaseClass` génère une erreur.

```
C#
```

```
// Assembly1.cs
// Compile with: /target:library
internal class BaseClass
{
   public static int intM = 0;
}
```

C#

```
// Assembly1_a.cs
// Compile with: /reference:Assembly1.dll
class TestAccess
{
   static void Main()
   {
      var myBase = new BaseClass();   // CS0122
   }
}
```

Exemple 2

Dans cet exemple, utilisez les mêmes fichiers que vous avez utilisés dans l'exemple 1, et remplacez le niveau d'accessibilité `BaseClass` par `public`. Remplacez également le niveau d'accessibilité du membre `intM` par `internal`. Dans ce cas, vous pouvez instancier la classe, mais vous ne pouvez pas accéder au membre interne.

C#

```
// Assembly2.cs
// Compile with: /target:library
public class BaseClass
{
   internal static int intM = 0;
}
```

C#

```
// Assembly2_a.cs
// Compile with: /reference:Assembly2.dll
public class TestAccess
{
   static void Main()
   {
      var myBase = new BaseClass();   // Ok.
      BaseClass.intM = 444;    // CS0117
   }
}
```

Spécification du langage C#

Pour plus d'informations, consultez Accessibilité déclarée dans la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Modificateurs d'accès
- Niveaux d'accessibilité
- Modificateurs
- public
- private
- protected

private (référence C#)

Article • 08/06/2023

Le mot clé `private` est un modificateur d'accès de membre.

> Cette page traite de l'accès `private`. Le mot clé `private` fait également partie du modificateur d'accès private protected.

L'accès privé est le niveau d'accès le moins permissif. Les membres privés sont accessibles uniquement dans le corps de la classe ou le struct dans lequel ils sont déclarés, comme dans cet exemple :

C#

```
class Employee
{
    private int _i;
    double _d;   // private access by default
}
```

Les types imbriqués dans le même corps peuvent également accéder à ces membres privés.

Référencer un membre privé en dehors d'une classe ou d'un struct où il est déclaré serait à l'origine d'une erreur de compilation.

Pour obtenir une comparaison de `private` et des autres modificateurs d'accès, consultez Niveaux d'accessibilité et Modificateurs d'accès.

Exemple

Dans cet exemple, la classe `Employee` contient deux membres de données privés, `_name` et `_salary`. S'agissant de membres privés, ils sont accessibles uniquement par les méthodes membres. Les méthodes publiques `GetName` et `Salary` sont ajoutées pour permettre un accès contrôlé aux membres privés. Le membre `_name` est accessible via une méthode publique et le membre `_salary` est accessible via une propriété publique en lecture seule. Pour plus d'informations, consultez Propriétés.

C#

```
class Employee2
{
```

```
    private readonly string _name = "FirstName, LastName";
    private readonly double _salary = 100.0;

    public string GetName()
    {
        return _name;
    }

    public double Salary
    {
        get { return _salary; }
    }
}

class PrivateTest
{
    static void Main()
    {
        var e = new Employee2();

        // The data members are inaccessible (private), so
        // they can't be accessed like this:
        //    string n = e._name;
        //    double s = e._salary;

        // '_name' is indirectly accessed via method:
        string n = e.GetName();

        // '_salary' is indirectly accessed via property
        double s = e.Salary;
    }
}
```

spécification du langage C#

Pour plus d'informations, consultez Accessibilité déclarée dans la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Modificateurs d'accès
- Niveaux d'accessibilité
- Modificateurs

- public
- protected
- internal

protected (référence C#)

Article • 07/04/2023

Le mot clé `protected` est un modificateur d'accès de membre.

> ⓘ **Notes**
>
> Cette page traite de l'accès `protected`. Le mot clé `protected` fait également partie des modificateurs d'accès **protected internal** et **private protected**.

Un membre protégé est accessible dans sa classe et par les instances de la classe dérivée.

Pour obtenir une comparaison de `protected` et des autres modificateurs d'accès, consultez Niveaux d'accessibilité.

Exemple 1

Un membre protégé d'une classe de base est accessible dans une classe dérivée uniquement si l'accès s'effectue par le biais du type de la classe dérivée. Prenons l'exemple de l'extrait de code suivant :

C#

```
class A
{
    protected int x = 123;
}

class B : A
{
    static void Main()
    {
        var a = new A();
        var b = new B();

        // Error CS1540, because x can only be accessed by
        // classes derived from A.
        // a.x = 10;

        // OK, because this class derives from A.
        b.x = 10;
    }
}
```

L'instruction `a.x = 10` génère une erreur, car elle est appelée dans la méthode statique Main, et pas dans une instance de la classe B.

Les membres de struct ne peuvent pas être protégés, car le struct ne peut pas être hérité.

Exemple 2

Dans cet exemple, la classe `DerivedPoint` est dérivée de `Point`. Vous pouvez donc accéder aux membres protégés de la classe de base directement à partir de la classe dérivée.

C#

```
class Point
{
    protected int x;
    protected int y;
}

class DerivedPoint: Point
{
    static void Main()
    {
        var dpoint = new DerivedPoint();

        // Direct access to protected members.
        dpoint.x = 10;
        dpoint.y = 15;
        Console.WriteLine($"x = {dpoint.x}, y = {dpoint.y}");
    }
}
// Output: x = 10, y = 15
```

Si vous changez les niveaux d'accès de `x` et `y` à private, le compilateur affiche les messages d'erreur suivants :

`'Point.y' is inaccessible due to its protection level.`

`'Point.x' is inaccessible due to its protection level.`

spécification du langage C#

Pour plus d'informations, consultez Accessibilité déclarée dans la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Modificateurs d’accès
- Niveaux d'accessibilité
- Modificateurs
- public
- private
- internal
- Problèmes de sécurité pour les mots clés virtuels internes

public (référence C#)

Article • 07/04/2023

Le mot clé `public` est un modificateur d'accès pour les types et les membres de types. L'accès public est le niveau d'accès le plus permissif. Il n'existe pas de restrictions d'accès aux membres publics, comme dans cet exemple :

C#

```
class SampleClass
{
    public int x; // No access restrictions.
}
```

Pour plus d'informations, consultez Modificateurs d'accès et Niveaux d'accessibilité.

Exemple

Dans l'exemple suivant, deux classes sont déclarées, `PointTest` et `Program`. L'accès aux membres publics `x` et `y` de `PointTest` s'effectue directement à partir de `Program`.

C#

```
class PointTest
{
    public int x;
    public int y;
}

class Program
{
    static void Main()
    {
        var p = new PointTest();
        // Direct access to public members.
        p.x = 10;
        p.y = 15;
        Console.WriteLine($"x = {p.x}, y = {p.y}");
    }
}
// Output: x = 10, y = 15
```

Si vous remplacez le niveau d'accès `public` par private ou protected, le message d'erreur suivant s'affiche :

'PointTest.y' est inaccessible en raison de son niveau de protection.

spécification du langage C#

Pour plus d'informations, consultez Accessibilité déclarée dans la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Modificateurs d'accès
- Mots clés C#
- Modificateurs d'accès
- Niveaux d'accessibilité
- Modificateurs
- private
- protected
- internal

protected internal (Référence C#)

Article • 07/04/2023

La combinaison de mots clés `protected internal` est un modificateur d'accès de membre. Un membre interne protégé est accessible depuis l'assembly actif ou depuis des types dérivés de la classe conteneur. Pour obtenir une comparaison de `protected internal` et des autres modificateurs d'accès, consultez Niveaux d'accessibilité.

Exemple

Un membre interne protégé d'une classe de base est accessible depuis n'importe quel type au sein de son assembly conteneur. Il est également accessible dans une classe dérivée qui se trouve dans un autre assembly seulement si l'accès s'effectue via une variable du type de la classe dérivée. Prenons l'exemple de l'extrait de code suivant :

C#

```
// Assembly1.cs
// Compile with: /target:library
public class BaseClass
{
   protected internal int myValue = 0;
}

class TestAccess
{
    void Access()
    {
        var baseObject = new BaseClass();
        baseObject.myValue = 5;
    }
}
```

C#

```
// Assembly2.cs
// Compile with: /reference:Assembly1.dll
class DerivedClass : BaseClass
{
    static void Main()
    {
        var baseObject = new BaseClass();
        var derivedObject = new DerivedClass();

        // Error CS1540, because myValue can only be accessed by
        // classes derived from BaseClass.
```

```
        // baseObject.myValue = 10;

        // OK, because this class derives from BaseClass.
        derivedObject.myValue = 10;
    }
}
```

Cet exemple contient deux fichiers : `Assembly1.cs` et `Assembly2.cs`. Le premier fichier contient une classe de base publique, `BaseClass`, et une autre classe, `TestAccess`. `BaseClass` possède un membre interne protégé, `myValue`, à laquelle le type `TestAccess` accède. Dans le deuxième fichier, une tentative d'accès à `myValue` via une instance de `BaseClass` génère une erreur, tandis qu'un accès à ce membre via une instance d'une classe dérivée, `DerivedClass`, réussit.

Les membres de struct ne peuvent pas être `protected internal`, car le struct ne peut pas être hérité.

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Modificateurs d'accès
- Niveaux d'accessibilité
- Modificateurs
- public
- private
- internal
- Problèmes de sécurité pour les mots clés virtuels internes

private protected (Référence C#)

Article • 07/04/2023

La combinaison de mots clés `private protected` est un modificateur d'accès de membre. Un membre protégé privé est accessible par les types dérivés de la classe conteneur, mais seulement au sein de son assembly conteneur. Pour obtenir une comparaison de `private protected` et des autres modificateurs d'accès, consultez Niveaux d'accessibilité.

> ⓘ **Notes**
>
> Le modificateur d'accès `private protected` est valide dans C# 7.2 et ultérieur.

Exemple

Un membre protégé privé d'une classe de base est accessible à partir des types dérivés de son assembly conteneur seulement si le type statique de la variable est le type de la classe dérivée. Prenons l'exemple de l'extrait de code suivant :

C#

```
public class BaseClass
{
    private protected int myValue = 0;
}

public class DerivedClass1 : BaseClass
{
    void Access()
    {
        var baseObject = new BaseClass();

        // Error CS1540, because myValue can only be accessed by
        // classes derived from BaseClass.
        // baseObject.myValue = 5;

        // OK, accessed through the current derived class instance
        myValue = 5;
    }
}
```

C#

```
// Assembly2.cs
// Compile with: /reference:Assembly1.dll
class DerivedClass2 : BaseClass
{
    void Access()
    {
        // Error CS0122, because myValue can only be
        // accessed by types in Assembly1
        // myValue = 10;
    }
}
```

Cet exemple contient deux fichiers : `Assembly1.cs` et `Assembly2.cs`. Le premier fichier contient une classe de base publique, `BaseClass`, et un type qui en est dérivé, `DerivedClass1`. `BaseClass` possède un membre protégé privé, `myValue`, auquel `DerivedClass1` tente d'accéder de deux manières. La première tentative d'accès à `myValue` via une instance de `BaseClass` génère une erreur. Cependant, la tentative de l'utiliser comme un membre hérité dans `DerivedClass1` réussit.

Dans le deuxième fichier, une tentative d'accès à `myValue` en tant que membre hérité de `DerivedClass2` génère une erreur, car il est accessible seulement par des types dérivés dans Assembly1.

Si `Assembly1.cs` contient un InternalsVisibleToAttribute qui nomme `Assembly2`, la classe dérivée `DerivedClass2` a accès aux membres de `private protected` déclarés dans `BaseClass`. `InternalsVisibleTo` rend les membres de `private protected` visibles par les classes dérivées dans d'autres assemblys.

Les membres de struct ne peuvent pas être `private protected`, car le struct ne peut pas être hérité.

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Modificateurs d'accès

- Niveaux d'accessibilité
- Modificateurs
- public
- private
- internal
- Problèmes de sécurité pour les mots clés virtuels internes

abstract (référence C#)

Article • 07/04/2023

Le modificateur `abstract` indique que l'élément en cours de modification a une implémentation manquante ou incomplète. Le modificateur abstract peut être utilisé avec des classes, des méthodes, des propriétés, des indexeurs et des événements. Dans une déclaration de classe, utilisez le modificateur `abstract` pour indiquer qu'une classe doit uniquement servir de classe de base pour d'autres classes, et ne pas être instanciée toute seule. Les membres définis comme abstraits doivent être implémentés par des classes non abstraites dérivées de la classe abstraite.

Exemple 1

Dans cet exemple, la classe `Square` doit fournir une implémentation de `GetArea`, car elle dérive de la classe `Shape` :

C#

```
abstract class Shape
{
    public abstract int GetArea();
}

class Square : Shape
{
    private int _side;

    public Square(int n) => _side = n;

    // GetArea method is required to avoid a compile-time error.
    public override int GetArea() => _side * _side;

    static void Main()
    {
        var sq = new Square(12);
        Console.WriteLine($"Area of the square = {sq.GetArea()}");
    }
}
// Output: Area of the square = 144
```

Les classes abstraites présentent les caractéristiques suivantes :

- Une classe abstraite ne peut pas être instanciée.
- Une classe abstraite peut contenir des méthodes et accesseurs abstraits.

- Il n'est pas possible de modifier une classe abstraite avec le modificateur sealed, car les deux modificateurs ont des significations opposées. Le modificateur `sealed` empêche qu'une classe soit héritée et le modificateur `abstract` exige qu'une classe soit héritée.

- Une classe non abstraite dérivée d'une classe abstraite doit inclure des implémentations réelles de tous les accesseurs et méthodes abstraits hérités.

Dans une déclaration de méthode ou de propriété, utilisez le modificateur `abstract` pour indiquer que la méthode ou la propriété ne contient pas d'implémentation.

Les méthodes abstraites présentent les caractéristiques suivantes :

- Une méthode abstraite est implicitement une méthode virtuelle.

- Les déclarations de méthodes abstraites sont autorisées uniquement dans les classes abstraites.

- Comme une déclaration de méthode abstraite ne fournit pas d'implémentation réelle, il n'y a pas de corps de méthode ; la déclaration de méthode se termine simplement par un point-virgule, et la signature n'est pas suivie d'accolades ({ }). Par exemple :

  ```
  C#

  public abstract void MyMethod();
  ```

  L'implémentation est fournie par une méthode override, qui est membre d'une classe non abstraite.

- L'utilisation des modificateurs static ou virtual dans une déclaration de méthode abstraite serait une erreur.

Les propriétés abstraites se comportent comme les méthodes abstraites, à l'exception des différences dans la syntaxe de déclaration et d'appel.

- L'utilisation du modificateur `abstract` sur une propriété statique serait une erreur.

- Une propriété abstraite héritée peut être substituée dans une classe dérivée en incluant une déclaration de propriété qui utilise le modificateur override.

Pour plus d'informations sur les classes abstraites, consultez Classes abstract et sealed et membres de classe.

Une classe abstraite doit fournir une implémentation pour tous les membres d'interface.

Une classe abstraite qui implémente une interface peut mapper les méthodes d'interface à des méthodes abstraites. Par exemple :

C#

```
interface I
{
    void M();
}

abstract class C : I
{
    public abstract void M();
}
```

Exemple 2

Dans cet exemple, la classe `DerivedClass` est dérivée de la classe abstraite `BaseClass`. La classe abstraite contient une méthode abstraite, `AbstractMethod`, et deux propriétés abstraites, `X` et `Y`.

C#

```
// Abstract class
abstract class BaseClass
{
    protected int _x = 100;
    protected int _y = 150;

    // Abstract method
    public abstract void AbstractMethod();

    // Abstract properties
    public abstract int X { get; }
    public abstract int Y { get; }
}

class DerivedClass : BaseClass
{
    public override void AbstractMethod()
    {
        _x++;
        _y++;
    }

    public override int X   // overriding property
    {
        get
        {
```

```
            return _x + 10;
        }
    }

    public override int Y   // overriding property
    {
        get
        {
            return _y + 10;
        }
    }

    static void Main()
    {
        var o = new DerivedClass();
        o.AbstractMethod();
        Console.WriteLine($"x = {o.X}, y = {o.Y}");
    }
}
// Output: x = 111, y = 161
```

Dans l'exemple précédent, si vous tentez d'instancier la classe abstraite en utilisant une instruction comme celle-ci :

```
C#

BaseClass bc = new BaseClass();   // Error
```

Vous obtenez une erreur indiquant que le compilateur ne peut pas créer une instance de la classe abstraite BaseClass.

Spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Modificateurs
- virtual
- override
- Mots clés C#

async (référence C#)

Article • 27/03/2023

Utilisez le modificateur `async` pour spécifier qu'une méthode, une expression lambda ou une méthode anonyme sont asynchrones. Si vous utilisez ce modificateur sur une méthode ou une expression, il s'agit d'une *méthode async*. L'exemple suivant définit une méthode async nommée `ExampleMethodAsync` :

C#

```
public async Task<int> ExampleMethodAsync()
{
    //...
}
```

Si vous débutez en programmation asynchrone ou que vous ne comprenez pas comment une méthode async utilise l'opérateur await pour un travail potentiellement long sans bloquer le thread de l'appelant, lisez l'introduction dans Programmation asynchrone avec async et await. Le code suivant se trouve dans une méthode async et appelle la méthode HttpClient.GetStringAsync :

C#

```
string contents = await httpClient.GetStringAsync(requestUrl);
```

Une méthode async s'exécute de façon synchrone jusqu'à ce qu'elle atteigne sa première expression `await`, où elle est suspendue jusqu'à ce que la tâche attendue soit terminée. Dans le même temps, le contrôle retourne à l'appelant de la méthode, comme le montre l'exemple indiqué dans la section suivante.

Si la méthode que le mot clé `async` modifie ne contient pas une expression ou une instruction `await`, la méthode s'exécute de façon synchrone. Un avertissement du compilateur vous signale toutes les méthodes async qui ne contiennent pas d'instructions `await`, car cette situation peut indiquer une erreur. Consultez Avertissement du compilateur (niveau 1) CS4014.

Le mot clé `async` est contextuel, car il est un mot clé uniquement lorsqu'il modifie une méthode, une expression lambda ou une méthode anonyme. Dans tous les autres contextes, il est interprété comme un identificateur.

Exemple

L'exemple suivant montre la structure et le flux de contrôle entre un gestionnaire d'événements asynchrones, `StartButton_Click`, et une méthode async, `ExampleMethodAsync`. Le résultat de la méthode async est le nombre de caractères d'une page web. Le code convient pour une application WPF (Windows Presentation Foundation) ou une application du Windows Store que vous créez dans Visual Studio ; consultez les commentaires du code pour configurer l'application.

Vous pouvez exécuter ce code dans Visual Studio en tant qu'application Windows Presentation Foundation (WPF) ou qu'application du Windows Store. Vous avez besoin d'un contrôle Button nommé `StartButton` et d'un contrôle Textbox nommé `ResultsTextBox`. N'oubliez pas de définir les noms et le gestionnaire afin d'obtenir un résultat semblable à ceci :

XAML

```
<Button Content="Button" HorizontalAlignment="Left" Margin="88,77,0,0"
VerticalAlignment="Top" Width="75"
        Click="StartButton_Click" Name="StartButton"/>
<TextBox HorizontalAlignment="Left" Height="137" Margin="88,140,0,0"
TextWrapping="Wrap"
         Text="&lt;Enter a URL&gt;" VerticalAlignment="Top" Width="310"
Name="ResultsTextBox"/>
```

Pour exécuter le code en tant qu'application WPF :

- Collez ce code dans la classe `MainWindow` dans MainWindow.xaml.cs.
- Ajoutez une référence à System.Net.Http.
- Ajoutez une directive `using` à System.Net.Http.

Pour exécuter le code comme une application du Windows Store :

- Collez ce code dans la classe `MainPage` dans MainPage.xaml.cs.
- Ajouter des directives using pour System.Net.Http et System.Threading.Tasks.

C#

```
private async void StartButton_Click(object sender, RoutedEventArgs e)
{
    // ExampleMethodAsync returns a Task<int>, which means that the method
    // eventually produces an int result. However, ExampleMethodAsync
returns
    // the Task<int> value as soon as it reaches an await.
    ResultsTextBox.Text += "\n";

    try
    {
        int length = await ExampleMethodAsync();
```

```
        // Note that you could put "await ExampleMethodAsync()" in the next
line where
        // "length" is, but due to when '+=' fetches the value of
ResultsTextBox, you
        // would not see the global side effect of ExampleMethodAsync
setting the text.
        ResultsTextBox.Text += String.Format("Length: {0:N0}\n", length);
    }
    catch (Exception)
    {
        // Process the exception if one occurs.
    }
}

public async Task<int> ExampleMethodAsync()
{
    var httpClient = new HttpClient();
    int exampleInt = (await
httpClient.GetStringAsync("http://msdn.microsoft.com")).Length;
    ResultsTextBox.Text += "Preparing to finish ExampleMethodAsync.\n";
    // After the following return statement, any method that's awaiting
    // ExampleMethodAsync (in this case, StartButton_Click) can get the
    // integer result.
    return exampleInt;
}
// The example displays the following output:
// Preparing to finish ExampleMethodAsync.
// Length: 53292
```

> **ⓘ Important**
>
> Pour plus d'informations sur les tâches et sur le code qui s'exécute en attendant une tâche, consultez **Programmation asynchrone avec async et await**. Pour obtenir un exemple de console complète qui utilise des éléments similaires, consultez **Traiter les tâches asynchrones à mesure qu'elles se terminent (C#)**.

Types de retour

Une méthode async peut avoir les types de retour suivants :

- Task
- Task<TResult>
- void. Les méthodes `async void` sont généralement déconseillées pour le code autre que les gestionnaires d'événements parce que les appelants ne peuvent pas `await` ces méthodes et doivent implémenter un mécanisme différent pour signaler les conditions d'erreur ou les complétions réussies.

- Tout type qui a une méthode accessible `GetAwaiter`. Le type `System.Threading.Tasks.ValueTask<TResult>` est une implémentation de ce genre. Il est disponible en ajoutant le package NuGet `System.Threading.Tasks.Extensions`.

La méthode async ne peut déclarer aucun paramètre in, ref ou out, ni avoir une valeur de retour de référence, mais elle peut appeler des méthodes qui ont ces paramètres.

Vous spécifiez `Task<TResult>` comme type de retour d'une méthode async si l'instruction return de la méthode spécifie un opérande de type `TResult`. Utilisez `Task` si aucune valeur significative n'est retournée lorsque la méthode est terminée. En d'autres termes, un appel à la méthode retourne `Task`, mais lorsque `Task` est terminé, toute expression `await` qui attend `Task` prend la valeur `void`.

Vous utilisez le type de retour `void` principalement pour définir les gestionnaires d'événements, qui ont besoin de ce type de retour. L'appelant d'une méthode async retournant `void` ne peut pas l'attendre et ne peut pas intercepter les exceptions levées par la méthode.

Vous retournez un autre type, en général un type valeur, qui a une méthode `GetAwaiter` permettant de limiter les allocations de mémoire dans les sections de code critiques pour les performances.

Pour obtenir plus d'informations et des exemples, consultez Types de retour Async.

Voir aussi

- AsyncStateMachineAttribute
- await
- Programmation asynchrone avec async et await
- Traiter les tâches asynchrones terminées
- Blog .NET : Fonctionnement réel d'async/await en C#

const (référence C#)

Article • 29/05/2023

Vous utilisez le mot clé `const` pour déclarer un champ constant ou une constante locale. Les champs et les éléments locaux constants ne sont pas des variables et ne peuvent pas être modifiés. Les constantes peuvent être des chiffres, des valeurs booléennes, des chaînes ou une référence null. Ne créez pas une constante pour représenter des informations qui doivent être modifiées. Par exemple, n'utilisez pas un champ constant pour stocker le prix d'un service, le numéro de version du produit ou le nom de la marque d'une société. Ces valeurs peuvent changer dans le temps, et dans la mesure où les compilateurs propagent les constantes, le code compilé avec vos bibliothèques devra être recompilé pour refléter ces modifications. Consultez également le mot clé readonly. Par exemple :

C#

```
const int X = 0;
public const double GravitationalConstant = 6.673e-11;
private const string ProductName = "Visual C#";
```

À partir de C# 10, les chaînes interpolées peuvent être des constantes si toutes les expressions utilisées sont également des chaînes constantes. Cette fonctionnalité peut améliorer le code qui génère des chaînes constantes :

C#

```
const string Language = "C#";
const string Platform = ".NET";
const string Version = "10.0";
const string FullProductName = $"{Platform} - Language: {Language} Version:
{Version}";
```

Remarques

Le type d'une déclaration constante indique le type des membres introduit par la déclaration. L'initialiseur d'une constante locale ou d'un champ constant doit être une expression constante qui peut être implicitement convertie en type cible.

Une expression constante est une expression qui peut être complètement évaluée au moment de la compilation. C'est pourquoi les seules valeurs possibles pour les constantes des types référence sont des chaînes et une référence null.

La déclaration constante peut déclarer plusieurs constantes, notamment :

C#

```
public const double X = 1.0, Y = 2.0, Z = 3.0;
```

Le modificateur `static` n'est pas autorisé dans une déclaration constante.

Une constante peut être utilisée dans une expression constante, comme suit :

C#

```
public const int C1 = 5;
public const int C2 = C1 + 100;
```

> ⓘ **Notes**
>
> Le mot clé **readonly** est différent du mot clé `const`. Un champ `const` ne peut être initialisé qu'au moment de la déclaration du champ. Un champ `readonly` peut être initialisé dans la déclaration ou dans un constructeur. C'est pourquoi, les champs `readonly` peuvent avoir des valeurs différentes en fonction du constructeur utilisé. De même, bien qu'un champ `const` soit une constante au moment de la compilation, le champ `readonly` peut être utilisé pour des constantes au moment de l'exécution, comme ci-après : `public static readonly uint l1 = (uint)DateTime.Now.Ticks;`

Exemples

C#

```
public class ConstTest
{
    class SampleClass
    {
        public int x;
        public int y;
        public const int C1 = 5;
        public const int C2 = C1 + 5;

        public SampleClass(int p1, int p2)
        {
            x = p1;
            y = p2;
        }
```

```
    }

    static void Main()
    {
        var mC = new SampleClass(11, 22);
        Console.WriteLine($"x = {mC.x}, y = {mC.y}");
        Console.WriteLine($"C1 = {SampleClass.C1}, C2 = {SampleClass.C2}");
    }
}
/* Output
    x = 11, y = 22
    C1 = 5, C2 = 10
*/
```

L'exemple suivant montre comment déclarer une constante locale :

C#

```
public class SealedTest
{
    static void Main()
    {
        const int C = 707;
        Console.WriteLine($"My local constant = {C}");
    }
}
// Output: My local constant = 707
```

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Constantes
- Expressions constantes

Voir aussi

- Informations de référence sur C#
- Mots clés C#
- readonly

événement (référence C#)

Article • 10/05/2023

Le mot clé `event` sert à déclarer un événement dans une classe d'éditeur.

Exemple

L'exemple suivant montre comment déclarer et déclencher un événement qui utilise EventHandler comme type délégué sous-jacent. Pour obtenir l'exemple de code complet qui illustre aussi comment utiliser le type délégué générique EventHandler<TEventArgs> et comment s'abonner à un événement et créer une méthode de gestionnaire d'événements, consultez Guide pratique pour publier des événements conformes aux lignes directrices .NET.

C#

```
public class SampleEventArgs
{
    public SampleEventArgs(string text) { Text = text; }
    public string Text { get; } // readonly
}

public class Publisher
{
    // Declare the delegate (if using non-generic pattern).
    public delegate void SampleEventHandler(object sender, SampleEventArgs
e);

    // Declare the event.
    public event SampleEventHandler SampleEvent;

    // Wrap the event in a protected virtual method
    // to enable derived classes to raise the event.
    protected virtual void RaiseSampleEvent()
    {
        // Raise the event in a thread-safe manner using the ?. operator.
        SampleEvent?.Invoke(this, new SampleEventArgs("Hello"));
    }
}
```

Les événements constituent un type spécial de délégué de multidiffusion qui peut uniquement être appelé au sein de la classe (ou classes dérivées) ou de la structure où ils sont déclarés (la classe d'éditeur). Si d'autres classes ou structs s'abonnent à l'événement, leurs méthodes de gestionnaire d'événements sont appelées quand la

classe d'éditeur déclenche l'événement. Pour plus d'informations et pour obtenir des exemples de code, consultez Événements et Délégués.

Les événements peuvent être marqués comme public, privé, protégé, interne, protégé interne ou privé protégé. Ces modificateurs d'accès définissent comment les utilisateurs de la classe peuvent accéder à l'événement. Pour plus d'informations, consultez Modificateurs d'accès.

Mots clés et événements

Les mots clés suivants s'appliquent aux événements.

Mot clé	Description	Informations supplémentaires
static	Rend le champ accessible à tout moment aux appelants, même s'il n'existe aucune instance de la classe.	Classes statiques et membres de classe statique
virtual	Permet aux classes dérivées de substituer le comportement d'événement à l'aide du mot clé override.	Héritage
sealed	Spécifie que pour les classes dérivées l'événement n'est plus virtuel.	
abstract	Le compilateur ne génère pas les blocs d'accesseurs d'événement `add` et `remove`, et par conséquent les classes dérivées doivent fournir leur propre implémentation.	

Un événement peut être déclaré comme événement statique à l'aide du mot clé static. Cela rend le champ accessible à tout moment aux appelants, même s'il n'existe aucune instance de la classe. Pour plus d'informations, consultez la page Classes statiques et membres de classes statiques.

Un événement peut être marqué comme événement virtuel à l'aide du mot clé virtual. Cela permet aux classes dérivées de substituer le comportement d'événement à l'aide du mot clé override. Pour plus d'informations, consultez Héritage. Un événement qui se substitue à un événement virtuel peut également être sealed, ce qui signifie que pour les classes dérivées il n'est plus virtuel. Pour finir, un événement peut être déclaré abstract, ce qui signifie que le compilateur ne génère pas les blocs d'accesseurs d'événement `add` et `remove`. Ainsi, les classes dérivées doivent fournir leur propre implémentation.

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- add
- remove
- Modificateurs
- Guide pratique pour combiner des délégués (délégués de multidiffusion)

extern (référence C#)

Article • 07/04/2023

Le modificateur `extern` permet de déclarer une méthode qui est implémentée en externe. Le modificateur `extern` est souvent utilisé avec l'attribut `DllImport` lors de l'utilisation de services Interop à appeler dans du code non managé. Dans ce cas, la méthode doit également être déclarée comme `static`, comme indiqué dans l'exemple suivant :

C#

```
[DllImport("avifil32.dll")]
private static extern void AVIFileInit();
```

Le mot clé `extern` peut également définir un alias d'assembly externe, permettant ainsi de référencer différentes versions du même composant à partir d'un seul assembly. Pour plus d'informations, consultez extern alias.

C'est une erreur d'utiliser conjointement les modificateurs abstract et `extern` pour modifier le même membre. L'utilisation du modificateur `extern` signifie que la méthode est implémentée en dehors du code C#, tandis que l'utilisation du modificateur `abstract` signifie que l'implémentation de la méthode n'est pas effectuée dans la classe.

L'utilisation du mot clé extern est plus restreinte dans C# que dans C++. Pour comparer son utilisation dans C# et C++, consultez : Utilisation d'extern pour spécifier la liaison dans le Guide de référence du langage C++.

Exemple 1

Dans cet exemple, le programme reçoit une chaîne provenant de l'utilisateur et l'affiche dans une boîte de message. Le programme utilise la méthode `MessageBox` importée de la bibliothèque User32.dll.

C#

```
//using System.Runtime.InteropServices;
class ExternTest
{
    [DllImport("User32.dll", CharSet=CharSet.Unicode)]
    public static extern int MessageBox(IntPtr h, string m, string c, int
type);
```

```
    static int Main()
    {
        string myString;
        Console.Write("Enter your message: ");
        myString = Console.ReadLine();
        return MessageBox((IntPtr)0, myString, "My Message Box", 0);
    }
}
```

Exemple 2

Cet exemple illustre un programme C# qui fait appel à une bibliothèque C (une DLL native).

1. Créez le fichier C suivant et nommez-le `cmdll.c` :

```
C

// cmdll.c
// Compile with: -LD
int __declspec(dllexport) SampleMethod(int i)
{
  return i*10;
}
```

2. Ouvrez une fenêtre d'invite de commandes d'outils natifs Visual Studio x64 (ou x32) à partir du répertoire d'installation de Visual Studio et compilez le fichier `cmdll.c` en tapant **cl -LD cmdll.c** à l'invite de commandes.

3. Dans le même répertoire, créez le fichier C# suivant et nommez-le `cm.cs` :

```
C#

// cm.cs
using System;
using System.Runtime.InteropServices;
public class MainClass
{
    [DllImport("Cmdll.dll")]
      public static extern int SampleMethod(int x);

    static void Main()
    {
        Console.WriteLine("SampleMethod() returns {0}.",
SampleMethod(5));
    }
}
```

4. Ouvrez une fenêtre d'invite de commandes d'outils natifs Visual Studio x64 (ou x32) à partir du répertoire d'installation de Visual Studio et compilez le fichier `cm.cs` en tapant :

   > **csc cm.cs** (pour l'invite de commandes x64) —ou— **csc -platform:x86 cm.cs** (pour l'invite de commandes x32)

   Cette opération crée le fichier exécutable `cm.exe`.

5. Exécuter `cm.exe`. La méthode `SampleMethod` passe la valeur 5 au fichier DLL, qui retourne la valeur multipliée par 10. Le programme génère la sortie suivante :

   Sortie

   ```
   SampleMethod() returns 50.
   ```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- System.Runtime.InteropServices.DllImportAttribute
- Référence C#
- Guide de programmation C#
- Mots clés C#
- Modificateurs

in (modificateur générique) (Référence C#)

Article • 07/04/2023

Pour les paramètres de type générique, le mot clé `in` spécifie que le paramètre de type est contravariant. Vous pouvez utiliser le mot clé `in` dans les interfaces et délégués génériques.

La contravariance permet d'utiliser un type moins dérivé que celui spécifié par le paramètre générique. Cela permet la conversion implicite des classes qui implémentent des interfaces contravariantes, ainsi que la conversion implicite des types délégués. La covariance et la contravariance des paramètres de type générique sont prises en charge pour les types référence, mais pas pour les types valeur.

Un type peut être déclaré comme étant contravariant dans une interface ou un délégué générique uniquement s'il définit le type des paramètres d'une méthode et non le type de retour d'une méthode. Les paramètres `In`, `ref` et `out` doivent être invariants, ce qui signifie qu'ils ne sont ni covariants ni contravariants.

Une interface qui possède un paramètre de type contravariant permet à ses méthodes d'accepter des arguments de types moins dérivés que ceux spécifiés par le paramètre de type d'interface. Par exemple, dans l'interface IComparer<T>, le type T est contravariant, vous pouvez attribuer un objet du type `IComparer<Person>` à un objet du type `IComparer<Employee>` sans utiliser de méthode de conversion spéciale si `Employee` hérite `Person`.

Un délégué contravariant peut être assigné à un autre délégué du même type, mais avec un paramètre de type générique moins dérivé.

Pour plus d'informations, consultez Covariance et contravariance.

Interface générique contravariante

L'exemple suivant montre comment déclarer, étendre et implémenter une interface générique contravariante. Il montre également comment utiliser la conversion implicite pour les classes qui implémentent cette interface.

C#

```
// Contravariant interface.
interface IContravariant<in A> { }
```

```
// Extending contravariant interface.
interface IExtContravariant<in A> : IContravariant<A> { }

// Implementing contravariant interface.
class Sample<A> : IContravariant<A> { }

class Program
{
    static void Test()
    {
        IContravariant<Object> iobj = new Sample<Object>();
        IContravariant<String> istr = new Sample<String>();

        // You can assign iobj to istr because
        // the IContravariant interface is contravariant.
        istr = iobj;
    }
}
```

Délégué générique contravariant

L'exemple de code suivant montre comment déclarer, instancier et invoquer un délégué générique contravariant. Il montre également comment convertir implicitement un type délégué.

C#

```
// Contravariant delegate.
public delegate void DContravariant<in A>(A argument);

// Methods that match the delegate signature.
public static void SampleControl(Control control)
{ }
public static void SampleButton(Button button)
{ }

public void Test()
{

    // Instantiating the delegates with the methods.
    DContravariant<Control> dControl = SampleControl;
    DContravariant<Button> dButton = SampleButton;

    // You can assign dControl to dButton
    // because the DContravariant delegate is contravariant.
    dButton = dControl;

    // Invoke the delegate.
    dButton(new Button());
}
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- out
- Covariance et contravariance
- Modificateurs

new, modificateur (référence C#)

Article • 07/04/2023

En cas d'utilisation comme un modificateur de déclaration, le mot clé `new` masque explicitement un membre qui est hérité d'une classe de base. Lorsque vous masquez un membre hérité, la version dérivée du membre remplace la version de classe de base. Cela suppose que la version de classe de base du membre est visible, car elle serait déjà masquée si elle était marquée comme `private` ou, dans certains cas, `internal`. Bien que vous puissiez masquer des membres `public` ou `protected` sans utiliser le modificateur `new`, vous obtenez un avertissement du compilateur. Si vous utilisez `new` pour masquer explicitement un membre, il supprime cet avertissement.

Vous pouvez également utiliser le mot clé `new` pour créer une instance d'un type ou l'utiliser comme une contrainte de type générique.

Pour masquer un membre hérité, déclarez-le dans la classe dérivée en utilisant le même nom de membre, puis modifiez-le à l'aide du mot-clé `new`. Par exemple :

C#

```
public class BaseC
{
    public int x;
    public void Invoke() { }
}
public class DerivedC : BaseC
{
    new public void Invoke() { }
}
```

Dans cet exemple, `BaseC.Invoke` est masqué par `DerivedC.Invoke`. Le champ `x` n'est pas affecté parce qu'il n'est pas masqué par un nom semblable.

Le masquage de noms par le biais de l'héritage peut prendre l'une des formes suivantes :

- En général, une constante, un champ, une propriété ou un type qui est introduit dans une classe ou une structure masque tous les membres de la classe de base qui partagent son nom. Il existe des cas spéciaux. Par exemple, si vous déclarez un nouveau champ avec le nom `N` pour avoir un type qui n'est pas invocable, et qu'un type de base déclare `N` en tant que méthode, le nouveau champ ne masquera pas la déclaration de base dans la syntaxe d'appel. Pour plus d'informations, consultez la section Recherche de membres de la spécification du langage C#.

- Une méthode introduite dans une classe ou une structure masque les propriétés, les champs et les types qui partagent ce nom, dans la classe de base. Cela masque également toutes les méthodes de la classe de base ayant la même signature.
- Un indexeur introduit dans une classe ou un struct masque tous les indexeurs de la classe de base ayant la même signature.

L'utilisation des opérateurs `new` et override sur le même membre est une erreur, car ces deux modificateurs ont des significations qui s'excluent mutuellement. Le modificateur `new` crée un nouveau membre avec le même nom et entraîne le masquage du membre d'origine. Le modificateur `override` étend l'implémentation pour un membre hérité.

L'utilisation du modificateur `new` dans une déclaration qui ne masque pas un membre hérité génère un avertissement.

Exemples

Dans cet exemple, une classe de base, `BaseC` et une classe dérivée, `DerivedC`, utilisent le même nom de champ `x`, masquant ainsi la valeur du champ hérité. Cet exemple illustre l'utilisation du modificateur `new`. Il montre aussi comment accéder aux membres masqués de la classe de base en utilisant leurs noms complets.

C#

```
public class BaseC
{
    public static int x = 55;
    public static int y = 22;
}

public class DerivedC : BaseC
{
    // Hide field 'x'.
    new public static int x = 100;

    static void Main()
    {
        // Display the new value of x:
        Console.WriteLine(x);

        // Display the hidden value of x:
        Console.WriteLine(BaseC.x);

        // Display the unhidden member y:
        Console.WriteLine(y);
    }
}
```

```
/*
Output:
100
55
22
*/
```

Dans cet exemple, une classe imbriquée masque une classe du même nom dans la classe de base. Cet exemple illustre l'utilisation du modificateur `new` pour éliminer le message d'avertissement, ainsi que l'accès aux membres de la classe masquée à l'aide de leurs noms complets.

C#

```
public class BaseC
{
    public class NestedC
    {
        public int x = 200;
        public int y;
    }
}

public class DerivedC : BaseC
{
    // Nested type hiding the base type members.
    new public class NestedC
    {
        public int x = 100;
        public int y;
        public int z;
    }

    static void Main()
    {
        // Creating an object from the overlapping class:
        NestedC c1  = new NestedC();

        // Creating an object from the hidden class:
        BaseC.NestedC c2 = new BaseC.NestedC();

        Console.WriteLine(c1.x);
        Console.WriteLine(c2.x);
    }
}
/*
Output:
100
200
*/
```

Si vous supprimez le modificateur `new`, le programme peut encore être compilé et exécuté, mais vous obtiendrez l'avertissement suivant :

```text
The keyword new is required on 'MyDerivedC.x' because it hides inherited
member 'MyBaseC.x'.
```

spécification du langage C#

Pour plus d'informations, consultez la section Modificateur new de la spécification du langage C#.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Modificateurs
- Versioning avec les mots clés override et new
- Savoir quand utiliser les mots clés override et new

out (modificateur générique) (référence C#)

Article • 09/05/2023

Pour les paramètres de type générique, le mot clé `out` spécifie que le paramètre de type est covariant. Vous pouvez utiliser le mot clé `out` dans les interfaces et délégués génériques.

La covariance permet d'utiliser un type plus dérivé que celui spécifié par le paramètre générique. Cela permet la conversion implicite des classes qui implémentent des interfaces covariantes, ainsi que la conversion implicite des types délégués. La covariance et la contravariance sont prises en charge pour les types référence, mais pas pour les types valeur.

Une interface qui possède un paramètre de type covariant permet à ses méthodes de retourner des types plus dérivés que ceux spécifiés par le paramètre de type. Par exemple, comme dans le .NET Framework 4, dans IEnumerable<T>, le type T est covariant, vous pouvez assigner un objet du type `IEnumerable(Of String)` à un objet du type `IEnumerable(Of Object)` sans utiliser de méthode de conversion spéciale.

Un délégué covariant peut être assigné à un autre délégué du même type, mais avec un paramètre de type générique plus dérivé.

Pour plus d'informations, consultez Covariance et contravariance.

Exemple - interface générique covariante

L'exemple suivant montre comment déclarer, étendre et implémenter une interface générique covariante. Il montre également comment utiliser la conversion implicite pour les classes qui implémentent une interface covariante.

C#

```
// Covariant interface.
interface ICovariant<out R> { }

// Extending covariant interface.
interface IExtCovariant<out R> : ICovariant<R> { }

// Implementing covariant interface.
class Sample<R> : ICovariant<R> { }

class Program
```

```
{
    static void Test()
    {
        ICovariant<Object> iobj = new Sample<Object>();
        ICovariant<String> istr = new Sample<String>();

        // You can assign istr to iobj because
        // the ICovariant interface is covariant.
        iobj = istr;
    }
}
```

Dans une interface générique, un paramètre de type peut être déclaré covariant s'il satisfait aux conditions suivantes :

- Le paramètre de type est utilisé uniquement comme type de retour des méthodes d'interface, mais pas comme type des arguments de méthode.

  > ⓘ **Notes**
  >
  > Il existe une exception à cette règle. Si une interface covariante a un délégué générique contravariant comme paramètre de méthode, vous pouvez utiliser le type covariant comme paramètre de type générique pour ce délégué. Pour plus d'informations sur les délégués génériques covariants et contravariants, consultez **Variance dans les délégués** et **Utilisation de la variance pour les délégués génériques Func et Action**.

- Le paramètre de type n'est pas utilisé comme contrainte générique pour les méthodes d'interface.

Exemple - délégué générique covariant

L'exemple de code suivant montre comment déclarer, instancier et appeler un délégué générique covariant. Il montre également comment convertir implicitement des types délégués.

C#

```
// Covariant delegate.
public delegate R DCovariant<out R>();

// Methods that match the delegate signature.
public static Control SampleControl()
{ return new Control(); }
```

```
public static Button SampleButton()
{ return new Button(); }

public void Test()
{
    // Instantiate the delegates with the methods.
    DCovariant<Control> dControl = SampleControl;
    DCovariant<Button> dButton = SampleButton;

    // You can assign dButton to dControl
    // because the DCovariant delegate is covariant.
    dControl = dButton;

    // Invoke the delegate.
    dControl();
}
```

Dans un délégué générique, un type peut être déclaré comme étant covariant s'il est utilisé uniquement comme type de retour des méthodes, mais pas pour des arguments de méthode.

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Variance dans les interfaces génériques
- in
- Modificateurs

override (référence C#)

Article • 07/04/2023

Le modificateur `override` est nécessaire pour étendre ou modifier l'implémentation abstraite ou virtuelle d'une méthode, d'une propriété, d'un indexeur ou d'un événement hérités.

Dans l'exemple suivant, la classe `Square` doit fournir une implémentation substituée de `GetArea`, car `GetArea` est héritée de la classe abstraite `Shape` :

C#

```
abstract class Shape
{
    public abstract int GetArea();
}

class Square : Shape
{
    private int _side;

    public Square(int n) => _side = n;

    // GetArea method is required to avoid a compile-time error.
    public override int GetArea() => _side * _side;

    static void Main()
    {
        var sq = new Square(12);
        Console.WriteLine($"Area of the square = {sq.GetArea()}");
    }
}
// Output: Area of the square = 144
```

Une méthode `override` fournit une nouvelle implémentation d'une méthode héritée d'une classe de base. La méthode substituée par une déclaration `override` est appelée méthode de base substituée. Une méthode `override` doit avoir la même signature que la méthode `override` de base substituée. Les méthodes `override` prennent en charge les types de retour covariants. En particulier, le type de retour d'une méthode `override` peut dériver du type de retour de la méthode de base correspondante.

Vous ne pouvez pas surcharger une méthode statique ou non virtuelle. La méthode de base substituée doit être `virtual`, `abstract` ou `override`.

Une déclaration `override` ne peut pas changer l'accessibilité de la méthode `virtual`. Les deux méthodes `override` et `virtual` doivent avoir le même modificateur de niveau d'accès.

Vous ne pouvez pas utiliser les modificateurs `new`, `static` ou `virtual` pour modifier une méthode `override`.

Une déclaration de propriété de substitution doit spécifier exactement les mêmes modificateur d'accès, type et nom que la propriété héritée. À compter de C# 9.0, les propriétés de substitution en lecture seule prennent en charge les types de retour covariants. La propriété substituée doit être `virtual`, `abstract` ou `override`.

Pour plus d'informations sur l'utilisation du mot clé `override`, consultez Gestion de version avec les mots clés override et new et Savoir quand utiliser les mots clés override et new. Pour plus d'informations sur l'héritage, consultez Héritage.

Exemple

Cet exemple définit une classe de base nommée `Employee` et une classe dérivée nommée `SalesEmployee`. La classe `SalesEmployee` inclut un champ supplémentaire (`salesbonus`) et substitue la méthode `CalculatePay` afin de la prendre en compte.

C#

```
class TestOverride
{
    public class Employee
    {
        public string Name { get; }

        // Basepay is defined as protected, so that it may be
        // accessed only by this class and derived classes.
        protected decimal _basepay;

        // Constructor to set the name and basepay values.
        public Employee(string name, decimal basepay)
        {
            Name = name;
            _basepay = basepay;
        }

        // Declared virtual so it can be overridden.
        public virtual decimal CalculatePay()
        {
            return _basepay;
        }
    }
```

```
    // Derive a new class from Employee.
    public class SalesEmployee : Employee
    {
        // New field that will affect the base pay.
        private decimal _salesbonus;

        // The constructor calls the base-class version, and
        // initializes the salesbonus field.
        public SalesEmployee(string name, decimal basepay, decimal
salesbonus)
            : base(name, basepay)
        {
            _salesbonus = salesbonus;
        }

        // Override the CalculatePay method
        // to take bonus into account.
        public override decimal CalculatePay()
        {
            return _basepay + _salesbonus;
        }
    }

    static void Main()
    {
        // Create some new employees.
        var employee1 = new SalesEmployee("Alice", 1000, 500);
        var employee2 = new Employee("Bob", 1200);

        Console.WriteLine($"Employee1 {employee1.Name} earned:
{employee1.CalculatePay()}");
        Console.WriteLine($"Employee2 {employee2.Name} earned:
{employee2.CalculatePay()}");
    }
}
/*
    Output:
    Employee1 Alice earned: 1500
    Employee2 Bob earned: 1200
*/
```

spécification du langage C#

Pour plus d'informations, consultez la section Méthodes de substitution de la spécification du langage C#.

Pour plus d'informations sur les types de retour covariants, voir la note de proposition de fonctionnalité.

Voir aussi

- Informations de référence sur C#
- Héritage
- Mots clés C#
- Modificateurs
- abstract
- virtual
- new (modificateur)
- Polymorphisme

readonly (référence C#)

Article • 07/04/2023

Le mot clé `readonly` est un modificateur qui peut être utilisé dans quatre contextes :

- Dans une déclaration de champ, `readonly` indique qu'une affectation à destination d'un champ peut survenir uniquement dans le cadre de la déclaration ou dans un constructeur de la même classe. Un champ en lecture seule peut être affecté et réaffecté plusieurs fois dans la déclaration de champ et le constructeur.

  Un champ `readonly` ne peut pas être affecté après l'arrêt du constructeur. Cette règle a des implications différentes selon les types de valeur et les types de référence :
  - Étant donné que les types de valeur contiennent directement leurs données, un champ qui est un type de valeur `readonly` est immuable.
  - Étant donné que les types de référence contiennent une référence à leurs données, un champ qui est un type de référence `readonly` doit toujours faire référence au même objet. Cet objet n'est pas immuable. Le modificateur `readonly` empêche le champ d'être remplacé par une autre instance du type de référence. Toutefois, le modificateur n'empêche pas les données d'instance du champ d'être modifiées par le champ en lecture seule.

  > ⚠ **Avertissement**
  >
  > Un type visible de l'extérieur qui contient un champ en lecture seule visible de l'extérieur qui est un type de référence mutable peut être une faille de sécurité et peut déclencher l'avertissement **CA2104** : « Ne pas déclarer de type référence mutable en lecture seule. »

- Dans une `readonly struct` définition de type, `readonly` indique que le type de structure est immuable. Pour plus d'informations, voir la section readonly struct de l'article Types de structure.

- Dans une déclaration de membre d'instance dans un type de structure, `readonly` indique qu'un membre d'instance ne modifie pas l'état de la structure. Pour plus d'informations, voir la section readonly membres d'instance de l'article Types de structure.

- Dans un ref readonlyretour de la méthode, le modificateur `readonly` indique que la méthode retourne une référence et que les écritures ne sont pas autorisés pour

cette référence.

Exemple de champ en lecture seule

Dans cet exemple, la valeur du champ `year` ne peut pas être modifiée dans la méthode `ChangeYear`, même si une valeur lui est affectée dans le constructeur de classe :

```C#
class Age
{
    private readonly int _year;
    Age(int year)
    {
        _year = year;
    }
    void ChangeYear()
    {
        //_year = 1967; // Compile error if uncommented.
    }
}
```

Vous pouvez affecter une valeur à un champ `readonly` uniquement dans les contextes suivants :

- Lorsque la variable est initialisée dans la déclaration, par exemple :

  ```C#
  public readonly int y = 5;
  ```

- Dans un constructeur d'instance de la classe qui contient la déclaration de champ d'instance.

- Dans le constructeur statique de la classe qui contient la déclaration de champ statique.

Ces contextes de constructeur sont aussi les seuls contextes dans lesquels il est possible de passer un champ `readonly` comme paramètre out ou ref.

> ⓘ **Notes**
>
> Le mot clé `readonly` est différent du mot clé **const**. Un champ `const` ne peut être initialisé qu'au moment de la déclaration du champ. Un champ `readonly` peut être assigné plusieurs fois dans la déclaration de champ et dans un constructeur. C'est

pourquoi, les champs `readonly` peuvent avoir des valeurs différentes en fonction du constructeur utilisé. De même, alors qu'un champ `const` est une constante au moment de la compilation, le champ `readonly` peut être utilisé pour des constantes au moment de l'exécution, comme dans l'exemple suivant :

C#

```
public static readonly uint timeStamp = (uint)DateTime.Now.Ticks;
```

C#

```
public class SamplePoint
{
    public int x;
    // Initialize a readonly field
    public readonly int y = 25;
    public readonly int z;

    public SamplePoint()
    {
        // Initialize a readonly instance field
        z = 24;
    }

    public SamplePoint(int p1, int p2, int p3)
    {
        x = p1;
        y = p2;
        z = p3;
    }

    public static void Main()
    {
        SamplePoint p1 = new SamplePoint(11, 21, 32);   // OK
        Console.WriteLine($"p1: x={p1.x}, y={p1.y}, z={p1.z}");
        SamplePoint p2 = new SamplePoint();
        p2.x = 55;   // OK
        Console.WriteLine($"p2: x={p2.x}, y={p2.y}, z={p2.z}");
    }
    /*
     Output:
        p1: x=11, y=21, z=32
        p2: x=55, y=25, z=24
    */
}
```

Dans l'exemple précédent, si vous utilisez une instruction telle que dans l'exemple suivant :

```
C#

p2.y = 66;        // Error
```

vous obtenez le message d'erreur du compilateur :

Un champ readonly ne peut pas être assigné (sauf s'il appartient à un constructeur ou un initialiseur de variable)

Exemple de retour ref readonly

Le modificateur `readonly` au niveau d'un `ref return` indique que la référence retournée ne peut pas être modifiée. L'exemple suivant retourne une référence à l'origine. Il utilise le modificateur `readonly` pour indiquer que les appelants ne peuvent pas modifier l'origine :

```
C#

private static readonly SamplePoint s_origin = new SamplePoint(0, 0, 0);
public static ref readonly SamplePoint Origin => ref s_origin;
```

Le type retourné ne doit pas nécessairement être un `readonly struct`. Tout type pouvant être retourné par `ref` peut être retourné par `ref readonly`.

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Vous pouvez également voir les propositions de spécification de langage :

- readonly ref et readonly struct
- membres readonly struct

Voir aussi

- Ajouter un modificateur readonly (règle de style IDE0044)
- Référence C#
- Guide de programmation C#
- Mots clés C#
- Modificateurs

- const
- Fields

sealed (référence C#)

Article • 07/04/2023

Lorsqu'il est appliqué à une classe, le modificateur `sealed` empêche les autres classes d'en hériter. Dans l'exemple suivant, la classe `B` hérite de la classe `A`, mais aucune classe ne peut hériter de la classe `B`.

C#

```
class A {}
sealed class B : A {}
```

Vous pouvez également utiliser le modificateur `sealed` sur une méthode ou une propriété qui substitue une méthode ou une propriété virtuelle dans une classe de base. Ainsi, vous pouvez autoriser les classes à dériver de votre classe et les empêcher de substituer des méthodes ou des propriétés virtuelles spécifiques.

Exemple

Dans l'exemple suivant, `Z` hérite de `Y` mais `Z` ne peut pas substituer la fonction virtuelle `F` qui est déclarée dans `X` et scellée (sealed) dans `Y`.

C#

```
class X
{
    protected virtual void F() { Console.WriteLine("X.F"); }
    protected virtual void F2() { Console.WriteLine("X.F2"); }
}

class Y : X
{
    sealed protected override void F() { Console.WriteLine("Y.F"); }
    protected override void F2() { Console.WriteLine("Y.F2"); }
}

class Z : Y
{
    // Attempting to override F causes compiler error CS0239.
    // protected override void F() { Console.WriteLine("Z.F"); }

    // Overriding F2 is allowed.
    protected override void F2() { Console.WriteLine("Z.F2"); }
}
```

Lorsque vous définissez de nouvelles méthodes ou propriétés dans une classe, vous pouvez empêcher les classes dérivées de les substituer en ne les déclarant pas comme virtuelles.

Une erreur consiste à utiliser le modificateur abstract avec une classe sealed, car une classe abstraite doit être héritée par une classe qui fournit une implémentation des méthodes ou des propriétés abstraites.

Lorsqu'il est appliqué à une méthode ou une propriété, le modificateur `sealed` doit toujours être utilisé avec override.

Comme les structs sont implicitement sealed, ils ne peuvent pas être hérités.

Pour plus d'informations, consultez Héritage.

Pour plus d'exemples, consultez Classes abstract et sealed et membres de classe.

C#

```
sealed class SealedClass
{
    public int x;
    public int y;
}

class SealedTest2
{
    static void Main()
    {
        var sc = new SealedClass();
        sc.x = 110;
        sc.y = 150;
        Console.WriteLine($"x = {sc.x}, y = {sc.y}");
    }
}
// Output: x = 110, y = 150
```

Dans l'exemple précédent, vous pouvez essayer d'hériter de la classe sealed à l'aide de l'instruction suivante :

```
class MyDerivedC: SealedClass {} // Error
```

Le résultat est un message d'erreur :

```
'MyDerivedC': cannot derive from sealed type 'SealedClass'
```

Notes

Pour déterminer s'il faut sceller une classe, une méthode ou une propriété, vous devez généralement prendre en compte les deux points suivants :

- Les avantages potentiels dont les classes dérivées peuvent bénéficier grâce à la possibilité de personnaliser votre classe.
- Le risque que les classes dérivées puissent modifier vos classes de telle manière qu'elles ne fonctionnent plus correctement ou comme prévu.

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Classes statiques et membres de classe statique
- Classes abstract et sealed et membres de classe
- Modificateurs d'accès
- Modificateurs
- override
- virtual

static (référence C#)

Article • 07/04/2023

Cette page couvre le mot clé `static` du modificateur. Le mot clé `static` fait également partie de la directive using static.

Utilisez le modificateur `static` pour déclarer un membre statique, qui appartient au type lui-même plutôt qu'à un objet spécifique. Le modificateur `static` peut être utilisé pour déclarer des classes `static`. Dans les classes, interfaces et structs, vous pouvez ajouter le modificateur `static` aux champs, méthodes, propriétés, opérateurs, événements et constructeurs. Le modificateur `static` ne peut pas être utilisé avec des indexeurs ou des finaliseurs. Pour plus d'informations, consultez la page Classes statiques et membres de classes statiques.

Vous pouvez ajouter le modificateur `static` à une fonction locale. Une fonction locale statique ne peut pas capturer les variables locales ou l'état de l'instance.

À compter de C# 9.0, vous pouvez ajouter le modificateur `static` à une expression lambda ou méthode anonyme. Une méthode lambda ou anonyme statique ne peut pas capturer les variables locales ou l'état de l'instance.

Exemple : classe statique

La classe suivante est déclarée comme `static` et contient uniquement des méthodes `static` :

C#

```
static class CompanyEmployee
{
    public static void DoSomething() { /*...*/ }
    public static void DoSomethingElse() { /*...*/  }
}
```

Une constante ou une déclaration de type est implicitement un membre `static`. Un membre `static` ne peut pas être référencé via une instance. Il sera référencé par le nom du type. Par exemple, considérons la classe suivante :

C#

```
public class MyBaseC
{
```

```
    public struct MyStruct
    {
        public static int x = 100;
    }
}
```

Pour faire référence au `static` membre `x`, utilisez le nom complet, `MyBaseC.MyStruct.x`, sauf si le membre est accessible à partir de la même étendue :

```
C#

Console.WriteLine(MyBaseC.MyStruct.x);
```

Alors qu'une instance d'une classe contient une copie distincte de tous les champs d'instance de la classe, il existe une seule copie de chaque champ `static`.

Il n'est pas possible d'utiliser this pour référencer des méthodes `static` ou des accesseurs de propriétés.

Si le mot clé `static` est appliqué à une classe, tous les membres de la classe doivent être `static`.

Les classes, interfaces et classes `static` peuvent avoir des constructeurs `static`. Un constructeur `static` est appelé à un moment donné entre le démarrage du programme et l'instanciation de la classe.

> ⓘ **Notes**
>
> L'utilisation du mot clé `static` est plus restreinte que dans C++. Pour comparer avec le mot clé C++, consultez **Classes de stockage (C++)**.

Pour illustrer membres `static`, considérez une classe qui représente un employé de l'entreprise. Supposons que la classe contient une méthode pour compter les employés et un champ pour stocker le nombre d'employés. La méthode et le champ n'appartiennent à aucune instance d'employé en particulier. Ils appartiennent à la classe des employés dans son ensemble. Ils doivent être déclarés en tant que membres `static` de la classe.

Exemple : champ statique et méthode

Cet exemple lit le nom et l'ID d'un nouvel employé, incrémente d'une unité le compteur d'employés et affiche les informations concernant le nouvel employé et le nouveau

nombre d'employés. Ce programme lit le nombre actuel d'employés à partir du clavier.

C#

```
public class Employee4
{
    public string id;
    public string name;

    public Employee4()
    {
    }

    public Employee4(string name, string id)
    {
        this.name = name;
        this.id = id;
    }

    public static int employeeCounter;

    public static int AddEmployee()
    {
        return ++employeeCounter;
    }
}

class MainClass : Employee4
{
    static void Main()
    {
        Console.Write("Enter the employee's name: ");
        string name = Console.ReadLine();
        Console.Write("Enter the employee's ID: ");
        string id = Console.ReadLine();

        // Create and configure the employee object.
        Employee4 e = new Employee4(name, id);
        Console.Write("Enter the current number of employees: ");
        string n = Console.ReadLine();
        Employee4.employeeCounter = Int32.Parse(n);
        Employee4.AddEmployee();

        // Display the new information.
        Console.WriteLine($"Name: {e.name}");
        Console.WriteLine($"ID:   {e.id}");
        Console.WriteLine($"New Number of Employees:
{Employee4.employeeCounter}");
    }
}
/*
Input:
Matthias Berndt
AF643G
```

```
15
 *
Sample Output:
Enter the employee's name: Matthias Berndt
Enter the employee's ID: AF643G
Enter the current number of employees: 15
Name: Matthias Berndt
ID:   AF643G
New Number of Employees: 16
*/
```

Exemple : initialisation statique

Cet exemple montre que vous pouvez initialiser un champ `static` à l'aide d'un autre champ `static` qui n'est pas encore déclaré. Les résultats ne seront pas définis jusqu'à ce que vous affectiez explicitement une valeur au champ `static`.

C#

```
class Test
{
    static int x = y;
    static int y = 5;

    static void Main()
    {
        Console.WriteLine(Test.x);
        Console.WriteLine(Test.y);

        Test.x = 99;
        Console.WriteLine(Test.x);
    }
}
/*
Output:
    0
    5
    99
*/
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Modificateurs
- utilisation de la directive static
- Classes statiques et membres de classe statique

unsafe (référence C#)

Article • 07/04/2023

Le mot clé `unsafe` désigne un contexte non sécurisé, qui est requis pour toute opération impliquant des pointeurs. Pour plus d'informations, consultez Pointeurs et code unsafe.

Vous pouvez utiliser le modificateur `unsafe` dans la déclaration d'un type ou d'un membre. Toute l'étendue de texte du type ou du membre est ainsi considérée comme un contexte unsafe. Par exemple, ce qui suit est une méthode déclarée avec le modificateur `unsafe` :

C#

```
unsafe static void FastCopy(byte[] src, byte[] dst, int count)
{
    // Unsafe context: can use pointers here.
}
```

La portée du contexte unsafe s'étend de la liste de paramètres à la fin de la méthode, de sorte que les pointeurs peuvent également être utilisés dans la liste de paramètres :

C#

```
unsafe static void FastCopy ( byte* ps, byte* pd, int count ) {...}
```

Vous pouvez également avoir recours à un bloc unsafe pour utiliser un code unsafe dans ce bloc. Par exemple :

C#

```
unsafe
{
    // Unsafe context: can use pointers here.
}
```

Pour compiler du code unsafe, vous devez spécifier l'option de compilateur **AllowUnsafeBlocks**. Le code unsafe n'est pas vérifiable par le service CLR (Common Language Runtime).

Exemple

C#

```
// compile with: -unsafe
class UnsafeTest
{
    // Unsafe method: takes pointer to int.
    unsafe static void SquarePtrParam(int* p)
    {
        *p *= *p;
    }

    unsafe static void Main()
    {
        int i = 5;
        // Unsafe method: uses address-of operator (&).
        SquarePtrParam(&i);
        Console.WriteLine(i);
    }
}
// Output: 25
```

spécification du langage C#

Pour plus d'informations, voir Code unsafe dans la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Informations de référence sur C#
- Mots clés C#
- instruction fixed
- Code, types de pointeurs et pointeurs de fonction non sécurisés
- Opérateurs associés au pointeur

virtual (référence C#)

Article • 07/04/2023

Le mot clé `virtual` sert à modifier une méthode, une propriété, un indexeur ou une déclaration event et leur permet d'être substitués dans une classe dérivée. Par exemple, cette méthode peut être substituée par toute classe qui en hérite :

C#

```
public virtual double Area()
{
    return x * y;
}
```

L'implémentation d'un membre virtuel peut être modifiée par un membre de substitution dans une classe dérivée. Pour plus d'informations sur l'utilisation du mot clé `virtual`, consultez Gestion de version avec les mots clés override et new et Savoir quand utiliser les mots clés override et new.

Remarques

Quand une méthode virtuelle est appelée, un membre de substitution est recherché dans le type d'objet au moment de l'exécution. Le membre de substitution de la classe la plus dérivée est appelé (cela peut être le membre d'origine), si aucune classe dérivée n'a substitué le membre.

Par défaut, les méthodes ne sont pas virtuelles. Vous ne pouvez pas substituer une méthode non virtuelle.

Vous ne pouvez pas utiliser le modificateur `virtual` avec les modificateurs `static`, `abstract`, `private` ou `override`. L'exemple suivant illustre une propriété virtuelle :

C#

```
class MyBaseClass
{
    // virtual auto-implemented property. Overrides can only
    // provide specialized behavior if they implement get and set accessors.
    public virtual string Name { get; set; }

    // ordinary virtual property with backing field
    private int _num;
    public virtual int Number
    {
```

```
        get { return _num; }
        set { _num = value; }
    }
}

class MyDerivedClass : MyBaseClass
{
    private string _name;

    // Override auto-implemented property with ordinary property
    // to provide specialized accessor behavior.
    public override string Name
    {
        get
        {
            return _name;
        }
        set
        {
            if (!string.IsNullOrEmpty(value))
            {
                _name = value;
            }
            else
            {
                _name = "Unknown";
            }
        }
    }
}
```

Les propriétés virtuelles se comportent comme les méthodes virtuelles, à l'exception des différences dans la syntaxe de déclaration et d'appel.

- L'utilisation du modificateur `virtual` sur une propriété statique serait une erreur.
- Une propriété virtuelle héritée peut être substituée dans une classe dérivée en incluant une déclaration de propriété qui utilise le modificateur `override`.

Exemple

Dans cet exemple, la classe `Shape` contient les deux coordonnées `x` et `y`, ainsi que la méthode virtuelle `Area()`. Différentes classes de formes, telles que `Circle`, `Cylinder` et `Sphere`, héritent de la classe `Shape`, et la surface est calculée pour chaque figure. Chaque classe dérivée a sa propre implémentation de substitution de `Area()`.

Notez que les classes héritées `Circle`, `Sphere` et `Cylinder` utilisent toutes des constructeurs qui initialisent la classe de base, comme indiqué dans la déclaration

suivante.

C#

```
public Cylinder(double r, double h): base(r, h) {}
```

Le programme suivant calcule et affiche la zone appropriée pour chaque figure en appelant l'implémentation appropriée de la méthode `Area()`, selon l'objet associé à la méthode.

C#

```
class TestClass
{
    public class Shape
    {
        public const double PI = Math.PI;
        protected double _x, _y;

        public Shape()
        {
        }

        public Shape(double x, double y)
        {
            _x = x;
            _y = y;
        }

        public virtual double Area()
        {
            return _x * _y;
        }
    }

    public class Circle : Shape
    {
        public Circle(double r) : base(r, 0)
        {
        }

        public override double Area()
        {
            return PI * _x * _x;
        }
    }

    public class Sphere : Shape
    {
        public Sphere(double r) : base(r, 0)
        {
        }
```

```
        public override double Area()
        {
            return 4 * PI * _x * _x;
        }
    }

    public class Cylinder : Shape
    {
        public Cylinder(double r, double h) : base(r, h)
        {
        }

        public override double Area()
        {
            return 2 * PI * _x * _x + 2 * PI * _x * _y;
        }
    }

    static void Main()
    {
        double r = 3.0, h = 5.0;
        Shape c = new Circle(r);
        Shape s = new Sphere(r);
        Shape l = new Cylinder(r, h);
        // Display results.
        Console.WriteLine("Area of Circle   = {0:F2}", c.Area());
        Console.WriteLine("Area of Sphere   = {0:F2}", s.Area());
        Console.WriteLine("Area of Cylinder = {0:F2}", l.Area());
    }
}
/*
Output:
Area of Circle   = 28.27
Area of Sphere   = 113.10
Area of Cylinder = 150.80
*/
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Polymorphisme
- abstract
- override
- new (modificateur)

volatile (référence C#)

Article • 07/04/2023

Le mot clé `volatile` indique qu'un champ peut être modifié par plusieurs threads qui s'exécutent simultanément. Le compilateur, le système de runtime et même le matériel sont susceptibles de réorganiser les lectures et les écritures sur des emplacements de mémoire pour des raisons de performances. Les champs déclarés `volatile` sont exclus de certains types d'optimisations. Rien ne garantit que les écritures volatiles présentent un ordre total unique, tel que le voient tous les threads d'exécution. Pour plus d'informations, consultez la classe Volatile.

> **Notes**
>
> Sur un système multiprocesseur, une opération de lecture volatile ne garantit pas d'obtenir la dernière valeur écrite à cet emplacement de mémoire par n'importe quel processeur. De même, une opération d'écriture volatile ne garantit pas que la valeur écrite serait immédiatement visible par d'autres processeurs.

Le mot clé `volatile` peut être appliqué aux champs des types suivants :

- Types référence.
- Types pointeur (dans un contexte unsafe). Notez que, même si le pointeur lui-même peut être volatile, l'objet sur lequel il pointe ne le peut pas. En d'autres termes, vous ne pouvez pas déclarer un pointeur vers un objet volatile.
- Types simples comme `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `char`, `float` et `bool`.
- Type `enum` avec l'un des types de base suivants : `byte`, `sbyte`, `short`, `ushort`, `int` ou `uint`.
- Paramètres de type générique connus comme des types référence.
- Voir IntPtr et UIntPtr.

Les autres types, notamment `double` et `long`, ne peuvent pas être marqués `volatile`, car il n'y a aucune garantie que les lectures et écritures sur des champs de ce type soient atomiques. Pour protéger l'accès multithread à ces types de champs, utilisez les membres de classe Interlocked ou l'instruction lock.

Le mot clé `volatile` ne peut s'appliquer qu'aux champs d'une `class` ou d'un `struct`. Les variables locales ne peuvent pas être déclarées `volatile`.

Exemple

L'exemple ci-dessous montre comment déclarer une variable de champ public comme `volatile`.

C#

```
class VolatileTest
{
    public volatile int sharedStorage;

    public void Test(int i)
    {
        sharedStorage = i;
    }
}
```

L'exemple suivant montre comment il est possible de créer un thread auxiliaire ou de travail et de l'utiliser pour effectuer le traitement en parallèle avec le thread principal. Pour plus d'informations sur le multithreading, voir Threading managé.

C#

```
public class Worker
{
    // This method is called when the thread is started.
    public void DoWork()
    {
        bool work = false;
        while (!_shouldStop)
        {
            work = !work; // simulate some work
        }
        Console.WriteLine("Worker thread: terminating gracefully.");
    }
    public void RequestStop()
    {
        _shouldStop = true;
    }
    // Keyword volatile is used as a hint to the compiler that this data
    // member is accessed by multiple threads.
    private volatile bool _shouldStop;
}

public class WorkerThreadExample
{
    public static void Main()
    {
        // Create the worker thread object. This does not start the thread.
        Worker workerObject = new Worker();
```

```
        Thread workerThread = new Thread(workerObject.DoWork);

        // Start the worker thread.
        workerThread.Start();
        Console.WriteLine("Main thread: starting worker thread...");

        // Loop until the worker thread activates.
        while (!workerThread.IsAlive)
            ;

        // Put the main thread to sleep for 500 milliseconds to
        // allow the worker thread to do some work.
        Thread.Sleep(500);

        // Request that the worker thread stop itself.
        workerObject.RequestStop();

        // Use the Thread.Join method to block the current thread
        // until the object's thread terminates.
        workerThread.Join();
        Console.WriteLine("Main thread: worker thread has terminated.");
    }
    // Sample output:
    // Main thread: starting worker thread...
    // Worker thread: terminating gracefully.
    // Main thread: worker thread has terminated.
}
```

Si vous ajoutez le modificateur `volatile` à la déclaration de `_shouldStop` en place, vous obtiendrez toujours les mêmes résultats (similaires à l'extrait indiqué dans le code précédent). Sans ce modificateur sur le membre `_shouldStop` en revanche, le comportement est imprévisible. La méthode `DoWork` peut optimiser l'accès au membre, ce qui entraîne la lecture de données périmées. En raison de la nature de la programmation multithread, le nombre de lectures obsolètes est imprévisible. Différentes exécutions du programme produiront des résultats légèrement différents.

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Spécification du langage C# : mot clé volatile
- Référence C#
- Guide de programmation C#

- Mots clés C#
- Modificateurs
- lock, instruction
- Interlocked

Mots clés d'instruction (Référence C#)

Article • 07/04/2023

Les instructions sont des instructions de programme. Sauf mention contraire dans les rubriques référencées dans le tableau suivant, les instructions sont exécutées en séquence. Le tableau suivant répertorie les mots clés d'instructions C#. Pour plus d'informations sur les instructions exprimées sans mots clés, consultez Instructions.

Category	Mots clés C#
Instructions de sélection	`if`, `switch`
Instructions d'itération	`do`, `for`, `foreach`, `while`
Instructions de saut	`break`, `continue`, `goto`, `return`
Instructions de gestion des exceptions	throw, try-catch, try-finally, try-catch-finally
Instructions checked et unchecked	`checked`, `unchecked`
fixed, instruction	`fixed`
lock, instruction	`lock`
Instruction yield	`yield`

Voir aussi

- Référence C#
- Instructions
- Mots clés C#

Instructions de gestion des exceptions : `throw`, `try-catch`, `try-finally` et `try-catch-finally`

Article • 05/06/2023

Vous utilisez les instructions `throw` et `try` pour travailler avec des exceptions. Utilisez l'instruction throw pour lever une exception. Utilisez l'instruction try pour intercepter et gérer les exceptions qui peuvent se produire pendant l'exécution d'un bloc de code.

Instruction `throw`

L'instruction `throw` lève une exception :

```
C#

if (shapeAmount <= 0)
{
    throw new ArgumentOutOfRangeException(nameof(shapeAmount), "Amount of
shapes must be positive.");
}
```

Dans une instruction `throw e;`, le résultat de l'expression `e` doit être implicitement convertible en System.Exception.

Vous pouvez utiliser les classes d'exception intégrées, par exemple ArgumentOutOfRangeException ou InvalidOperationException. .NET fournit également les méthodes d'assistance pour lever des exceptions dans certaines conditions : ArgumentNullException.ThrowIfNull et ArgumentException.ThrowIfNullOrEmpty. Vous pouvez également définir vos propres classes d'exception qui dérivent de System.Exception. Pour plus d'informations, consultez Création et levée d'exceptions.

À l'intérieur d'un bloc catch, vous pouvez utiliser une instruction `throw;` pour lever à nouveau l'exception gérée par le bloc `catch` :

```
C#

try
{
    ProcessShapes(shapeAmount);
}
catch (Exception e)
```

```
{
    LogError(e, "Shape processing failed.");
    throw;
}
```

> ⓘ Notes
>
> `throw;` conserve la trace de pile d'origine de l'exception, qui est stockée dans la propriété **Exception.StackTrace**. À l'inverse, `throw e;` met à jour la propriété **StackTrace** de `e`.

Quand une exception est levée, le Common Language Runtime (CLR) recherche le bloc catch qui gère cette exception. Si la méthode exécutée ne contient pas un tel bloc `catch`, le CLR examine la méthode qui a appelé la méthode actuelle, puis remonte la pile des appels. Si aucun bloc `catch` n'est trouvé, le CLR met fin au thread en cours d'exécution. Pour plus d'informations, consultez la section Comment les exceptions sont gérées de la spécification du langage C#.

Expression `throw`

Vous pouvez également utiliser `throw` comme expression. Cela peut être pratique dans un certain nombre de cas, notamment :

- l'opérateur conditionnel : L'exemple suivant utilise une expression `throw` pour lever un ArgumentException lorsque le tableau `args` passé est vide :

  C#

  ```
  string first = args.Length >= 1
      ? args[0]
      : throw new ArgumentException("Please supply at least one
  argument.");
  ```

- l'opérateur de fusion de Null : L'exemple suivant utilise une expression `throw` pour lever un ArgumentNullException lorsque la chaîne à affecter à une propriété est `null` :

  C#

  ```
  public string Name
  {
      get => name;
      set => name = value ??
  ```

```
        throw new ArgumentNullException(paramName: nameof(value),
message: "Name cannot be null");
}
```

- un lambda ou une méthode expression-bodied : L'exemple suivant utilise une expression `throw` pour lever un InvalidCastException pour indiquer qu'une conversion en valeur DateTime n'est pas prise en charge :

```
C#

DateTime ToDateTime(IFormatProvider provider) =>
         throw new InvalidCastException("Conversion to a DateTime is
not supported.");
```

Instruction `try`

Vous pouvez utiliser l'instruction `try` sous l'une des formes suivantes : try-catch - pour gérer les exceptions qui peuvent se produire pendant l'exécution du code à l'intérieur d'un bloc `try`, try-finally - pour spécifier le code qui est exécuté lorsque le contrôle quitte le bloc `try` et try-catch-finally - en combinaison des deux formulaires précédents.

Instruction `try-catch`

Utilisez l'instruction `try-catch` pour intercepter et gérer les exceptions qui peuvent se produire pendant l'exécution d'un bloc de code. Placez le code là où une exception peut se produire à l'intérieur d'un bloc `try`. Utilisez une *clause catch* pour spécifier le type de base des exceptions que vous souhaitez gérer dans le bloc `catch` correspondant :

```
C#

try
{
    var result = Process(-3, 4);
    Console.WriteLine($"Processing succeeded: {result}");
}
catch (ArgumentException e)
{
    Console.WriteLine($"Processing failed: {e.Message}");
}
```

Vous pouvez fournir plusieurs clauses catch :

```
C#
```

```
try
{
    var result = await ProcessAsync(-3, 4, cancellationToken);
    Console.WriteLine($"Processing succeeded: {result}");
}
catch (ArgumentException e)
{
    Console.WriteLine($"Processing failed: {e.Message}");
}
catch (OperationCanceledException)
{
    Console.WriteLine("Processing is cancelled.");
}
```

Lorsqu'une exception se produit, les clauses catch sont examinées dans l'ordre spécifié, de haut en bas. Au maximum, un seul bloc `catch` est exécuté pour toute exception levée. Comme le montre également l'exemple précédent, vous pouvez omettre la déclaration d'une variable d'exception et spécifier uniquement le type d'exception dans une clause catch. Une clause catch sans type d'exception spécifié correspond à n'importe quelle exception et, le cas échéant, doit être la dernière clause catch.

Si vous souhaitez lever à nouveau une exception interceptée, utilisez l'instruction throw, comme dans l'exemple suivant :

C#

```
try
{
    var result = Process(-3, 4);
    Console.WriteLine($"Processing succeeded: {result}");
}
catch (Exception e)
{
    LogError(e, "Processing failed.");
    throw;
}
```

> ⓘ **Notes**
>
> `throw;` conserve la trace de pile d'origine de l'exception, qui est stockée dans la propriété **Exception.StackTrace**. À l'inverse, `throw e;` met à jour la propriété **StackTrace** de `e`.

Un filtre d'exception `when`

En plus d'un type d'exception, vous pouvez également spécifier un filtre d'exception qui examine plus en détail une exception et détermine si le bloc `catch` correspondant gère cette exception. Un filtre d'exception est une expression booléenne qui suit le mot clé `when`, comme dans l'exemple suivant :

C#

```
try
{
    var result = Process(-3, 4);
    Console.WriteLine($"Processing succeeded: {result}");
}
catch (Exception e) when (e is ArgumentException || e is
DivideByZeroException)
{
    Console.WriteLine($"Processing failed: {e.Message}");
}
```

L'exemple précédent utilise un filtre d'exception pour fournir un seul bloc `catch` pour gérer les exceptions de deux types spécifiés.

Vous pouvez fournir plusieurs clauses `catch` pour le même type d'exception si elles se distinguent par des filtres d'exception. L'une de ces clauses peut ne pas avoir de filtre d'exception. Si une telle clause existe, il doit s'agir de la dernière des clauses qui spécifient ce type d'exception.

Si une clause `catch` a un filtre d'exception, elle peut spécifier le type d'exception qui est le même ou moins dérivé qu'un type d'exception d'une clause `catch` qui apparaît après elle. Par exemple, si un filtre d'exception est présent, une clause `catch (Exception e)` n'a pas besoin d'être la dernière clause.

Exceptions dans les méthodes asynchrones et itérateurs

Si une exception se produit dans une fonction asynchrone, elle se propage à l'appelant de la fonction lorsque vous attendez le résultat de la fonction, comme le montre l'exemple suivant :

C#

```
public static async Task Run()
{
    try
    {
        Task<int> processing = ProcessAsync(-1);
        Console.WriteLine("Launched processing.");
```

```
        int result = await processing;
        Console.WriteLine($"Result: {result}.");
    }
    catch (ArgumentException e)
    {
        Console.WriteLine($"Processing failed: {e.Message}");
    }
    // Output:
    // Launched processing.
    // Processing failed: Input must be non-negative. (Parameter 'input')
}

private static async Task<int> ProcessAsync(int input)
{
    if (input < 0)
    {
        throw new ArgumentOutOfRangeException(nameof(input), "Input must be
non-negative.");
    }

    await Task.Delay(500);
    return input;
}
```

Si une exception se produit dans une méthode d'itérateur, elle se propage à l'appelant uniquement lorsque l'itérateur passe à l'élément suivant.

Instruction `try-finally`

Dans une instruction `try-finally`, le bloc `finally` est exécuté lorsque le contrôle quitte le bloc `try`. Le contrôle peut quitter le bloc `try` à la suite de

- une exécution normale,
- l'exécution d'une instruction de saut (c'est-à-dire, `return`, `break`, `continue` ou `goto`), ou
- la propagation d'une exception hors du bloc `try`.

L'exemple suivant utilise le bloc `finally` pour réinitialiser l'état d'un objet avant que le contrôle quitte la méthode :

C#

```
public async Task HandleRequest(int itemId, CancellationToken ct)
{
    Busy = true;

    try
    {
        await ProcessAsync(itemId, ct);
```

```
    }
    finally
    {
        Busy = false;
    }
}
```

Vous pouvez également utiliser le bloc `finally` pour nettoyer les ressources allouées utilisées dans le bloc `try`.

> ⓘ Notes
>
> Lorsque le type d'une ressource implémente l'interface **IDisposable** ou **IAsyncDisposable**, considérez l'**instruction using**. L'instruction `using` garantit que les ressources acquises sont supprimées lorsque le contrôle quitte l'instruction `using`. Le compilateur transforme une instruction `using` en instruction `try-finally`.

Dans presque tous les cas, les blocs `finally` sont exécutés. Les seuls cas où les blocs `finally` ne sont pas exécutés impliquent l'arrêt immédiat d'un programme. Par exemple, un tel arrêt peut se produire en raison de l'appel Environment.FailFast ou d'une exception OverflowException ou InvalidProgramException. La plupart des systèmes d'exploitation effectuent un nettoyage raisonnable des ressources dans le cadre de l'arrêt et du déchargement du processus.

Instruction `try-catch-finally`

Vous utilisez une instruction `try-catch-finally` pour gérer les exceptions qui peuvent se produire pendant l'exécution du bloc `try` et pour spécifier le code qui doit être exécuté lorsque le contrôle quitte l'instruction `try` :

C#

```
public async Task ProcessRequest(int itemId, CancellationToken ct)
{
    Busy = true;

    try
    {
        await ProcessAsync(itemId, ct);
    }
    catch (Exception e) when (e is not OperationCanceledException)
    {
        LogError(e, $"Failed to process request for item ID {itemId}.");
```

```
            throw;
        }
        finally
        {
            Busy = false;
        }

    }
```

Lorsqu'une exception est gérée par un bloc `catch`, le bloc `finally` est exécuté après l'exécution de ce bloc `catch` (même si une autre exception se produit pendant l'exécution du bloc `catch`). Pour plus d'informations sur les blocs `catch` et `finally`, consultez respectivement les sections L'instruction try-catch et L'instructiontry-finally.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Instruction throw
- Instruction try
- Exceptions

Voir aussi

- Informations de référence sur C#
- Exceptions et gestion des exceptions
- Gestion et levée d'exceptions dans .NET
- préférences de levée (règle de style IDE0016)
- AppDomain.FirstChanceException
- AppDomain.UnhandledException
- TaskScheduler.UnobservedTaskException

Instructions de gestion des exceptions : `throw`, `try-catch`, `try-finally` et `try-catch-finally`

Article • 05/06/2023

Vous utilisez les instructions `throw` et `try` pour travailler avec des exceptions. Utilisez l'instruction throw pour lever une exception. Utilisez l'instruction try pour intercepter et gérer les exceptions qui peuvent se produire pendant l'exécution d'un bloc de code.

Instruction `throw`

L'instruction `throw` lève une exception :

```
C#

if (shapeAmount <= 0)
{
    throw new ArgumentOutOfRangeException(nameof(shapeAmount), "Amount of
shapes must be positive.");
}
```

Dans une instruction `throw e;`, le résultat de l'expression `e` doit être implicitement convertible en System.Exception.

Vous pouvez utiliser les classes d'exception intégrées, par exemple ArgumentOutOfRangeException ou InvalidOperationException. .NET fournit également les méthodes d'assistance pour lever des exceptions dans certaines conditions : ArgumentNullException.ThrowIfNull et ArgumentException.ThrowIfNullOrEmpty. Vous pouvez également définir vos propres classes d'exception qui dérivent de System.Exception. Pour plus d'informations, consultez Création et levée d'exceptions.

À l'intérieur d'un bloc catch, vous pouvez utiliser une instruction `throw;` pour lever à nouveau l'exception gérée par le bloc `catch` :

```
C#

try
{
    ProcessShapes(shapeAmount);
}
catch (Exception e)
```

```
{
    LogError(e, "Shape processing failed.");
    throw;
}
```

> ⓘ Notes
>
> `throw;` conserve la trace de pile d'origine de l'exception, qui est stockée dans la propriété **Exception.StackTrace**. À l'inverse, `throw e;` met à jour la propriété **StackTrace** de `e`.

Quand une exception est levée, le Common Language Runtime (CLR) recherche le bloc catch qui gère cette exception. Si la méthode exécutée ne contient pas un tel bloc `catch`, le CLR examine la méthode qui a appelé la méthode actuelle, puis remonte la pile des appels. Si aucun bloc `catch` n'est trouvé, le CLR met fin au thread en cours d'exécution. Pour plus d'informations, consultez la section Comment les exceptions sont gérées de la spécification du langage C#.

Expression `throw`

Vous pouvez également utiliser `throw` comme expression. Cela peut être pratique dans un certain nombre de cas, notamment :

- l'opérateur conditionnel : L'exemple suivant utilise une expression `throw` pour lever un ArgumentException lorsque le tableau `args` passé est vide :

  C#

  ```
  string first = args.Length >= 1
      ? args[0]
      : throw new ArgumentException("Please supply at least one
  argument.");
  ```

- l'opérateur de fusion de Null : L'exemple suivant utilise une expression `throw` pour lever un ArgumentNullException lorsque la chaîne à affecter à une propriété est `null` :

  C#

  ```
  public string Name
  {
      get => name;
      set => name = value ??
  ```

```
        throw new ArgumentNullException(paramName: nameof(value),
message: "Name cannot be null");
}
```

- un lambda ou une méthode expression-bodied : L'exemple suivant utilise une expression `throw` pour lever un InvalidCastException pour indiquer qu'une conversion en valeur DateTime n'est pas prise en charge :

```
C#

DateTime ToDateTime(IFormatProvider provider) =>
         throw new InvalidCastException("Conversion to a DateTime is
not supported.");
```

Instruction `try`

Vous pouvez utiliser l'instruction `try` sous l'une des formes suivantes : try-catch - pour gérer les exceptions qui peuvent se produire pendant l'exécution du code à l'intérieur d'un bloc `try`, try-finally - pour spécifier le code qui est exécuté lorsque le contrôle quitte le bloc `try` et try-catch-finally - en combinaison des deux formulaires précédents.

Instruction `try-catch`

Utilisez l'instruction `try-catch` pour intercepter et gérer les exceptions qui peuvent se produire pendant l'exécution d'un bloc de code. Placez le code là où une exception peut se produire à l'intérieur d'un bloc `try`. Utilisez une *clause catch* pour spécifier le type de base des exceptions que vous souhaitez gérer dans le bloc `catch` correspondant :

```
C#

try
{
    var result = Process(-3, 4);
    Console.WriteLine($"Processing succeeded: {result}");
}
catch (ArgumentException e)
{
    Console.WriteLine($"Processing failed: {e.Message}");
}
```

Vous pouvez fournir plusieurs clauses catch :

```
C#
```

```
try
{
    var result = await ProcessAsync(-3, 4, cancellationToken);
    Console.WriteLine($"Processing succeeded: {result}");
}
catch (ArgumentException e)
{
    Console.WriteLine($"Processing failed: {e.Message}");
}
catch (OperationCanceledException)
{
    Console.WriteLine("Processing is cancelled.");
}
```

Lorsqu'une exception se produit, les clauses catch sont examinées dans l'ordre spécifié, de haut en bas. Au maximum, un seul bloc `catch` est exécuté pour toute exception levée. Comme le montre également l'exemple précédent, vous pouvez omettre la déclaration d'une variable d'exception et spécifier uniquement le type d'exception dans une clause catch. Une clause catch sans type d'exception spécifié correspond à n'importe quelle exception et, le cas échéant, doit être la dernière clause catch.

Si vous souhaitez lever à nouveau une exception interceptée, utilisez l'instruction throw, comme dans l'exemple suivant :

C#

```
try
{
    var result = Process(-3, 4);
    Console.WriteLine($"Processing succeeded: {result}");
}
catch (Exception e)
{
    LogError(e, "Processing failed.");
    throw;
}
```

> ⓘ **Notes**
>
> `throw;` conserve la trace de pile d'origine de l'exception, qui est stockée dans la propriété **Exception.StackTrace**. À l'inverse, `throw e;` met à jour la propriété **StackTrace** de `e`.

Un filtre d'exception `when`

En plus d'un type d'exception, vous pouvez également spécifier un filtre d'exception qui examine plus en détail une exception et détermine si le bloc `catch` correspondant gère cette exception. Un filtre d'exception est une expression booléenne qui suit le mot clé `when`, comme dans l'exemple suivant :

C#

```
try
{
    var result = Process(-3, 4);
    Console.WriteLine($"Processing succeeded: {result}");
}
catch (Exception e) when (e is ArgumentException || e is
DivideByZeroException)
{
    Console.WriteLine($"Processing failed: {e.Message}");
}
```

L'exemple précédent utilise un filtre d'exception pour fournir un seul bloc `catch` pour gérer les exceptions de deux types spécifiés.

Vous pouvez fournir plusieurs clauses `catch` pour le même type d'exception si elles se distinguent par des filtres d'exception. L'une de ces clauses peut ne pas avoir de filtre d'exception. Si une telle clause existe, il doit s'agir de la dernière des clauses qui spécifient ce type d'exception.

Si une clause `catch` a un filtre d'exception, elle peut spécifier le type d'exception qui est le même ou moins dérivé qu'un type d'exception d'une clause `catch` qui apparaît après elle. Par exemple, si un filtre d'exception est présent, une clause `catch (Exception e)` n'a pas besoin d'être la dernière clause.

Exceptions dans les méthodes asynchrones et itérateurs

Si une exception se produit dans une fonction asynchrone, elle se propage à l'appelant de la fonction lorsque vous attendez le résultat de la fonction, comme le montre l'exemple suivant :

C#

```
public static async Task Run()
{
    try
    {
        Task<int> processing = ProcessAsync(-1);
        Console.WriteLine("Launched processing.");
```

```
        int result = await processing;
        Console.WriteLine($"Result: {result}.");
    }
    catch (ArgumentException e)
    {
        Console.WriteLine($"Processing failed: {e.Message}");
    }
    // Output:
    // Launched processing.
    // Processing failed: Input must be non-negative. (Parameter 'input')
}

private static async Task<int> ProcessAsync(int input)
{
    if (input < 0)
    {
        throw new ArgumentOutOfRangeException(nameof(input), "Input must be
non-negative.");
    }

    await Task.Delay(500);
    return input;
}
```

Si une exception se produit dans une méthode d'itérateur, elle se propage à l'appelant uniquement lorsque l'itérateur passe à l'élément suivant.

Instruction `try-finally`

Dans une instruction `try-finally`, le bloc `finally` est exécuté lorsque le contrôle quitte le bloc `try`. Le contrôle peut quitter le bloc `try` à la suite de

- une exécution normale,
- l'exécution d'une instruction de saut (c'est-à-dire, `return`, `break`, `continue` ou `goto`), ou
- la propagation d'une exception hors du bloc `try`.

L'exemple suivant utilise le bloc `finally` pour réinitialiser l'état d'un objet avant que le contrôle quitte la méthode :

C#

```
public async Task HandleRequest(int itemId, CancellationToken ct)
{
    Busy = true;

    try
    {
        await ProcessAsync(itemId, ct);
```

```
    }
    finally
    {
        Busy = false;
    }
}
```

Vous pouvez également utiliser le bloc `finally` pour nettoyer les ressources allouées utilisées dans le bloc `try`.

> ⓘ Notes
>
> Lorsque le type d'une ressource implémente l'interface **IDisposable** ou **IAsyncDisposable**, considérez l'**instruction using**. L'instruction `using` garantit que les ressources acquises sont supprimées lorsque le contrôle quitte l'instruction `using`. Le compilateur transforme une instruction `using` en instruction `try-finally`.

Dans presque tous les cas, les blocs `finally` sont exécutés. Les seuls cas où les blocs `finally` ne sont pas exécutés impliquent l'arrêt immédiat d'un programme. Par exemple, un tel arrêt peut se produire en raison de l'appel Environment.FailFast ou d'une exception OverflowException ou InvalidProgramException. La plupart des systèmes d'exploitation effectuent un nettoyage raisonnable des ressources dans le cadre de l'arrêt et du déchargement du processus.

Instruction `try-catch-finally`

Vous utilisez une instruction `try-catch-finally` pour gérer les exceptions qui peuvent se produire pendant l'exécution du bloc `try` et pour spécifier le code qui doit être exécuté lorsque le contrôle quitte l'instruction `try` :

C#

```
public async Task ProcessRequest(int itemId, CancellationToken ct)
{
    Busy = true;

    try
    {
        await ProcessAsync(itemId, ct);
    }
    catch (Exception e) when (e is not OperationCanceledException)
    {
        LogError(e, $"Failed to process request for item ID {itemId}.");
```

```
            throw;
        }
        finally
        {
            Busy = false;
        }

    }
```

Lorsqu'une exception est gérée par un bloc `catch`, le bloc `finally` est exécuté après l'exécution de ce bloc `catch` (même si une autre exception se produit pendant l'exécution du bloc `catch`). Pour plus d'informations sur les blocs `catch` et `finally`, consultez respectivement les sections L'instruction try-catch et L'instructiontry-finally.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Instruction throw
- Instruction try
- Exceptions

Voir aussi

- Informations de référence sur C#
- Exceptions et gestion des exceptions
- Gestion et levée d'exceptions dans .NET
- préférences de levée (règle de style IDE0016)
- AppDomain.FirstChanceException
- AppDomain.UnhandledException
- TaskScheduler.UnobservedTaskException

Instructions de gestion des exceptions : `throw`, `try-catch`, `try-finally` et `try-catch-finally`

Article • 05/06/2023

Vous utilisez les instructions `throw` et `try` pour travailler avec des exceptions. Utilisez l'instruction throw pour lever une exception. Utilisez l'instruction try pour intercepter et gérer les exceptions qui peuvent se produire pendant l'exécution d'un bloc de code.

Instruction `throw`

L'instruction `throw` lève une exception :

```
C#

if (shapeAmount <= 0)
{
    throw new ArgumentOutOfRangeException(nameof(shapeAmount), "Amount of
shapes must be positive.");
}
```

Dans une instruction `throw e;`, le résultat de l'expression `e` doit être implicitement convertible en System.Exception.

Vous pouvez utiliser les classes d'exception intégrées, par exemple ArgumentOutOfRangeException ou InvalidOperationException. .NET fournit également les méthodes d'assistance pour lever des exceptions dans certaines conditions : ArgumentNullException.ThrowIfNull et ArgumentException.ThrowIfNullOrEmpty. Vous pouvez également définir vos propres classes d'exception qui dérivent de System.Exception. Pour plus d'informations, consultez Création et levée d'exceptions.

À l'intérieur d'un bloc catch, vous pouvez utiliser une instruction `throw;` pour lever à nouveau l'exception gérée par le bloc `catch` :

```
C#

try
{
    ProcessShapes(shapeAmount);
}
catch (Exception e)
```

```
{
    LogError(e, "Shape processing failed.");
    throw;
}
```

> ⓘ Notes
>
> `throw;` conserve la trace de pile d'origine de l'exception, qui est stockée dans la propriété **Exception.StackTrace**. À l'inverse, `throw e;` met à jour la propriété **StackTrace** de `e`.

Quand une exception est levée, le Common Language Runtime (CLR) recherche le bloc catch qui gère cette exception. Si la méthode exécutée ne contient pas un tel bloc `catch`, le CLR examine la méthode qui a appelé la méthode actuelle, puis remonte la pile des appels. Si aucun bloc `catch` n'est trouvé, le CLR met fin au thread en cours d'exécution. Pour plus d'informations, consultez la section Comment les exceptions sont gérées de la spécification du langage C#.

Expression `throw`

Vous pouvez également utiliser `throw` comme expression. Cela peut être pratique dans un certain nombre de cas, notamment :

- l'opérateur conditionnel : L'exemple suivant utilise une expression `throw` pour lever un ArgumentException lorsque le tableau `args` passé est vide :

  C#

  ```
  string first = args.Length >= 1
      ? args[0]
      : throw new ArgumentException("Please supply at least one
  argument.");
  ```

- l'opérateur de fusion de Null : L'exemple suivant utilise une expression `throw` pour lever un ArgumentNullException lorsque la chaîne à affecter à une propriété est `null` :

  C#

  ```
  public string Name
  {
      get => name;
      set => name = value ??
  ```

```
        throw new ArgumentNullException(paramName: nameof(value),
message: "Name cannot be null");
}
```

- un lambda ou une méthode expression-bodied : L'exemple suivant utilise une expression `throw` pour lever un InvalidCastException pour indiquer qu'une conversion en valeur DateTime n'est pas prise en charge :

```
C#

DateTime ToDateTime(IFormatProvider provider) =>
         throw new InvalidCastException("Conversion to a DateTime is
not supported.");
```

Instruction `try`

Vous pouvez utiliser l'instruction `try` sous l'une des formes suivantes : try-catch - pour gérer les exceptions qui peuvent se produire pendant l'exécution du code à l'intérieur d'un bloc `try`, try-finally - pour spécifier le code qui est exécuté lorsque le contrôle quitte le bloc `try` et try-catch-finally - en combinaison des deux formulaires précédents.

Instruction `try-catch`

Utilisez l'instruction `try-catch` pour intercepter et gérer les exceptions qui peuvent se produire pendant l'exécution d'un bloc de code. Placez le code là où une exception peut se produire à l'intérieur d'un bloc `try`. Utilisez une *clause catch* pour spécifier le type de base des exceptions que vous souhaitez gérer dans le bloc `catch` correspondant :

```
C#

try
{
    var result = Process(-3, 4);
    Console.WriteLine($"Processing succeeded: {result}");
}
catch (ArgumentException e)
{
    Console.WriteLine($"Processing failed: {e.Message}");
}
```

Vous pouvez fournir plusieurs clauses catch :

```
C#
```

```
try
{
    var result = await ProcessAsync(-3, 4, cancellationToken);
    Console.WriteLine($"Processing succeeded: {result}");
}
catch (ArgumentException e)
{
    Console.WriteLine($"Processing failed: {e.Message}");
}
catch (OperationCanceledException)
{
    Console.WriteLine("Processing is cancelled.");
}
```

Lorsqu'une exception se produit, les clauses catch sont examinées dans l'ordre spécifié, de haut en bas. Au maximum, un seul bloc `catch` est exécuté pour toute exception levée. Comme le montre également l'exemple précédent, vous pouvez omettre la déclaration d'une variable d'exception et spécifier uniquement le type d'exception dans une clause catch. Une clause catch sans type d'exception spécifié correspond à n'importe quelle exception et, le cas échéant, doit être la dernière clause catch.

Si vous souhaitez lever à nouveau une exception interceptée, utilisez l'instruction throw, comme dans l'exemple suivant :

C#

```
try
{
    var result = Process(-3, 4);
    Console.WriteLine($"Processing succeeded: {result}");
}
catch (Exception e)
{
    LogError(e, "Processing failed.");
    throw;
}
```

> ⓘ **Notes**
>
> `throw;` conserve la trace de pile d'origine de l'exception, qui est stockée dans la propriété **Exception.StackTrace**. À l'inverse, `throw e;` met à jour la propriété **StackTrace** de `e`.

Un filtre d'exception `when`

En plus d'un type d'exception, vous pouvez également spécifier un filtre d'exception qui examine plus en détail une exception et détermine si le bloc `catch` correspondant gère cette exception. Un filtre d'exception est une expression booléenne qui suit le mot clé `when`, comme dans l'exemple suivant :

C#

```
try
{
    var result = Process(-3, 4);
    Console.WriteLine($"Processing succeeded: {result}");
}
catch (Exception e) when (e is ArgumentException || e is
DivideByZeroException)
{
    Console.WriteLine($"Processing failed: {e.Message}");
}
```

L'exemple précédent utilise un filtre d'exception pour fournir un seul bloc `catch` pour gérer les exceptions de deux types spécifiés.

Vous pouvez fournir plusieurs clauses `catch` pour le même type d'exception si elles se distinguent par des filtres d'exception. L'une de ces clauses peut ne pas avoir de filtre d'exception. Si une telle clause existe, il doit s'agir de la dernière des clauses qui spécifient ce type d'exception.

Si une clause `catch` a un filtre d'exception, elle peut spécifier le type d'exception qui est le même ou moins dérivé qu'un type d'exception d'une clause `catch` qui apparaît après elle. Par exemple, si un filtre d'exception est présent, une clause `catch (Exception e)` n'a pas besoin d'être la dernière clause.

Exceptions dans les méthodes asynchrones et itérateurs

Si une exception se produit dans une fonction asynchrone, elle se propage à l'appelant de la fonction lorsque vous attendez le résultat de la fonction, comme le montre l'exemple suivant :

C#

```
public static async Task Run()
{
    try
    {
        Task<int> processing = ProcessAsync(-1);
        Console.WriteLine("Launched processing.");
```

```
        int result = await processing;
        Console.WriteLine($"Result: {result}.");
    }
    catch (ArgumentException e)
    {
        Console.WriteLine($"Processing failed: {e.Message}");
    }
    // Output:
    // Launched processing.
    // Processing failed: Input must be non-negative. (Parameter 'input')
}

private static async Task<int> ProcessAsync(int input)
{
    if (input < 0)
    {
        throw new ArgumentOutOfRangeException(nameof(input), "Input must be
non-negative.");
    }

    await Task.Delay(500);
    return input;
}
```

Si une exception se produit dans une méthode d'itérateur, elle se propage à l'appelant uniquement lorsque l'itérateur passe à l'élément suivant.

Instruction `try-finally`

Dans une instruction `try-finally`, le bloc `finally` est exécuté lorsque le contrôle quitte le bloc `try`. Le contrôle peut quitter le bloc `try` à la suite de

- une exécution normale,
- l'exécution d'une instruction de saut (c'est-à-dire, `return`, `break`, `continue` ou `goto`), ou
- la propagation d'une exception hors du bloc `try`.

L'exemple suivant utilise le bloc `finally` pour réinitialiser l'état d'un objet avant que le contrôle quitte la méthode :

C#

```
public async Task HandleRequest(int itemId, CancellationToken ct)
{
    Busy = true;

    try
    {
        await ProcessAsync(itemId, ct);
```

```
    }
    finally
    {
        Busy = false;
    }
}
```

Vous pouvez également utiliser le bloc `finally` pour nettoyer les ressources allouées utilisées dans le bloc `try`.

> ⓘ Notes
>
> Lorsque le type d'une ressource implémente l'interface **IDisposable** ou **IAsyncDisposable**, considérez l'**instruction using**. L'instruction `using` garantit que les ressources acquises sont supprimées lorsque le contrôle quitte l'instruction `using`. Le compilateur transforme une instruction `using` en instruction `try-finally`.

Dans presque tous les cas, les blocs `finally` sont exécutés. Les seuls cas où les blocs `finally` ne sont pas exécutés impliquent l'arrêt immédiat d'un programme. Par exemple, un tel arrêt peut se produire en raison de l'appel Environment.FailFast ou d'une exception OverflowException ou InvalidProgramException. La plupart des systèmes d'exploitation effectuent un nettoyage raisonnable des ressources dans le cadre de l'arrêt et du déchargement du processus.

Instruction `try-catch-finally`

Vous utilisez une instruction `try-catch-finally` pour gérer les exceptions qui peuvent se produire pendant l'exécution du bloc `try` et pour spécifier le code qui doit être exécuté lorsque le contrôle quitte l'instruction `try` :

C#

```
public async Task ProcessRequest(int itemId, CancellationToken ct)
{
    Busy = true;

    try
    {
        await ProcessAsync(itemId, ct);
    }
    catch (Exception e) when (e is not OperationCanceledException)
    {
        LogError(e, $"Failed to process request for item ID {itemId}.");
```

```
            throw;
        }
        finally
        {
            Busy = false;
        }

    }
```

Lorsqu'une exception est gérée par un bloc `catch`, le bloc `finally` est exécuté après l'exécution de ce bloc `catch` (même si une autre exception se produit pendant l'exécution du bloc `catch`). Pour plus d'informations sur les blocs `catch` et `finally`, consultez respectivement les sections L'instruction try-catch et L'instructiontry-finally.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Instruction throw
- Instruction try
- Exceptions

Voir aussi

- Informations de référence sur C#
- Exceptions et gestion des exceptions
- Gestion et levée d'exceptions dans .NET
- préférences de levée (règle de style IDE0016)
- AppDomain.FirstChanceException
- AppDomain.UnhandledException
- TaskScheduler.UnobservedTaskException

Instructions de gestion des exceptions : `throw`, `try-catch`, `try-finally` et `try-catch-finally`

Article • 05/06/2023

Vous utilisez les instructions `throw` et `try` pour travailler avec des exceptions. Utilisez l'instruction throw pour lever une exception. Utilisez l'instruction try pour intercepter et gérer les exceptions qui peuvent se produire pendant l'exécution d'un bloc de code.

Instruction `throw`

L'instruction `throw` lève une exception :

```
C#

if (shapeAmount <= 0)
{
    throw new ArgumentOutOfRangeException(nameof(shapeAmount), "Amount of
shapes must be positive.");
}
```

Dans une instruction `throw e;`, le résultat de l'expression `e` doit être implicitement convertible en System.Exception.

Vous pouvez utiliser les classes d'exception intégrées, par exemple ArgumentOutOfRangeException ou InvalidOperationException. .NET fournit également les méthodes d'assistance pour lever des exceptions dans certaines conditions : ArgumentNullException.ThrowIfNull et ArgumentException.ThrowIfNullOrEmpty. Vous pouvez également définir vos propres classes d'exception qui dérivent de System.Exception. Pour plus d'informations, consultez Création et levée d'exceptions.

À l'intérieur d'un bloc catch, vous pouvez utiliser une instruction `throw;` pour lever à nouveau l'exception gérée par le bloc `catch` :

```
C#

try
{
    ProcessShapes(shapeAmount);
}
catch (Exception e)
```

```
{
    LogError(e, "Shape processing failed.");
    throw;
}
```

> ⓘ Notes
>
> `throw;` conserve la trace de pile d'origine de l'exception, qui est stockée dans la propriété **Exception.StackTrace**. À l'inverse, `throw e;` met à jour la propriété **StackTrace** de `e`.

Quand une exception est levée, le Common Language Runtime (CLR) recherche le bloc catch qui gère cette exception. Si la méthode exécutée ne contient pas un tel bloc `catch`, le CLR examine la méthode qui a appelé la méthode actuelle, puis remonte la pile des appels. Si aucun bloc `catch` n'est trouvé, le CLR met fin au thread en cours d'exécution. Pour plus d'informations, consultez la section Comment les exceptions sont gérées de la spécification du langage C#.

Expression `throw`

Vous pouvez également utiliser `throw` comme expression. Cela peut être pratique dans un certain nombre de cas, notamment :

- l'opérateur conditionnel : L'exemple suivant utilise une expression `throw` pour lever un ArgumentException lorsque le tableau `args` passé est vide :

  C#

  ```
  string first = args.Length >= 1
      ? args[0]
      : throw new ArgumentException("Please supply at least one
  argument.");
  ```

- l'opérateur de fusion de Null : L'exemple suivant utilise une expression `throw` pour lever un ArgumentNullException lorsque la chaîne à affecter à une propriété est `null` :

  C#

  ```
  public string Name
  {
      get => name;
      set => name = value ??
  ```

```
        throw new ArgumentNullException(paramName: nameof(value),
message: "Name cannot be null");
}
```

- un lambda ou une méthode expression-bodied : L'exemple suivant utilise une expression `throw` pour lever un InvalidCastException pour indiquer qu'une conversion en valeur DateTime n'est pas prise en charge :

```
C#

DateTime ToDateTime(IFormatProvider provider) =>
         throw new InvalidCastException("Conversion to a DateTime is
not supported.");
```

Instruction `try`

Vous pouvez utiliser l'instruction `try` sous l'une des formes suivantes : try-catch - pour gérer les exceptions qui peuvent se produire pendant l'exécution du code à l'intérieur d'un bloc `try`, try-finally - pour spécifier le code qui est exécuté lorsque le contrôle quitte le bloc `try` et try-catch-finally - en combinaison des deux formulaires précédents.

Instruction `try-catch`

Utilisez l'instruction `try-catch` pour intercepter et gérer les exceptions qui peuvent se produire pendant l'exécution d'un bloc de code. Placez le code là où une exception peut se produire à l'intérieur d'un bloc `try`. Utilisez une *clause catch* pour spécifier le type de base des exceptions que vous souhaitez gérer dans le bloc `catch` correspondant :

```
C#

try
{
    var result = Process(-3, 4);
    Console.WriteLine($"Processing succeeded: {result}");
}
catch (ArgumentException e)
{
    Console.WriteLine($"Processing failed: {e.Message}");
}
```

Vous pouvez fournir plusieurs clauses catch :

```
C#
```

```
try
{
    var result = await ProcessAsync(-3, 4, cancellationToken);
    Console.WriteLine($"Processing succeeded: {result}");
}
catch (ArgumentException e)
{
    Console.WriteLine($"Processing failed: {e.Message}");
}
catch (OperationCanceledException)
{
    Console.WriteLine("Processing is cancelled.");
}
```

Lorsqu'une exception se produit, les clauses catch sont examinées dans l'ordre spécifié, de haut en bas. Au maximum, un seul bloc `catch` est exécuté pour toute exception levée. Comme le montre également l'exemple précédent, vous pouvez omettre la déclaration d'une variable d'exception et spécifier uniquement le type d'exception dans une clause catch. Une clause catch sans type d'exception spécifié correspond à n'importe quelle exception et, le cas échéant, doit être la dernière clause catch.

Si vous souhaitez lever à nouveau une exception interceptée, utilisez l'instruction throw, comme dans l'exemple suivant :

C#

```
try
{
    var result = Process(-3, 4);
    Console.WriteLine($"Processing succeeded: {result}");
}
catch (Exception e)
{
    LogError(e, "Processing failed.");
    throw;
}
```

> ⓘ **Notes**
>
> `throw;` conserve la trace de pile d'origine de l'exception, qui est stockée dans la propriété **Exception.StackTrace**. À l'inverse, `throw e;` met à jour la propriété **StackTrace** de `e`.

Un filtre d'exception `when`

En plus d'un type d'exception, vous pouvez également spécifier un filtre d'exception qui examine plus en détail une exception et détermine si le bloc `catch` correspondant gère cette exception. Un filtre d'exception est une expression booléenne qui suit le mot clé `when`, comme dans l'exemple suivant :

C#

```
try
{
    var result = Process(-3, 4);
    Console.WriteLine($"Processing succeeded: {result}");
}
catch (Exception e) when (e is ArgumentException || e is
DivideByZeroException)
{
    Console.WriteLine($"Processing failed: {e.Message}");
}
```

L'exemple précédent utilise un filtre d'exception pour fournir un seul bloc `catch` pour gérer les exceptions de deux types spécifiés.

Vous pouvez fournir plusieurs clauses `catch` pour le même type d'exception si elles se distinguent par des filtres d'exception. L'une de ces clauses peut ne pas avoir de filtre d'exception. Si une telle clause existe, il doit s'agir de la dernière des clauses qui spécifient ce type d'exception.

Si une clause `catch` a un filtre d'exception, elle peut spécifier le type d'exception qui est le même ou moins dérivé qu'un type d'exception d'une clause `catch` qui apparaît après elle. Par exemple, si un filtre d'exception est présent, une clause `catch (Exception e)` n'a pas besoin d'être la dernière clause.

Exceptions dans les méthodes asynchrones et itérateurs

Si une exception se produit dans une fonction asynchrone, elle se propage à l'appelant de la fonction lorsque vous attendez le résultat de la fonction, comme le montre l'exemple suivant :

C#

```
public static async Task Run()
{
    try
    {
        Task<int> processing = ProcessAsync(-1);
        Console.WriteLine("Launched processing.");
```

```
        int result = await processing;
        Console.WriteLine($"Result: {result}.");
    }
    catch (ArgumentException e)
    {
        Console.WriteLine($"Processing failed: {e.Message}");
    }
    // Output:
    // Launched processing.
    // Processing failed: Input must be non-negative. (Parameter 'input')
}

private static async Task<int> ProcessAsync(int input)
{
    if (input < 0)
    {
        throw new ArgumentOutOfRangeException(nameof(input), "Input must be
non-negative.");
    }

    await Task.Delay(500);
    return input;
}
```

Si une exception se produit dans une méthode d'itérateur, elle se propage à l'appelant uniquement lorsque l'itérateur passe à l'élément suivant.

Instruction `try-finally`

Dans une instruction `try-finally`, le bloc `finally` est exécuté lorsque le contrôle quitte le bloc `try`. Le contrôle peut quitter le bloc `try` à la suite de

- une exécution normale,
- l'exécution d'une instruction de saut (c'est-à-dire, `return`, `break`, `continue` ou `goto`), ou
- la propagation d'une exception hors du bloc `try`.

L'exemple suivant utilise le bloc `finally` pour réinitialiser l'état d'un objet avant que le contrôle quitte la méthode :

C#

```
public async Task HandleRequest(int itemId, CancellationToken ct)
{
    Busy = true;

    try
    {
        await ProcessAsync(itemId, ct);
```

```
    }
    finally
    {
        Busy = false;
    }
}
```

Vous pouvez également utiliser le bloc `finally` pour nettoyer les ressources allouées utilisées dans le bloc `try`.

> ⓘ Notes
>
> Lorsque le type d'une ressource implémente l'interface **IDisposable** ou **IAsyncDisposable**, considérez l'**instruction using**. L'instruction `using` garantit que les ressources acquises sont supprimées lorsque le contrôle quitte l'instruction `using`. Le compilateur transforme une instruction `using` en instruction `try-finally`.

Dans presque tous les cas, les blocs `finally` sont exécutés. Les seuls cas où les blocs `finally` ne sont pas exécutés impliquent l'arrêt immédiat d'un programme. Par exemple, un tel arrêt peut se produire en raison de l'appel Environment.FailFast ou d'une exception OverflowException ou InvalidProgramException. La plupart des systèmes d'exploitation effectuent un nettoyage raisonnable des ressources dans le cadre de l'arrêt et du déchargement du processus.

Instruction `try-catch-finally`

Vous utilisez une instruction `try-catch-finally` pour gérer les exceptions qui peuvent se produire pendant l'exécution du bloc `try` et pour spécifier le code qui doit être exécuté lorsque le contrôle quitte l'instruction `try` :

C#

```
public async Task ProcessRequest(int itemId, CancellationToken ct)
{
    Busy = true;

    try
    {
        await ProcessAsync(itemId, ct);
    }
    catch (Exception e) when (e is not OperationCanceledException)
    {
        LogError(e, $"Failed to process request for item ID {itemId}.");
```

```
            throw;
        }
        finally
        {
            Busy = false;
        }

    }
```

Lorsqu'une exception est gérée par un bloc `catch`, le bloc `finally` est exécuté après l'exécution de ce bloc `catch` (même si une autre exception se produit pendant l'exécution du bloc `catch`). Pour plus d'informations sur les blocs `catch` et `finally`, consultez respectivement les sections L'instruction try-catch et L'instructiontry-finally.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Instruction throw
- Instruction try
- Exceptions

Voir aussi

- Informations de référence sur C#
- Exceptions et gestion des exceptions
- Gestion et levée d'exceptions dans .NET
- préférences de levée (règle de style IDE0016)
- AppDomain.FirstChanceException
- AppDomain.UnhandledException
- TaskScheduler.UnobservedTaskException

Instructions checked et unchecked (informations de référence sur C#)

Article • 03/06/2023

Les instructions `checked` et `unchecked` spécifient le contexte de vérification de dépassement pour les opérations arithmétiques et les conversions de type intégral. En cas de dépassement arithmétique d'un entier, le contexte de vérification de dépassement définit ce qui doit se produire. Dans un contexte vérifié, une exception System.OverflowException est levée. Si le dépassement se produit dans une expression constante, une erreur de compilation se produit. Dans un contexte non vérifié, le résultat de l'opération est tronqué en supprimant tous les bits de poids fort qui ne tiennent pas dans le type destinataire. Par exemple, dans le cas d'une addition, un cycle complet est effectué pour revenir à la valeur minimale après avoir dépassé la valeur maximale. L'exemple suivant montre la même opération dans un contexte checked et unchecked :

C#

```
uint a = uint.MaxValue;

unchecked
{
    Console.WriteLine(a + 3);  // output: 2
}

try
{
    checked
    {
        Console.WriteLine(a + 3);
    }
}
catch (OverflowException e)
{
    Console.WriteLine(e.Message);  // output: Arithmetic operation resulted
in an overflow.
}
```

Notes

Le comportement des conversions et des opérateurs *définis par l'utilisateur* en cas de dépassement peut différer de celui décrit dans le paragraphe précédent. En particulier, les **opérateurs vérifiés définis par l'utilisateur** peuvent ne pas lever d'exception dans un contexte vérifié.

Pour plus d'informations, consultez les sections Débordement arithmétique et division par zéro et Opérateurs vérifiés définis par l'utilisateur de l'article Opérateurs arithmétiques.

Pour spécifier le contexte de contrôle de dépassement pour une expression, vous pouvez également utiliser les opérateurs `checked` et `unchecked`, comme le montre l'exemple suivant :

C#

```
double a = double.MaxValue;

int b = unchecked((int)a);
Console.WriteLine(b);  // output: -2147483648

try
{
    b = checked((int)a);
}
catch (OverflowException e)
{
    Console.WriteLine(e.Message);  // output: Arithmetic operation resulted
in an overflow.
}
```

Les instructions `checked` et `unchecked` et les opérateurs affectent uniquement le contexte de contrôle de dépassement pour les opérations qui se trouvent *textuellement* dans le bloc d'instructions ou les parenthèses de l'opérateur, comme le montre l'exemple suivant :

C#

```
int Multiply(int a, int b) => a * b;

int factor = 2;

try
{
    checked
    {
        Console.WriteLine(Multiply(factor, int.MaxValue));  // output: -2
    }
}
catch (OverflowException e)
{
    Console.WriteLine(e.Message);
}

try
{
```

```
    checked
    {
        Console.WriteLine(Multiply(factor, factor * int.MaxValue));
    }
}
catch (OverflowException e)
{
    Console.WriteLine(e.Message);  // output: Arithmetic operation resulted
in an overflow.
}
```

Dans l'exemple précédent, le premier appel de la fonction locale `Multiply` montre que l'instruction `checked` n'affecte pas le contexte de contrôle de dépassement dans la fonction `Multiply`, car aucune exception n'est levée. Lors de la deuxième invocation de la fonction `Multiply`, l'expression qui calcule le deuxième argument de la fonction est évaluée dans un contexte vérifié et génère une exception car elle se trouve textuellement à l'intérieur du bloc de l'instruction `checked`.

Opérations affectées par le contexte de contrôle de dépassement

Le contexte de contrôle de dépassement affecte les opérations suivantes :

- Les opérateurs arithmétiques intégrés suivants : opérateurs unaires `++`, `--`, `-` et binaires `+`, `-`, `*` et `/`, quand leurs opérandes sont de type intégral (c'est-à-dire de type intégral numérique ou char) ou de type enum.
- Conversions numériques explicites entre types intégraux, ou de float ou double en un type intégral.

  > ⓘ **Notes**
  >
  > Lorsque vous convertissez une valeur `decimal` en type intégral et que le résultat est en dehors de la plage du type de destination, une exception **OverflowException** est toujours levée, quel que soit le contexte de contrôle de dépassement.

- À compter de C# 11, les conversions et les opérateurs vérifiés définis par l'utilisateur. Pour plus d'informations, consultez la section Opérateurs vérifiés définis par l'utilisateur de l'article Opérateurs arithmétiques.

Contexte de contrôle de dépassement par défaut

Si vous ne spécifiez pas le contexte de contrôle de dépassement, la valeur de l'option de compilateur **CheckForOverflowUnderflow** définit le contexte par défaut pour les expressions non constantes. Par défaut, la valeur de cette option n'est pas définie et les conversions et opérations arithmétiques de type intégral sont exécutées dans un contexte **non vérifié** (unchecked).

Les expressions constantes sont évaluées par défaut dans un contexte checked, et une erreur de compilation se produit en cas de dépassement. Vous pouvez spécifier explicitement un contexte non vérifié pour une expression constante avec l'instruction ou l'opérateur `unchecked`.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Les instructions checked et unchecked
- Opérateurs vérifiés et non vérifiés
- Opérateurs checked et unchecked définis par l'utilisateur - C# 11

Voir aussi

- Informations de référence sur C#
- Option du compilateur **CheckForOverflowUnderflow**

Instructions checked et unchecked (informations de référence sur C#)

Article • 03/06/2023

Les instructions `checked` et `unchecked` spécifient le contexte de vérification de dépassement pour les opérations arithmétiques et les conversions de type intégral. En cas de dépassement arithmétique d'un entier, le contexte de vérification de dépassement définit ce qui doit se produire. Dans un contexte vérifié, une exception System.OverflowException est levée. Si le dépassement se produit dans une expression constante, une erreur de compilation se produit. Dans un contexte non vérifié, le résultat de l'opération est tronqué en supprimant tous les bits de poids fort qui ne tiennent pas dans le type destinataire. Par exemple, dans le cas d'une addition, un cycle complet est effectué pour revenir à la valeur minimale après avoir dépassé la valeur maximale. L'exemple suivant montre la même opération dans un contexte checked et unchecked :

C#

```
uint a = uint.MaxValue;

unchecked
{
    Console.WriteLine(a + 3);  // output: 2
}

try
{
    checked
    {
        Console.WriteLine(a + 3);
    }
}
catch (OverflowException e)
{
    Console.WriteLine(e.Message);  // output: Arithmetic operation resulted
in an overflow.
}
```

Notes

Le comportement des conversions et des opérateurs *définis par l'utilisateur* en cas de dépassement peut différer de celui décrit dans le paragraphe précédent. En particulier, les **opérateurs vérifiés définis par l'utilisateur** peuvent ne pas lever d'exception dans un contexte vérifié.

Pour plus d'informations, consultez les sections Débordement arithmétique et division par zéro et Opérateurs vérifiés définis par l'utilisateur de l'article Opérateurs arithmétiques.

Pour spécifier le contexte de contrôle de dépassement pour une expression, vous pouvez également utiliser les opérateurs `checked` et `unchecked`, comme le montre l'exemple suivant :

C#

```
double a = double.MaxValue;

int b = unchecked((int)a);
Console.WriteLine(b);  // output: -2147483648

try
{
    b = checked((int)a);
}
catch (OverflowException e)
{
    Console.WriteLine(e.Message);  // output: Arithmetic operation resulted
in an overflow.
}
```

Les instructions `checked` et `unchecked` et les opérateurs affectent uniquement le contexte de contrôle de dépassement pour les opérations qui se trouvent *textuellement* dans le bloc d'instructions ou les parenthèses de l'opérateur, comme le montre l'exemple suivant :

C#

```
int Multiply(int a, int b) => a * b;

int factor = 2;

try
{
    checked
    {
        Console.WriteLine(Multiply(factor, int.MaxValue));  // output: -2
    }
}
catch (OverflowException e)
{
    Console.WriteLine(e.Message);
}

try
{
```

```
    checked
    {
        Console.WriteLine(Multiply(factor, factor * int.MaxValue));
    }
}
catch (OverflowException e)
{
    Console.WriteLine(e.Message);  // output: Arithmetic operation resulted
in an overflow.
}
```

Dans l'exemple précédent, le premier appel de la fonction locale `Multiply` montre que l'instruction `checked` n'affecte pas le contexte de contrôle de dépassement dans la fonction `Multiply`, car aucune exception n'est levée. Lors de la deuxième invocation de la fonction `Multiply`, l'expression qui calcule le deuxième argument de la fonction est évaluée dans un contexte vérifié et génère une exception car elle se trouve textuellement à l'intérieur du bloc de l'instruction `checked`.

Opérations affectées par le contexte de contrôle de dépassement

Le contexte de contrôle de dépassement affecte les opérations suivantes :

- Les opérateurs arithmétiques intégrés suivants : opérateurs unaires `++`, `--`, `-` et binaires `+`, `-`, `*` et `/`, quand leurs opérandes sont de type intégral (c'est-à-dire de type intégral numérique ou char) ou de type enum.
- Conversions numériques explicites entre types intégraux, ou de float ou double en un type intégral.

  > ⓘ **Notes**
  >
  > Lorsque vous convertissez une valeur `decimal` en type intégral et que le résultat est en dehors de la plage du type de destination, une exception **OverflowException** est toujours levée, quel que soit le contexte de contrôle de dépassement.

- À compter de C# 11, les conversions et les opérateurs vérifiés définis par l'utilisateur. Pour plus d'informations, consultez la section Opérateurs vérifiés définis par l'utilisateur de l'article Opérateurs arithmétiques.

Contexte de contrôle de dépassement par défaut

Si vous ne spécifiez pas le contexte de contrôle de dépassement, la valeur de l'option de compilateur **CheckForOverflowUnderflow** définit le contexte par défaut pour les expressions non constantes. Par défaut, la valeur de cette option n'est pas définie et les conversions et opérations arithmétiques de type intégral sont exécutées dans un contexte **non vérifié** (unchecked).

Les expressions constantes sont évaluées par défaut dans un contexte checked, et une erreur de compilation se produit en cas de dépassement. Vous pouvez spécifier explicitement un contexte non vérifié pour une expression constante avec l'instruction ou l'opérateur `unchecked`.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Les instructions checked et unchecked
- Opérateurs vérifiés et non vérifiés
- Opérateurs checked et unchecked définis par l'utilisateur - C# 11

Voir aussi

- Informations de référence sur C#
- Option du compilateur **CheckForOverflowUnderflow**

Instructions checked et unchecked (informations de référence sur C#)

Article • 03/06/2023

Les instructions `checked` et `unchecked` spécifient le contexte de vérification de dépassement pour les opérations arithmétiques et les conversions de type intégral. En cas de dépassement arithmétique d'un entier, le contexte de vérification de dépassement définit ce qui doit se produire. Dans un contexte vérifié, une exception System.OverflowException est levée. Si le dépassement se produit dans une expression constante, une erreur de compilation se produit. Dans un contexte non vérifié, le résultat de l'opération est tronqué en supprimant tous les bits de poids fort qui ne tiennent pas dans le type destinataire. Par exemple, dans le cas d'une addition, un cycle complet est effectué pour revenir à la valeur minimale après avoir dépassé la valeur maximale. L'exemple suivant montre la même opération dans un contexte checked et unchecked :

C#

```
uint a = uint.MaxValue;

unchecked
{
    Console.WriteLine(a + 3);  // output: 2
}

try
{
    checked
    {
        Console.WriteLine(a + 3);
    }
}
catch (OverflowException e)
{
    Console.WriteLine(e.Message);  // output: Arithmetic operation resulted
in an overflow.
}
```

Notes

Le comportement des conversions et des opérateurs *définis par l'utilisateur* en cas de dépassement peut différer de celui décrit dans le paragraphe précédent. En particulier, les **opérateurs vérifiés définis par l'utilisateur** peuvent ne pas lever d'exception dans un contexte vérifié.

Pour plus d'informations, consultez les sections Débordement arithmétique et division par zéro et Opérateurs vérifiés définis par l'utilisateur de l'article Opérateurs arithmétiques.

Pour spécifier le contexte de contrôle de dépassement pour une expression, vous pouvez également utiliser les opérateurs `checked` et `unchecked`, comme le montre l'exemple suivant :

C#

```
double a = double.MaxValue;

int b = unchecked((int)a);
Console.WriteLine(b);  // output: -2147483648

try
{
    b = checked((int)a);
}
catch (OverflowException e)
{
    Console.WriteLine(e.Message);  // output: Arithmetic operation resulted
in an overflow.
}
```

Les instructions `checked` et `unchecked` et les opérateurs affectent uniquement le contexte de contrôle de dépassement pour les opérations qui se trouvent *textuellement* dans le bloc d'instructions ou les parenthèses de l'opérateur, comme le montre l'exemple suivant :

C#

```
int Multiply(int a, int b) => a * b;

int factor = 2;

try
{
    checked
    {
        Console.WriteLine(Multiply(factor, int.MaxValue));  // output: -2
    }
}
catch (OverflowException e)
{
    Console.WriteLine(e.Message);
}

try
{
```

```
    checked
    {
        Console.WriteLine(Multiply(factor, factor * int.MaxValue));
    }
}
catch (OverflowException e)
{
    Console.WriteLine(e.Message);  // output: Arithmetic operation resulted
in an overflow.
}
```

Dans l'exemple précédent, le premier appel de la fonction locale `Multiply` montre que l'instruction `checked` n'affecte pas le contexte de contrôle de dépassement dans la fonction `Multiply`, car aucune exception n'est levée. Lors de la deuxième invocation de la fonction `Multiply`, l'expression qui calcule le deuxième argument de la fonction est évaluée dans un contexte vérifié et génère une exception car elle se trouve textuellement à l'intérieur du bloc de l'instruction `checked`.

Opérations affectées par le contexte de contrôle de dépassement

Le contexte de contrôle de dépassement affecte les opérations suivantes :

- Les opérateurs arithmétiques intégrés suivants : opérateurs unaires `++`, `--`, `-` et binaires `+`, `-`, `*` et `/`, quand leurs opérandes sont de type intégral (c'est-à-dire de type intégral numérique ou char) ou de type enum.
- Conversions numériques explicites entre types intégraux, ou de float ou double en un type intégral.

  > ⓘ **Notes**
  >
  > Lorsque vous convertissez une valeur `decimal` en type intégral et que le résultat est en dehors de la plage du type de destination, une exception **OverflowException** est toujours levée, quel que soit le contexte de contrôle de dépassement.

- À compter de C# 11, les conversions et les opérateurs vérifiés définis par l'utilisateur. Pour plus d'informations, consultez la section Opérateurs vérifiés définis par l'utilisateur de l'article Opérateurs arithmétiques.

Contexte de contrôle de dépassement par défaut

Si vous ne spécifiez pas le contexte de contrôle de dépassement, la valeur de l'option de compilateur **CheckForOverflowUnderflow** définit le contexte par défaut pour les expressions non constantes. Par défaut, la valeur de cette option n'est pas définie et les conversions et opérations arithmétiques de type intégral sont exécutées dans un contexte **non vérifié** (unchecked).

Les expressions constantes sont évaluées par défaut dans un contexte checked, et une erreur de compilation se produit en cas de dépassement. Vous pouvez spécifier explicitement un contexte non vérifié pour une expression constante avec l'instruction ou l'opérateur `unchecked`.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Les instructions checked et unchecked
- Opérateurs vérifiés et non vérifiés
- Opérateurs checked et unchecked définis par l'utilisateur - C# 11

Voir aussi

- Informations de référence sur C#
- Option du compilateur **CheckForOverflowUnderflow**

fixed, instruction - épingler une variable pour les opérations de pointeur

Article • 03/06/2023

L'instruction `fixed` empêche le récupérateur de mémoire de déplacer une variable déplaçable, et déclare un pointeur vers cette variable. L'adresse d'une variable fixe ou épinglée ne change pas durant l'exécution de l'instruction. Vous pouvez utiliser le pointeur déclaré uniquement à l'intérieur de l'instruction `fixed` correspondante. Le pointeur déclaré est en lecture seule et ne peut pas être modifié :

C#

```
unsafe
{
    byte[] bytes = { 1, 2, 3 };
    fixed (byte* pointerToFirst = bytes)
    {
        Console.WriteLine($"The address of the first array element:
{(long)pointerToFirst:X}.");
        Console.WriteLine($"The value of the first array element:
{*pointerToFirst}.");
    }
}
// Output is similar to:
// The address of the first array element: 2173F80B5C8.
// The value of the first array element: 1.
```

Notes

Vous pouvez utiliser l'instruction `fixed` uniquement dans un contexte **non sécurisé**. Le code qui contient des blocs unsafe doit être compilé avec l'option de compilateur **AllowUnsafeBlocks**.

Vous pouvez initialiser le pointeur déclaré comme suit :

- Avec un tableau, comme le montre l'exemple au début de cet article. Le pointeur initialisé contient l'adresse du premier élément de tableau.
- Avec l'adresse d'une variable. Utilisez l'opérateur address-of &, comme le montre l'exemple suivant :

  C#

```
unsafe
{
    int[] numbers = { 10, 20, 30 };
    fixed (int* toFirst = &numbers[0], toLast = &numbers[^1])
    {
        Console.WriteLine(toLast - toFirst);  // output: 2
    }
}
```

Les champs d'objet sont un autre exemple de variables déplaçables qui peuvent être épinglées.

Lorsque le pointeur initialisé contient l'adresse d'un champ d'objet ou d'un élément de tableau, l'instruction `fixed` garantit que le récupérateur de mémoire ne déplace pas ou ne supprime pas l'instance d'objet conteneur pendant l'exécution du corps de l'instruction.

- Avec l'instance du type qui implémente une méthode nommée `GetPinnableReference`. Cette méthode doit retourner une variable `ref` d'un type non managé. Les types System.Span<T> et System.ReadOnlySpan<T> de .NET utilisent ce modèle. Vous pouvez épingler des instances span, comme le montre l'exemple suivant :

C#

```
unsafe
{
    int[] numbers = { 10, 20, 30, 40, 50 };
    Span<int> interior = numbers.AsSpan()[1..^1];
    fixed (int* p = interior)
    {
        for (int i = 0; i < interior.Length; i++)
        {
            Console.Write(p[i]);
        }
        // output: 203040
    }
}
```

Pour plus d'informations, consultez la référence d'API Span<T>.GetPinnableReference().

- Avec une chaîne, comme le montre l'exemple suivant :

C#

```
unsafe
{
    var message = "Hello!";
    fixed (char* p = message)
    {
        Console.WriteLine(*p);  // output: H
    }
}
```

- Avec une mémoire tampon de taille fixe.

Vous pouvez allouer de la mémoire à la pile, où elle n'est pas soumise au nettoyage de la mémoire et n'a donc pas besoin d'être épinglée. Pour ce faire, utilisez une expression stackalloc.

Vous pouvez également utiliser le mot clé `fixed` pour déclarer une mémoire tampon de taille fixe.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- fixed, instruction
- Variables fixes et déplaçables

Voir aussi

- Informations de référence sur C#
- Code, types de pointeurs et pointeurs de fonction non sécurisés
- Opérateurs liés aux pointeurs
- unsafe

Paramètres de méthode (référence C#)

Article • 07/04/2023

En C#, les arguments peuvent être passés aux paramètres par valeur ou par référence. N'oubliez pas que les types C# peuvent être des types référence (`class`) ou des types valeur (`struct`) :

- *Passer par valeur* signifie **passer une copie de la variable** à la méthode.
- *Passer par référence* signifie **passer l'accès à la variable** à la méthode.
- Une variable d'un *type référence* contient une référence à ses données.
- Une variable d'un *type valeur* contient ses données directement.

Étant donné qu'un struct est un type valeur, quand vous passez un struct par valeur à une méthode, la méthode reçoit et traite une copie de l'argument struct. La méthode n'a pas accès au struct d'origine dans la méthode d'appel, et ne peut donc pas le modifier de quelque manière que ce soit. La méthode peut modifier uniquement la copie.

Une instance de classe est un type référence, pas un type valeur. Quand un type référence est passé par valeur à une méthode, celle-ci reçoit une copie de la référence à l'instance de classe. Autrement dit, la méthode appelée reçoit une copie de l'adresse de l'instance, et la méthode appelante conserve l'adresse d'origine de l'instance. L'instance de classe dans la méthode d'appel a une adresse, le paramètre dans la méthode appelée a une copie de l'adresse, et les deux adresses font référence au même objet. Étant donné que le paramètre contient uniquement une copie de l'adresse, la méthode appelée ne peut pas modifier l'adresse de l'instance de la classe dans la méthode. Toutefois, la méthode appelée peut utiliser la copie de l'adresse pour accéder aux membres de la classe qui ont à la fois l'adresse d'origine et la copie de la référence d'adresse. Si la méthode appelée modifie un membre de classe, l'instance de classe d'origine dans la méthode d'appel change également.

La sortie de l'exemple suivant illustre la différence. La valeur du champ `willIChange` de l'instance de classe est changée par l'appel à la méthode `ClassTaker`, car la méthode utilise l'adresse mentionnée dans le paramètre pour rechercher le champ spécifié de l'instance de classe. Le champ `willIChange` du struct de la méthode d'appel n'est pas changé par l'appel à la méthode `StructTaker`, car la valeur de l'argument est une copie du struct lui-même, et non pas une copie de son adresse. `StructTaker` change la copie, et cette dernière est perdue quand l'appel à `StructTaker` est terminé.

```
C#
```

```
class TheClass
{
    public string? willIChange;
}

struct TheStruct
{
    public string willIChange;
}

class TestClassAndStruct
{
    static void ClassTaker(TheClass c)
    {
        c.willIChange = "Changed";
    }

    static void StructTaker(TheStruct s)
    {
        s.willIChange = "Changed";
    }

    public static void Main()
    {
        TheClass testClass = new TheClass();
        TheStruct testStruct = new TheStruct();

        testClass.willIChange = "Not Changed";
        testStruct.willIChange = "Not Changed";

        ClassTaker(testClass);
        StructTaker(testStruct);

        Console.WriteLine("Class field = {0}", testClass.willIChange);
        Console.WriteLine("Struct field = {0}", testStruct.willIChange);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Class field = Changed
    Struct field = Not Changed
*/
```

Les modifications apportées à l'argument qui sont visibles à partir de l'appelant dépendent de la façon dont un argument est passé et de s'il s'agit d'un type référence ou d'un type valeur.

Passer un type valeur par valeur

Lorsque vous passez un type *valeurpar valeur* :

- Si la méthode affecte le paramètre pour faire référence à un autre objet, ces modifications **ne sont pas** visibles à partir de l'appelant.
- Si la méthode modifie l'état de l'objet auquel le paramètre fait référence, ces modifications **ne sont pas** visibles par l'appelant.

L'exemple suivant illustre le passage de paramètres de type valeur par valeur. La variable `n` est passée par valeur à la méthode `SquareIt`. Toutes les modifications qui ont lieu à l'intérieur de la méthode n'ont aucun effet sur la valeur d'origine de la variable.

C#

```
int n = 5;
System.Console.WriteLine("The value before calling the method: {0}", n);

SquareIt(n);  // Passing the variable by value.
System.Console.WriteLine("The value after calling the method: {0}", n);

// Keep the console window open in debug mode.
System.Console.WriteLine("Press any key to exit.");
System.Console.ReadKey();

static void SquareIt(int x)
// The parameter x is passed by value.
// Changes to x will not affect the original value of x.
{
    x *= x;
    System.Console.WriteLine("The value inside the method: {0}", x);
}
/* Output:
    The value before calling the method: 5
    The value inside the method: 25
    The value after calling the method: 5
*/
```

La variable `n` est un type valeur. Elle contient ses données, la valeur `5`. Quand `SquareIt` est appelée, le contenu de `n` est copié dans le paramètre `x`, qui est mis au carré à l'intérieur de la méthode. Dans `Main`, toutefois, la valeur de `n` est identique avant et après l'appel de la méthode `SquareIt`. La modification qui intervient à l'intérieur de la méthode affecte uniquement la variable locale `x`.

Passer un type valeur par référence

Lorsque vous transmettez un type *valeurpar référence* :

- Si la méthode affecte le paramètre pour faire référence à un autre objet, ces modifications **ne sont pas** visibles à partir de l'appelant.
- Si la méthode modifie l'état de l'objet auquel le paramètre fait référence, ces modifications **sont visibles** à partir de l'appelant.

L'exemple suivant est identique au précédent, sauf que l'argument est passé en tant que paramètre `ref`. La valeur de l'argument sous-jacent, `n`, est modifiée quand `x` est modifiée dans la méthode.

C#

```
int n = 5;
System.Console.WriteLine("The value before calling the method: {0}", n);

SquareIt(ref n);  // Passing the variable by reference.
System.Console.WriteLine("The value after calling the method: {0}", n);

// Keep the console window open in debug mode.
System.Console.WriteLine("Press any key to exit.");
System.Console.ReadKey();

static void SquareIt(ref int x)
// The parameter x is passed by reference.
// Changes to x will affect the original value of x.
{
    x *= x;
    System.Console.WriteLine("The value inside the method: {0}", x);
}
/* Output:
    The value before calling the method: 5
    The value inside the method: 25
    The value after calling the method: 25
*/
```

Dans cet exemple, ce n'est pas la valeur de `n` qui est passée, mais plutôt une référence à `n`. Le paramètre `x` n'est pas un int ; il s'agit d'une référence à un `int`, dans le cas présent une référence à `n`. Ainsi, quand `x` est mis au carré à l'intérieur de la méthode, ce qui est réellement mis au carré est ce à quoi `x` fait référence, c'est-à-dire `n`.

Passer un type référence par valeur

Lorsque vous passez un type *référencepar valeur* :

- Si la méthode affecte le paramètre pour faire référence à un autre objet, ces modifications **ne sont pas** visibles à partir de l'appelant.

- Si la méthode modifie l'état de l'objet auquel le paramètre fait référence, ces modifications **sont visibles** à partir de l'appelant.

L'exemple suivant illustre le passage d'un paramètre de type référence (`arr`) par valeur, à une méthode (`Change`). Sachant que le paramètre est une référence à `arr`, il est possible de modifier les valeurs des éléments du tableau. Cependant, la tentative de réassignation du paramètre à un emplacement de mémoire différent n'aboutit qu'à l'intérieur de la méthode et n'affecte pas la variable d'origine, `arr`.

C#

```
int[] arr = { 1, 4, 5 };
System.Console.WriteLine("Inside Main, before calling the method, the first
element is: {0}", arr[0]);

Change(arr);
System.Console.WriteLine("Inside Main, after calling the method, the first
element is: {0}", arr[0]);

static void Change(int[] pArray)
{
    pArray[0] = 888;  // This change affects the original element.
    pArray = new int[5] { -3, -1, -2, -3, -4 };   // This change is local.
    System.Console.WriteLine("Inside the method, the first element is: {0}",
pArray[0]);
}
/* Output:
    Inside Main, before calling the method, the first element is: 1
    Inside the method, the first element is: -3
    Inside Main, after calling the method, the first element is: 888
*/
```

Dans l'exemple précédent, le tableau `arr`, qui est un type référence, est passé à la méthode sans le paramètre `ref`. Dans ce cas, une copie de la référence, qui pointe vers `arr`, est passée à la méthode. La sortie montre que la méthode peut modifier le contenu d'un élément du tableau, dans ce cas de `1` à `888`. Cependant, si vous allouez une nouvelle portion de mémoire via l'opérateur new à l'intérieur de la méthode `Change`, la variable `pArray` fait référence à un nouveau tableau. Par conséquent, les modifications apportées à la suite de cette opération n'ont pas d'effet sur le tableau d'origine (`arr`), qui est créé à l'intérieur de `Main`. En fait, deux tableaux sont créés dans cet exemple : un à l'intérieur de `Main` et un autre à l'intérieur de la méthode `Change`.

Passer un type référence par référence

Lorsque vous passez un type *référencepar référence* :

- Si la méthode affecte le paramètre pour faire référence à un autre objet, ces modifications **sont visibles** à partir de l'appelant.
- Si la méthode modifie l'état de l'objet auquel le paramètre fait référence, ces modifications **sont visibles** à partir de l'appelant.

L'exemple suivant est identique au précédent, sauf que le mot clé `ref` est ajouté à l'en-tête et à l'appel de la méthode. Les modifications qui se produisent dans la méthode affectent la variable d'origine dans le programme appelant.

C#

```
int[] arr = { 1, 4, 5 };
System.Console.WriteLine("Inside Main, before calling the method, the first
element is: {0}", arr[0]);

Change(ref arr);
System.Console.WriteLine("Inside Main, after calling the method, the first
element is: {0}", arr[0]);

static void Change(ref int[] pArray)
{
    // Both of the following changes will affect the original variables:
    pArray[0] = 888;
    pArray = new int[5] { -3, -1, -2, -3, -4 };
    System.Console.WriteLine("Inside the method, the first element is: {0}",
pArray[0]);
}
/* Output:
    Inside Main, before calling the method, the first element is: 1
    Inside the method, the first element is: -3
    Inside Main, after calling the method, the first element is: -3
*/
```

Les modifications qui se produisent à l'intérieur de la méthode affectent le tableau d'origine dans `Main`. En fait, le tableau d'origine est réalloué à l'aide de l'opérateur `new`. Par conséquent, après l'appel à la méthode `Change`, toute référence à `arr` pointe vers le tableau à cinq éléments, qui est créé dans la méthode `Change`.

Étendue des références et des valeurs

Les méthodes peuvent stocker les valeurs des paramètres dans des champs. Lorsque les paramètres sont passés par valeur, cela est toujours fait de façon sécurisée. Les valeurs sont copiées et les types de référence sont accessibles lorsqu'ils sont stockés dans un champ. La transmission de paramètres par référence de façon sûre nécessite que le compilateur définisse quand il est sûr d'affecter une référence à une nouvelle variable. Pour chaque expression, le compilateur définit une *étendue* qui limite l'accès à une

expression ou à une variable. Le compilateur utilise deux étendues : *safe_to_escape* et *ref_safe_to_escape*.

- L'étendue *safe_to_escape* définit l'étendue dans laquelle n'importe quelle expression est accessible en toute sécurité.
- L'étendue *ref_safe_to_escape* définit l'étendue dans laquelle une *référence* à n'importe quelle expression peut être accessible ou modifiée en toute sécurité.

De manière informelle, vous pouvez considérer ces étendues comme le mécanisme permettant de vous assurer que votre code n'accède jamais ou ne modifie jamais une référence qui n'est plus valide. Une référence est valide tant qu'elle fait référence à un objet ou à un struct valide. L'étendue *safe_to_escape* définit quand une variable peut être affectée ou réaffectée. L'étendue *ref_safe_to_escape* définit quand une variable peut être affectée par *référence* ou réaffectée par *référence*. L'affectation affecte une variable à une nouvelle valeur ; *l'affectation par référence* affecte la variable pour *faire référence à* un autre emplacement de stockage.

Modificateurs

Les paramètres déclarés pour une méthode sans in, ref ou out sont passés à la méthode appelée par valeur. Les modificateurs `ref`, `in` et `out` diffèrent dans les règles d'affectation :

- L'argument d'un paramètre `ref` doit être définitivement affecté. La méthode appelée peut réaffecter ce paramètre.
- L'argument d'un paramètre `in` doit être définitivement affecté. La méthode appelée ne peut pas réaffecter ce paramètre.
- L'argument d'un paramètre `out` n'a pas besoin d'être définitivement affecté. La méthode appelée doit affecter le paramètre.

Cette section décrit les mots clés que vous pouvez utiliser pour déclarer des paramètres de méthode :

- params spécifie que ce paramètre peut prendre un nombre variable d'arguments.
- in spécifie que ce paramètre est passé par référence, mais qu'il est lu uniquement par la méthode appelée.
- ref spécifie que ce paramètre est passé par référence et qu'il peut être lu ou écrit par la méthode appelée.
- out spécifie que ce paramètre est passé par référence et qu'il est écrit par la méthode appelée.

Voir aussi

- Référence C#
- Mots clés C#
- Liste des arguments dans la Spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

params (référence C#)

Article • 07/04/2023

Avec le mot clé `params`, vous pouvez spécifier un paramètre de méthode qui prend un nombre variable d'arguments. Le type de paramètres doit être un tableau à une seule dimension.

Dans une déclaration de méthode, vous ne pouvez pas spécifier de paramètre supplémentaire après le mot clé `params` et vous pouvez utiliser un seul mot clé `params`.

Si le type déclaré du paramètre `params` n'est pas un tableau unidimensionnel, l'erreur de compilateur CS0225 se produit.

Lorsque vous appelez une méthode avec un paramètre `params`, vous pouvez passer :

- Une liste séparée par des virgules d'arguments du type des éléments de tableau.
- Un tableau d'arguments du type spécifié.
- Aucun argument. Si vous n'envoyez aucun argument, la longueur de la liste `params` est égale à zéro.

Exemple

L'exemple suivant montre différentes façons d'envoyer des arguments à un paramètre `params`.

C#

```
public class MyClass
{
    public static void UseParams(params int[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    public static void UseParams2(params object[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }
```

```
    static void Main()
    {
        // You can send a comma-separated list of arguments of the
        // specified type.
        UseParams(1, 2, 3, 4);
        UseParams2(1, 'a', "test");

        // A params parameter accepts zero or more arguments.
        // The following calling statement displays only a blank line.
        UseParams2();

        // An array argument can be passed, as long as the array
        // type matches the parameter type of the method being called.
        int[] myIntArray = { 5, 6, 7, 8, 9 };
        UseParams(myIntArray);

        object[] myObjArray = { 2, 'b', "test", "again" };
        UseParams2(myObjArray);

        // The following call causes a compiler error because the object
        // array cannot be converted into an integer array.
        //UseParams(myObjArray);

        // The following call does not cause an error, but the entire
        // integer array becomes the first element of the params array.
        UseParams2(myIntArray);
    }
}
/*
Output:
    1 2 3 4
    1 a test

    5 6 7 8 9
    2 b test again
    System.Int32[]
*/
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#

- Paramètres de méthodes

in, modificateur de paramètre (référence C#)

Article • 07/04/2023

Le mot clé `in` provoque le passage d'arguments par référence, mais garantit que l'argument n'est pas modifié. Il fait du paramètre formel un alias de l'argument, qui doit être une variable. En d'autres termes, toute opération portant sur le paramètre est effectuée sur l'argument. Il est similaire aux mots clés ref ou out, sauf que les arguments `in` ne peuvent pas être modifiés par la méthode appelée. Alors que la modification des arguments `ref` est facultative, les arguments `out` doivent être modifiés par la méthode appelée ; ces modifications sont observables dans le contexte d'appel.

C#

```
int readonlyArgument = 44;
InArgExample(readonlyArgument);
Console.WriteLine(readonlyArgument);     // value is still 44

void InArgExample(in int number)
{
    // Uncomment the following line to see error CS8331
    //number = 19;
}
```

L'exemple précédent montre que le modificateur `in` n'est généralement pas nécessaire sur le site d'appel. Il l'est uniquement dans la déclaration de méthode.

> ⓘ **Notes**
>
> Le mot clé `in` peut également être utilisé avec un paramètre de type générique pour spécifier que le paramètre de type est contravariant, dans le cadre d'une instruction `foreach` ou d'une clause `join` dans une requête LINQ. Pour plus d'informations sur l'utilisation du mot clé `in` dans ces contextes, consultez **in**, qui fournit des liens vers toutes ces utilisations.

Les variables passées comme des arguments `in` doivent être initialisées avant d'être passées dans un appel de méthode. Toutefois, la méthode appelée ne peut pas attribuer de valeur ou modifier l'argument.

Bien que les modificateurs de paramètre `in`, `out` et `ref` soient considérés comme faisant partie d'une signature, les membres déclarés dans un type unique ne peuvent

pas différer uniquement dans la signature par `in`, `ref` et `out`. Par conséquent, les méthodes ne peuvent pas être surchargées si la seule différence est que l'une d'elles accepte un argument `ref` ou `out` et que l'autre accepte un argument `in`. Le code suivant, par exemple, ne se compilera pas :

C#

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on in, ref and out".
    public void SampleMethod(in int i) { }
    public void SampleMethod(ref int i) { }
}
```

Une surcharge basée sur la présence de `in` est autorisée :

C#

```
class InOverloads
{
    public void SampleMethod(in int i) { }
    public void SampleMethod(int i) { }
}
```

Règles de résolution de surcharge

Vous pouvez comprendre les règles de résolution de surcharge pour les méthodes par valeur par rapport à celles qui utilisent des arguments `in` en comprenant pourquoi on utilise des arguments `in`. Vous pouvez optimiser les performances en définissant des méthodes à l'aide de paramètres `in`. Certains arguments de type `struct` peuvent être volumineux et, quand les méthodes sont appelées dans des boucles serrées ou des chemins de code critiques, le coût de la copie de ces structures est crucial. Les méthodes déclarent des paramètres `in` pour spécifier que des arguments peuvent être passés par référence en toute sécurité, car la méthode appelée ne modifie pas l'état de l'argument. Le passage de ces arguments par référence permet d'éviter une copie (potentiellement) coûteuse.

La spécification de `in` pour des arguments au niveau de l'appel de site est généralement facultative. Il n'existe aucune différence sémantique entre le passage d'arguments par valeur et leur passage par référence à l'aide du modificateur `in`. Le modificateur `in` sur le site d'appel est facultatif, car vous n'avez pas besoin d'indiquer que la valeur de l'argument peut être changée. Vous ajoutez explicitement le

modificateur `in` sur le site d'appel pour vérifier que l'argument est passé par référence, non par valeur. L'utilisation explicite de `in` a les deux effets suivants :

Tout d'abord, la spécification de `in` sur le site d'appel force le compilateur à sélectionner une méthode définie avec un paramètre `in` correspondant. Sinon, quand deux méthodes diffèrent uniquement par la présence de `in`, la surcharge par valeur convient mieux.

Ensuite, la spécification de `in` déclare votre intention de passer un argument par référence. L'argument utilisé avec `in` doit représenter un emplacement directement référençable. Les mêmes règles générales pour les arguments `out` et `ref` s'appliquent : vous ne pouvez pas utiliser de constantes, de propriétés ordinaires ou d'autres expressions qui produisent des valeurs. Sinon, l'omission de `in` sur le site d'appel informe le compilateur que vous l'autorisez à créer une variable temporaire à passer par référence en lecture seule à la méthode. Le compilateur crée une variable temporaire pour surmonter plusieurs restrictions avec les arguments `in` :

- Une variable temporaire autorise des constantes au moment de la compilation comme paramètres `in`.
- Une variable temporaire autorise des propriétés ou d'autres expressions pour les paramètres `in`.
- Une variable temporaire autorise des arguments pour lesquels il existe une conversion implicite de type d'argument en type de paramètre.

Dans toutes les instances précédentes, le compilateur crée une variable temporaire qui stocke la valeur de la constante, la propriété ou une autre expression.

Le code suivant illustre ces règles :

C#

```
static void Method(in int argument)
{
    // implementation removed
}

Method(5); // OK, temporary variable created.
Method(5L); // CS1503: no implicit conversion from long to int
short s = 0;
Method(s); // OK, temporary int created with the value 0
Method(in s); // CS1503: cannot convert from in short to in int
int i = 42;
Method(i); // passed by readonly reference
Method(in i); // passed by readonly reference, explicitly using `in`
```

Supposons à présent qu'une autre méthode utilisant des arguments par valeur est disponible. Les résultats changent comme illustré dans le code suivant :

```C#
static void Method(int argument)
{
    // implementation removed
}

static void Method(in int argument)
{
    // implementation removed
}

Method(5); // Calls overload passed by value
Method(5L); // CS1503: no implicit conversion from long to int
short s = 0;
Method(s); // Calls overload passed by value.
Method(in s); // CS1503: cannot convert from in short to in int
int i = 42;
Method(i); // Calls overload passed by value
Method(in i); // passed by readonly reference, explicitly using `in`
```

Le seul appel de méthode dans lequel l'argument est passé par référence est le dernier.

> ⓘ Notes
>
> Le code précédent utilise `int` comme type d'argument par souci de simplicité. Comme `int` n'est pas plus volumineux qu'une référence dans la plupart des ordinateurs modernes, il n'y aucun avantage à passer un seul `int` comme référence en lecture seule.

Limitations sur les paramètres `in`

Vous ne pouvez pas utiliser les mots clés `in`, `ref` ou `out` pour les types de méthodes suivants :

- Méthodes async, que vous définissez à l'aide du modificateur async.
- Les méthodes Iterator, qui incluent une instruction yield return ou `yield break`.
- Le premier argument d'une méthode d'extension ne peut pas avoir le modificateur `in`, sauf si cet argument est une struct.
- Premier argument d'une méthode d'extension où cet argument est un type générique (même quand ce type est contraint d'être une struct.)

Vous pouvez en savoir plus sur le modificateur `in`, la façon dont il diffère de `ref` et `out` dans l'article sur les allocations.

Spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

ref (Informations de référence sur C#)

Article • 28/06/2023

Vous utilisez le mot clé `ref` dans les contextes suivants :

- Dans une signature de méthode et dans un appel de méthode, pour passer un argument à une méthode par référence. Pour plus d'informations, consultez Passage d'un argument par référence.
- Dans une signature de méthode, pour retourner une valeur à l'appelant par référence. Pour plus d'informations, consultez Valeurs de retour de référence.
- Dans une déclaration d'une variable locale, pour déclarer une variable de référence. Pour plus d'informations, consultez la section Variables de référence de l'article Instructions de déclaration.
- En tant que partie d'une expression ref conditionnelle ou d'un opérateur d'assignation par référence.
- Dans une déclaration de `struct`, pour déclarer un `ref struct`. Pour plus d'informations, consultez l'article Types de structures ref.
- Dans une définition `ref struct`, pour déclarer un champ `ref`. Pour plus d'informations, consultez la section Champs ref de l'article Types de structures ref.

Passage d'un argument par référence

Quand il est utilisé dans la liste de paramètres d'une méthode, le mot clé `ref` indique qu'un argument est passé par référence, et non par valeur. Le mot clé `ref` fait du paramètre formel un alias de l'argument, qui doit être une variable. En d'autres termes, toute opération portant sur le paramètre est effectuée sur l'argument.

Par exemple, supposons que l'appelant transmette une expression de variable locale ou une expression d'accès d'élément de tableau. La méthode appelée peut ensuite remplacer l'objet auquel le paramètre ref fait référence. Dans ce cas, la variable locale de l'appelant ou l'élément de tableau fait référence au nouvel objet lorsque la méthode est retournée.

> ⓘ **Notes**
>
> Ne confondez pas le concept de passage par référence avec celui de types de référence. Les deux concepts ne sont pas identiques. Un paramètre de méthode peut être modifié par `ref`, qu'il s'agisse d'un type valeur ou d'un type référence. Il n'y a aucun boxing d'un type valeur lorsqu'il est passé par référence.

Pour utiliser un paramètre `ref`, la définition de la méthode et la méthode d'appel doivent utiliser explicitement le mot clé `ref`, comme indiqué dans l'exemple suivant. (Sauf que la méthode appelante peut omettre `ref` lors de l'appel COM.)

C#

```
void Method(ref int refArgument)
{
    refArgument = refArgument + 44;
}

int number = 1;
Method(ref number);
Console.WriteLine(number);
// Output: 45
```

Un argument passé à un paramètre `ref` ou `in` doit être initialisé avant son passage. Cette exigence diffère des paramètres out , dont les arguments n'ont pas à être initialisés explicitement avant leur passage.

Les membres d'une classe ne peuvent pas avoir de signatures qui diffèrent uniquement par `ref`, `in` ou `out`. Une erreur de compilation se produit si la seule différence entre deux membres d'un type est que l'un d'eux a un paramètre `ref` et que l'autre un a un paramètre `out` ou `in`. Le code suivant, par exemple, ne se compile pas.

C#

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on ref and out".
    public void SampleMethod(out int i) { }
    public void SampleMethod(ref int i) { }
}
```

Toutefois, les méthodes peuvent être surchargées quand une méthode a un paramètre `ref`, `in` ou `out` et que l'autre a un paramètre transmis par valeur, comme illustré dans l'exemple suivant.

C#

```
class RefOverloadExample
{
    public void SampleMethod(int i) { }
    public void SampleMethod(ref int i) { }
}
```

Dans d'autres situations nécessitant une correspondance de signature, comme le masquage ou la substitution, `in`, `ref` et `out` font partie de la signature et ne correspondent pas l'un à l'autre.

Les propriétés ne sont pas des variables. Elles sont des méthodes et ne peuvent pas être transmises aux paramètres `ref`.

Vous ne pouvez pas utiliser les mots clés `ref`, `in` ou `out` pour les types de méthodes suivants :

- Méthodes async, que vous définissez à l'aide du modificateur async.
- Les méthodes Iterator, qui incluent une instruction yield return ou `yield break`.

Les méthodes d'extension ont également des restrictions sur l'utilisation de ces mots clés :

- Le `out` mot clé ne peut pas être utilisé sur le premier argument d'une méthode d'extension.
- Le `ref` mot clé ne peut pas être utilisé sur le premier argument d'une méthode d'extension lorsque l'argument n'est pas un struct, ou qu'un type générique n'est pas contraint d'être un struct.
- Le `in` mot clé ne peut pas être utilisé, sauf si le premier argument est un struct. Le `in` mot clé ne peut pas être utilisé sur n'importe quel type générique, même s'il est contraint d'être un struct.

Passage d'un argument par référence : exemple

Les exemples précédents passent les types valeur par référence. Vous pouvez également utiliser le mot clé `ref` pour passer les types référence par référence. Le passage d'un type référence par référence permet à la méthode appelée de remplacer l'objet auquel fait référence le paramètre de référence dans l'appelant. L'emplacement de stockage de l'objet est passé à la méthode comme valeur du paramètre de référence. Si vous modifiez la valeur de l'emplacement de stockage du paramètre (pour pointer vers un nouvel objet), vous modifiez également l'emplacement de stockage auquel fait référence l'appelant. L'exemple suivant passe une instance d'un type référence en tant que paramètre `ref`.

C#

```
class Product
{
    public Product(string name, int newID)
    {
```

```
        ItemName = name;
        ItemID = newID;
    }

    public string ItemName { get; set; }
    public int ItemID { get; set; }
}

private static void ChangeByReference(ref Product itemRef)
{
    // Change the address that is stored in the itemRef parameter.
    itemRef = new Product("Stapler", 99999);

    // You can change the value of one of the properties of
    // itemRef. The change happens to item in Main as well.
    itemRef.ItemID = 12345;
}

private static void ModifyProductsByReference()
{
    // Declare an instance of Product and display its initial values.
    Product item = new Product("Fasteners", 54321);
    System.Console.WriteLine("Original values in Main.  Name: {0}, ID:
{1}\n",
        item.ItemName, item.ItemID);

    // Pass the product instance to ChangeByReference.
    ChangeByReference(ref item);
    System.Console.WriteLine("Back in Main.  Name: {0}, ID: {1}\n",
        item.ItemName, item.ItemID);
}

// This method displays the following output:
// Original values in Main.  Name: Fasteners, ID: 54321
// Back in Main.  Name: Stapler, ID: 12345
```

Pour plus d'informations sur la manière de passer des types référence par valeur et par référence, consultez Passage de paramètres de type référence .

Valeurs de retour de référence

Les valeurs de retour de référence (ou retours ref) sont des valeurs qu'une méthode retourne par référence à l'appelant. Autrement dit, l'appelant peut modifier la valeur retournée par une méthode, et cette modification est reflétée dans l'état de l'objet dans la méthode appelée.

Une valeur de retour de référence est définie à l'aide du mot clé `ref` :

- Dans la signature de la méthode. Par exemple, la signature de méthode suivante indique que la méthode `GetCurrentPrice` retourne une valeur Decimal par référence.

```
C#

public ref decimal GetCurrentPrice()
```

- Entre le jeton `return` et la variable retournée dans une instruction `return` dans la méthode. Par exemple :

```
C#

return ref DecimalArray[0];
```

Pour que l'appelant puisse modifier l'état de l'objet, la valeur de retour de référence doit être stockée dans une variable qui est explicitement définie comme variable de référence.

Voici un exemple de retour ref plus complet, montrant à la fois la signature de méthode et le corps de méthode.

```
C#

public static ref int Find(int[,] matrix, Func<int, bool> predicate)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
        for (int j = 0; j < matrix.GetLength(1); j++)
            if (predicate(matrix[i, j]))
                return ref matrix[i, j];
    throw new InvalidOperationException("Not found");
}
```

La méthode appelée peut également déclarer la valeur renvoyée en tant que `ref readonly` pour renvoyer la valeur par référence et faire en sorte que le code d'appel ne puisse pas modifier la valeur renvoyée. La méthode appelante peut éviter de copier la valeur retournée en stockant cette valeur dans une variable de référence `ref readonly` locale.

Pour obtenir un exemple, voir Exemple de valeurs de retour de référence et de variables locales ref.

Exemple de retours ref et de variables locales ref

L'exemple suivant définit une classe `Book` qui a deux champs String, `Title` et `Author`. Il définit également une classe `BookCollection` qui inclut un tableau privé d'objets `Book`. Des objets livres individuels sont retournés par référence en appelant sa méthode `GetBookByTitle`.

C#

```
public class Book
{
    public string Author;
    public string Title;
}

public class BookCollection
{
    private Book[] books = { new Book { Title = "Call of the Wild, The",
Author = "Jack London" },
                             new Book { Title = "Tale of Two Cities, A", Author =
"Charles Dickens" }
                           };
    private Book nobook = null;

    public ref Book GetBookByTitle(string title)
    {
        for (int ctr = 0; ctr < books.Length; ctr++)
        {
            if (title == books[ctr].Title)
                return ref books[ctr];
        }
        return ref nobook;
    }

    public void ListBooks()
    {
        foreach (var book in books)
        {
            Console.WriteLine($"{book.Title}, by {book.Author}");
        }
        Console.WriteLine();
    }
}
```

Quand l'appelant stocke la valeur retournée par la méthode `GetBookByTitle` comme variable locale ref, les modifications apportées par l'appelant à la valeur de retour sont reflétées dans l'objet `BookCollection`, comme le montre l'exemple suivant.

C#

```
var bc = new BookCollection();
bc.ListBooks();

ref var book = ref bc.GetBookByTitle("Call of the Wild, The");
if (book != null)
    book = new Book { Title = "Republic, The", Author = "Plato" };
bc.ListBooks();
// The example displays the following output:
//       Call of the Wild, The, by Jack London
//       Tale of Two Cities, A, by Charles Dickens
//
//       Republic, The, by Plato
//       Tale of Two Cities, A, by Charles Dickens
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Informations de référence sur C#
- Mots clés C#
- Paramètres de la méthode

out, modificateur de paramètre (référence C#)

Article • 07/04/2023

Le mot clé `out` entraîne le passage des arguments par référence. Il fait du paramètre formel un alias de l'argument, qui doit être une variable. En d'autres termes, toute opération portant sur le paramètre est effectuée sur l'argument. Il est similaire au mot clé ref, à la différence que `ref` nécessite que la variable soit initialisée avant d'être passée. Il est également similaire au mot clé in, à la différence que `in` n'autorise pas la méthode appelée à modifier la valeur d'argument. Pour utiliser un paramètre `out`, la définition de la méthode et la méthode d'appel doivent utiliser explicitement le mot clé `out`. Par exemple :

C#

```
int initializeInMethod;
OutArgExample(out initializeInMethod);
Console.WriteLine(initializeInMethod);     // value is now 44

void OutArgExample(out int number)
{
    number = 44;
}
```

> ⓘ **Notes**
>
> Le mot clé `out` peut également être utilisé avec un paramètre de type générique pour spécifier que le paramètre de type est covariant. Pour plus d'informations sur l'utilisation du mot clé `out` dans ce contexte, consultez **out, modificateur générique**.

Les variables passées comme arguments `out` n'ont pas à être initialisées avant d'être passées dans un appel de méthode. Toutefois, la méthode appelée doit assigner une valeur avant le retour de la méthode.

Les mots clés `in`, `ref` et `out` ne sont pas considérés comme faisant partie de la signature de méthode à des fins de résolution de surcharge. Par conséquent, les méthodes ne peuvent pas être surchargées si la seule différence est que l'une d'elles accepte un argument `ref` ou `in` et que l'autre accepte un argument `out`. Le code suivant, par exemple, ne se compilera pas :

C#

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on ref and out".
    public void SampleMethod(out int i) { }
    public void SampleMethod(ref int i) { }
}
```

Toutefois, la surcharge est autorisée si une méthode prend un argument `ref`, `in` ou `out`, et que l'autre méthode n'a aucun de ces modificateurs, comme suit :

C#

```
class OutOverloadExample
{
    public void SampleMethod(int i) { }
    public void SampleMethod(out int i) => i = 5;
}
```

Le compilateur choisit la meilleure surcharge en mettant en correspondance les modificateurs de paramètre sur le site d'appel et les modificateurs de paramètre utilisés dans l'appel de méthode.

Les propriétés ne sont pas des variables et, par conséquent, ne peuvent pas être passées en tant que paramètres `out`.

Vous ne pouvez pas utiliser les mots clés `in`, `ref` ou `out` pour les types de méthodes suivants :

- Méthodes async, que vous définissez à l'aide du modificateur async.
- Les méthodes Iterator, qui incluent une instruction yield return ou `yield break`.

En outre, les méthodes d'extension ont les restrictions suivantes :

- Le `out` mot clé ne peut pas être utilisé sur le premier argument d'une méthode d'extension.
- Le `ref` mot clé ne peut pas être utilisé sur le premier argument d'une méthode d'extension lorsque l'argument n'est pas un struct, ou qu'un type générique n'est pas contraint d'être un struct.
- Le `in` mot clé ne peut pas être utilisé, sauf si le premier argument est un struct. Le `in` mot clé ne peut pas être utilisé sur un type générique, même s'il est contraint d'être un struct.

Déclarer des paramètres `out`

Il arrive souvent que l'on déclare une méthode avec des arguments `out` pour qu'elle retourne plusieurs valeurs. Envisagez des tuples de valeur pour des scénarios similaires. L'exemple suivant utilise `out` pour retourner trois variables avec un seul appel de méthode. Le troisième argument est assigné à la valeur null. Cela permet aux méthodes de retourner des valeurs le cas échéant.

C#

```
void Method(out int answer, out string message, out string? stillNull)
{
    answer = 44;
    message = "I've been returned";
    stillNull = null;
}

int argNumber;
string argMessage;
string? argDefault;
Method(out argNumber, out argMessage, out argDefault);
Console.WriteLine(argNumber);
Console.WriteLine(argMessage);
Console.WriteLine(argDefault == null);

// The example displays the following output:
//      44
//      I've been returned
//      True
```

Appel d'une méthode avec un argument `out`

Vous pouvez déclarer une variable dans une instruction distincte avant de la passer en tant qu'`out` argument. L'exemple suivant déclare une variable nommée `number` avant de la passer à la méthode Int32.TryParse, qui tente de convertir une chaîne en nombre.

C#

```
string numberAsString = "1640";

int number;
if (Int32.TryParse(numberAsString, out number))
    Console.WriteLine($"Converted '{numberAsString}' to {number}");
else
    Console.WriteLine($"Unable to convert '{numberAsString}'");
```

```
// The example displays the following output:
//       Converted '1640' to 1640
```

Vous pouvez également déclarer la `out` variable dans la liste d'arguments de l'appel de méthode au lieu de le faire dans une déclaration de variable distincte. Cette technique rend le code plus simple et plus lisible, et vous empêche également d'assigner par inadvertance une valeur à la variable avant l'appel de méthode. L'exemple suivant est semblable au précédent, sauf qu'il définit la variable `number` dans l'appel de la méthode Int32.TryParse.

C#

```
string numberAsString = "1640";

if (Int32.TryParse(numberAsString, out int number))
    Console.WriteLine($"Converted '{numberAsString}' to {number}");
else
    Console.WriteLine($"Unable to convert '{numberAsString}'");
// The example displays the following output:
//       Converted '1640' to 1640
```

Dans l'exemple précédent, la variable `number` est fortement typée en `int`. Vous pouvez également déclarer une variable locale implicitement typée, comme dans l'exemple suivant.

C#

```
string numberAsString = "1640";

if (Int32.TryParse(numberAsString, out var number))
    Console.WriteLine($"Converted '{numberAsString}' to {number}");
else
    Console.WriteLine($"Unable to convert '{numberAsString}'");
// The example displays the following output:
//       Converted '1640' to 1640
```

Spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Déclaration de variable inlined (de règle de style IDE0018)

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Paramètres de méthodes

espace de noms

Article • 07/04/2023

Le mot clé `namespace` est utilisé pour déclarer une portée qui contient un ensemble d'objets connexes. Vous pouvez utiliser un espace de noms pour organiser les éléments de code et créer des types globaux uniques.

C#

```
namespace SampleNamespace
{
    class SampleClass { }

    interface ISampleInterface { }

    struct SampleStruct { }

    enum SampleEnum { a, b }

    delegate void SampleDelegate(int i);

    namespace Nested
    {
        class SampleClass2 { }
    }
}
```

Les déclarations d'espace de noms délimitées aux fichiers vous permettent de déclarer que tous les types d'un fichier se trouvent dans un espace de noms unique. Les déclarations d'espace de noms délimitées aux fichiers sont disponibles avec C# 10. L'exemple suivant est similaire à l'exemple précédent, mais il utilise une déclaration d'espace de noms délimitée par un fichier :

C#

```
using System;

namespace SampleFileScopedNamespace;

class SampleClass { }

interface ISampleInterface { }

struct SampleStruct { }

enum SampleEnum { a, b }

delegate void SampleDelegate(int i);
```

L'exemple précédent n'inclut pas d'espace de noms imbriqué. Les espaces de noms délimités aux fichiers ne peuvent pas inclure de déclarations d'espace de noms supplémentaires. Vous ne pouvez pas déclarer un espace de noms imbriqué ou un deuxième espace de noms délimité au fichier :

C#

```
namespace SampleNamespace;

class AnotherSampleClass
{
    public void AnotherSampleMethod()
    {
        System.Console.WriteLine(
            "SampleMethod inside SampleNamespace");
    }
}

namespace AnotherNamespace; // Not allowed!

namespace ANestedNamespace // Not allowed!
{
   // declarations...
}
```

Dans un espace de noms, vous pouvez déclarer zéro ou plusieurs des types suivants :

- class
- interface
- struct
- enum
- delegate
- les espaces de noms imbriqués peuvent être déclarés, sauf dans les déclarations d'espace de noms délimitées aux fichiers

Le compilateur ajoute un espace de noms par défaut. Cet espace de noms sans nom, parfois appelé espace de noms global, est présent dans chaque fichier. Il contient des déclarations non incluses dans un espace de noms déclaré. Tout identificateur dans l'espace de noms global peut être utilisé dans un espace de noms nommé.

Les espaces de noms ont implicitement un accès public. Consultez Modificateurs d'accès pour connaître les modificateurs d'accès que vous pouvez assigner à des éléments dans un espace de noms.

Il est possible de définir un espace de noms dans deux déclarations ou plus. Par exemple, l'exemple suivant définit deux classes comme appartenant à l'espace de noms `MyCompany` :

C#

```
namespace MyCompany.Proj1
{
    class MyClass
    {
    }
}

namespace MyCompany.Proj1
{
    class MyClass1
    {
    }
}
```

L'exemple suivant montre comment appeler une méthode statique dans un espace de noms imbriqué.

C#

```
namespace SomeNameSpace
{
    public class MyClass
    {
        static void Main()
        {
            Nested.NestedNameSpaceClass.SayHello();
        }
    }

    // a nested namespace
    namespace Nested
    {
        public class NestedNameSpaceClass
        {
            public static void SayHello()
            {
                Console.WriteLine("Hello");
            }
        }
    }
}
// Output: Hello
```

spécification du langage C#

Pour plus d'informations, voir la section Espace de noms de la spécification du langage C#. Pour plus d'informations sur les déclarations d'espace de noms délimitées aux fichiers, consultez la spécification des fonctionnalités.

Voir aussi

- Préférences de déclaration d'espace de noms (IDE0160 et IDE0161)
- Informations de référence sur C#
- Mots clés C#
- using
- using static
- Qualificateur d'alias d'espace de noms::
- Espaces de noms

using (Référence C#)

Article • 21/03/2023

Le mot clé `using` a deux utilisations principales :

- L'instruction using définit une étendue à la fin de laquelle un objet est supprimé.
- La directive using crée un alias pour un espace de noms ou importe des types définis dans d'autres espaces de noms.

Voir aussi

- Informations de référence sur C#
- Mots clés C#

using (directive)

Article • 29/05/2023

La directive `using` vous permet d'utiliser des types définis dans un espace de noms sans spécifier l'espace de noms complet de ce type. Dans sa forme de base, la directive `using` importe tous les types à partir d'un espace de noms unique, comme illustré dans l'exemple suivant :

```
C#

using System.Text;
```

Vous pouvez appliquer deux modificateurs à une directive `using` :

- Le modificateur `global` a le même effet que l'ajout de la même directive `using` à chaque fichier source dans votre projet. Ce modificateur a été introduit dans C# 10.
- Le modificateur `static` importe les membres `static` et les types imbriqués à partir d'un type unique, plutôt que d'importer tous les types dans un espace de noms.

Vous pouvez combiner les deux modificateurs pour importer les membres statiques à partir d'un type dans tous les fichiers sources de votre projet.

Vous pouvez également créer un alias pour un espace de noms ou un type avec une *directive d'alias using*.

```
C#

using Project = PC.MyCompany.Project;
```

Vous pouvez utiliser le modificateur `global` sur une *directive d'alias using*.

> ⓘ **Notes**
>
> Le mot clé `using` est également utilisé pour créer des *instructions usingIDisposable, qui garantissent que les objets* tels que les fichiers et les polices sont gérés correctement. Pour plus d'informations sur *l'instruction using*, consultez **using, instruction**.

La portée d'une directive `using` sans le modificateur `global` est le fichier dans lequel elle apparaît.

La directive `using` peut être placée :

- Au début d'un fichier de code source, avant les définitions de type ou d'espace de noms.
- Dans n'importe quel espace de noms, mais avant tous les espaces de noms ou types déclarés dans cet espace de noms, sauf si le modificateur `global` est utilisé, auquel cas la directive doit apparaître avant toutes les déclarations d'espace de noms et de type.

Sinon, une erreur de compilateur CS1529 est générée.

Créez une directive `using` pour utiliser les types dans un espace de noms sans avoir à spécifier l'espace de noms. Une directive `using` ne vous donne pas accès à des espaces de noms imbriqués dans l'espace de noms que vous spécifiez. Les espaces de noms sont organisés en deux catégories : définis par l'utilisateur et définis par le système. Les espaces de noms définis par l'utilisateur sont des espaces de noms définis dans votre code. Pour obtenir la liste des espaces de noms définis par le système, consultez Navigateur d'API .NET.

modificateur global

L'ajout du modificateur `global` à une directive `using`signifie que l'utilisation est appliquée à tous les fichiers de la compilation (généralement un projet). La directive `global using` a été ajoutée dans C# 10 Sa syntaxe est la suivante :

```
C#

global using <fully-qualified-namespace>;
```

où *fully-qualified-namespace* est le nom complet de l'espace de noms dont les types peuvent être référencés sans spécifier l'espace de noms.

Une directive *using globale* peut apparaître au début de n'importe quel fichier de code source. Toutes les directives `global using` d'un seul fichier doivent apparaître avant :

- Toutes les directives `using` sans le modificateur `global`.
- Toutes les déclarations d'espace de noms et de type dans le fichier.

Vous pouvez ajouter des directives `global using` à n'importe quel fichier source. En règle générale, il convient de les conserver dans un emplacement unique. L'ordre des directives `global using` n'a pas d'importance, soit dans un fichier unique, soit entre des fichiers.

Le modificateur `global` peut être combiné avec le modificateur `static`. Le modificateur `global` peut être appliqué à une *directive alias using*. Dans les deux cas, l'étendue de la directive est tous les fichiers de la compilation actuelle. L'exemple suivant active l'utilisation de toutes les méthodes déclarées dans System.Math dans tous les fichiers de votre projet :

```
C#

global using static System.Math;
```

Vous pouvez également inclure un espace de noms de manière globale en ajoutant un `<Using>` élément à votre fichier projet, par exemple, `<Using Include="My.Awesome.Namespace" />`. Pour plus d'informations, consultez l'élément<Using>.

> **ⓘ Important**
>
> Les modèles C# pour .NET 6 utilisent des *instructions de niveau supérieur*. Votre application ne correspond peut-être pas au code de cet article, si vous avez déjà effectué une mise à niveau vers .NET 6. Pour plus d'informations, consultez l'article **Les nouveaux modèles C# génèrent des instructions de niveau supérieur**.
>
> Le SDK .NET 6 ajoute également un ensemble de directives *implicites* `global using` pour les projets qui utilisent les kits SDK suivants :
>
> - Microsoft.NET.Sdk
> - Microsoft.NET.Sdk.Web
> - Microsoft.NET.Sdk.Worker
>
> Ces directives implicites `global using` incluent les espaces de noms les plus courants pour le type de projet.

static (modificateur)

La directive `using static` nomme un type dont vous pouvez accéder aux membres statiques et aux types imbriqués sans spécifier de nom de type. Sa syntaxe est la suivante :

```
C#

using static <fully-qualified-type-name>;
```

`<fully-qualified-type-name>` est le nom du type dont les membres statiques et les types imbriqués peuvent être référencés sans spécifier de nom de type. Si vous ne fournissez pas de nom de type complet (le nom de l'espace de noms complet avec le nom du type), C# génère l'erreur du compilateur CS0246 : « Le nom du type ou de l'espace de noms 'type/espace_de_nom' est introuvable (il manque peut-être une directive using ou une référence d'assembly) ».

La directive `using static` s'applique à tout type ayant des membres statiques (ou des types imbriqués), même s'il a également des membres d'instance. Toutefois, les membres d'instance ne peuvent être appelés que par l'instance du type.

Vous pouvez accéder aux membres statiques d'un type sans devoir qualifier l'accès avec le nom du type :

C#

```
using static System.Console;
using static System.Math;
class Program
{
    static void Main()
    {
        WriteLine(Sqrt(3*3 + 4*4));
    }
}
```

En général, quand vous appelez un membre statique, vous indiquez le nom du type, ainsi que le nom du membre. Entrer plusieurs fois le même nom de type pour appeler des membres du type peut produire du code détaillé et peu clair. Par exemple, la définition suivante d'une classe `Circle` référence de nombreux membres de la classe Math.

C#

```
using System;

public class Circle
{
   public Circle(double radius)
   {
      Radius = radius;
   }

   public double Radius { get; set; }

   public double Diameter
```

```
    {
        get { return 2 * Radius; }
    }

    public double Circumference
    {
        get { return 2 * Radius * Math.PI; }
    }

    public double Area
    {
        get { return Math.PI * Math.Pow(Radius, 2); }
    }
}
```

En éliminant la nécessité de référencer explicitement la classe Math chaque fois qu'un membre est référencé, la directive `using static` génère du code plus clair :

C#

```
using System;
using static System.Math;

public class Circle
{
    public Circle(double radius)
    {
        Radius = radius;
    }

    public double Radius { get; set; }

    public double Diameter
    {
        get { return 2 * Radius; }
    }

    public double Circumference
    {
        get { return 2 * Radius * PI; }
    }

    public double Area
    {
        get { return PI * Pow(Radius, 2); }
    }
}
```

`using static` importe uniquement les membres statiques accessibles et les types imbriqués déclarés dans le type spécifié. Les membres hérités ne sont pas importés. Vous pouvez importer à partir de n'importe quel type nommé avec une directive `using`

`static`, notamment des modules Visual Basic. Si des fonctions de niveau supérieur F# apparaissent dans les métadonnées comme membres statiques d'un type nommé dont le nom est un identificateur C# valide, les fonctions F# peuvent être importées.

`using static` rend les méthodes d'extension déclarées dans le type spécifié disponibles pour la recherche de méthode d'extension. Toutefois, les noms des méthodes d'extension ne sont pas importés dans la portée pour la référence non qualifiée dans le code.

Les méthodes portant le même nom et qui sont importées à partir de différents types par différentes directives `using static` dans la même unité de compilation ou le même espace de noms forment un groupe de méthodes. La résolution de surcharge au sein de ces groupes de méthodes suit des règles C# normales.

L'exemple suivant utilise la directive `using static` pour que les membres statiques des classes Console, Math et String soient disponibles sans que vous ayez à spécifier leur nom de type.

C#

```
using System;
using static System.Console;
using static System.Math;
using static System.String;

class Program
{
   static void Main()
   {
      Write("Enter a circle's radius: ");
      var input = ReadLine();
      if (!IsNullOrEmpty(input) && double.TryParse(input, out var radius)) {
         var c = new Circle(radius);

         string s = "\nInformation about the circle:\n";
         s = s + Format("   Radius: {0:N2}\n", c.Radius);
         s = s + Format("   Diameter: {0:N2}\n", c.Diameter);
         s = s + Format("   Circumference: {0:N2}\n", c.Circumference);
         s = s + Format("   Area: {0:N2}\n", c.Area);
         WriteLine(s);
      }
      else {
         WriteLine("Invalid input...");
      }
   }
}

public class Circle
{
   public Circle(double radius)
```

```
   {
      Radius = radius;
   }

   public double Radius { get; set; }

   public double Diameter
   {
      get { return 2 * Radius; }
   }

   public double Circumference
   {
      get { return 2 * Radius * PI; }
   }

   public double Area
   {
      get { return PI * Pow(Radius, 2); }
   }
}
// The example displays the following output:
//       Enter a circle's radius: 12.45
//
//       Information about the circle:
//          Radius: 12.45
//          Diameter: 24.90
//          Circumference: 78.23
//          Area: 486.95
```

Dans cet exemple, la directive `using static` aurait également pu être appliquée au type Double. Ajouter cette directive aurait permis d'appeler la méthode TryParse(String, Double) sans spécifier un nom de type. Toutefois, utiliser `TryParse` sans nom de type crée un code moins lisible, car il est nécessaire de vérifier les directives `using static`, afin de déterminer quelle méthode du type numérique `TryParse` est appelée.

`using static` s'applique également aux types `enum`. En ajoutant `using static` avec l'énumération, le type n'est plus nécessaire pour utiliser les membres enum.

C#

```
using static Color;

enum Color
{
    Red,
    Green,
    Blue
}

class Program
```

```
{
    public static void Main()
    {
        Color color = Green;
    }
}
```

alias using

Créez une directive d'alias `using` afin de faciliter la qualification d'un identificateur pour un espace de noms ou un type. Dans une directive `using`, le type ou l'espace de noms complet doit toujours être utilisé, indépendamment des directives `using` placées avant. Aucun alias `using` ne peut être utilisé dans la déclaration d'une directive `using`. Par exemple, l'exemple suivant génère une erreur de compilation :

C#

```
using s = System.Text;
using s.RegularExpressions; // Generates a compiler error.
```

L'exemple suivant montre comment définir et utiliser un alias `using` pour un espace de noms :

C#

```
namespace PC
{
    // Define an alias for the nested namespace.
    using Project = PC.MyCompany.Project;
    class A
    {
        void M()
        {
            // Use the alias
            var mc = new Project.MyClass();
        }
    }
    namespace MyCompany
    {
        namespace Project
        {
            public class MyClass { }
        }
    }
}
```

Une directive d'alias using ne peut pas avoir un type générique ouvert sur le côté droit. Par exemple, vous ne pouvez pas créer un alias using pour un `List<T>`, mais vous pouvez en créer un pour un `List<int>`.

L'exemple suivant montre comment définir une directive `using` et un alias `using` pour une classe :

C#

```
using System;

// Using alias directive for a class.
using AliasToMyClass = NameSpace1.MyClass;

// Using alias directive for a generic class.
using UsingAlias = NameSpace2.MyClass<int>;

namespace NameSpace1
{
    public class MyClass
    {
        public override string ToString()
        {
            return "You are in NameSpace1.MyClass.";
        }
    }
}

namespace NameSpace2
{
    class MyClass<T>
    {
        public override string ToString()
        {
            return "You are in NameSpace2.MyClass.";
        }
    }
}

namespace NameSpace3
{
    class MainClass
    {
        static void Main()
        {
            var instance1 = new AliasToMyClass();
            Console.WriteLine(instance1);

            var instance2 = new UsingAlias();
            Console.WriteLine(instance2);
        }
    }
```

```
}
// Output:
//    You are in NameSpace1.MyClass.
//    You are in NameSpace2.MyClass.
```

À compter de C# 12, vous pouvez créer des alias pour les types qui ont été précédemment restreints, notamment les types de tuples, les types pointeurs et d'autres types non sécurisés. Pour plus d'informations sur les règles mises à jour, consultez les spécifications de fonctionnalité.

Comment utiliser l'espace de noms Visual Basic `My`

L'espace de noms Microsoft.VisualBasic.MyServices (`My` dans Visual Basic) permet d'accéder de manière simple et intuitive à un certain nombre de classes .NET, ce qui vous permet d'écrire du code qui interagit avec l'ordinateur, l'application, les paramètres, les ressources, etc. Bien que conçu à l'origine pour une utilisation avec Visual Basic, l'espace de noms `MyServices` peut être utilisé dans les applications C#.

Pour plus d'informations sur l'utilisation de l'espace de noms `MyServices` de Visual Basic, consultez Développement avec My.

Vous devez ajouter une référence à l'assembly *Microsoft.VisualBasic.dll* dans votre projet. Certaines classes de l'espace de noms `MyServices` ne peuvent pas être appelées à partir d'une application C# : par exemple, la classe FileSystemProxy n'est pas compatible. Dans ce cas particulier, les méthodes statiques, qui font partie intégrante de FileSystem et qui sont aussi contenues dans VisualBasic.dll, peuvent être utilisées à la place. Par exemple, voici comment utiliser une méthode de ce type pour dupliquer un répertoire :

C#

```
// Duplicate a directory
Microsoft.VisualBasic.FileIO.FileSystem.CopyDirectory(
    @"C:\original_directory",
    @"C:\copy_of_original_directory");
```

spécification du langage C#

Pour plus d'informations, consultez Directives using dans la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Pour plus d'informations sur le modificateur *global using*, consultez la spécification de fonctionnalité globale usings - C# 10.

Voir aussi

- Informations de référence sur C#
- Mots clés C#
- Espaces de noms
- Règle de style IDE0005 - Supprimer les directives « using » inutiles
- Règle de style IDE0065 : placement de directive « using »
- instruction using

instruction using : garantir l'utilisation correcte des objets supprimables

Article • 21/03/2023

L'instruction `using` garantit l'utilisation correcte d'une instance IDisposable :

C#

```
var numbers = new List<int>();
using (StreamReader reader = File.OpenText("numbers.txt"))
{
    string line;
    while ((line = reader.ReadLine()) is not null)
    {
        if (int.TryParse(line, out int number))
        {
            numbers.Add(number);
        }
    }
}
```

Quand le contrôle quitte le bloc de l'instruction `using`, une instance IDisposable acquise est supprimée. En particulier, l'instruction `using` garantit qu'une instance supprimable est supprimée même si une exception se produit dans le bloc de l'instruction `using`. Dans l'exemple précédent, un fichier ouvert est fermé après le traitement de toutes les lignes.

Utilisez l'instruction `await using` pour utiliser correctement une instance IAsyncDisposable :

C#

```
await using (var resource = new AsyncDisposableExample())
{
    // Use the resource
}
```

Pour plus d'informations sur l'utilisation d'instances IAsyncDisposable, consultez la section Utilisation de la suppression asynchrone de l'article Implémenter une méthode DisposeAsync.

Vous pouvez aussi utiliser une `using` *déclaration* qui ne nécessite pas d'accolades :

C#

```
static IEnumerable<int> LoadNumbers(string filePath)
{
    using StreamReader reader = File.OpenText(filePath);

    var numbers = new List<int>();
    string line;
    while ((line = reader.ReadLine()) is not null)
    {
        if (int.TryParse(line, out int number))
        {
            numbers.Add(number);
        }
    }
    return numbers;
}
```

Quand elle est déclarée dans une déclaration `using`, une variable locale est supprimée à la fin de l'étendue dans laquelle elle est déclarée. Dans l'exemple précédent, la suppression se produit à la fin d'une méthode.

Une variable déclarée par l'instruction ou la déclaration `using` est en lecture seule. Vous ne pouvez pas la réaffecter ou la passer en tant que paramètre ref ou out.

Vous pouvez déclarer plusieurs instances du même type dans une instruction `using`, comme le montre l'exemple suivant :

C#

```
using (StreamReader numbersFile = File.OpenText("numbers.txt"), wordsFile =
File.OpenText("words.txt"))
{
    // Process both files
}
```

Quand vous déclarez plusieurs instances dans une instruction `using`, elles sont supprimées dans l'ordre inverse de déclaration.

Vous pouvez aussi utiliser l'instruction et la `using` déclaration avec une instance d'un struct ref qui correspond au modèle supprimable. Autrement dit, il doit avoir une méthode `Dispose` d'instance qui soit accessible et sans paramètre, avec un type de retour `void`.

L'instruction `using` peut également avoir la forme suivante :

C#

```
using (expression)
{
    // ...
}
```

où `expression` produit une instance supprimable. L'exemple suivant illustre ce qui suit :

C#

```
StreamReader reader = File.OpenText(filePath);

using (reader)
{
    // Process file content
}
```

> ⚠ **Avertissement**
>
> Dans l'exemple précédent, une fois que le contrôle a quitté l'instruction `using`, une instance supprimable reste dans l'étendue alors qu'elle est déjà supprimée. Si vous utilisez cette instance plus loin, vous pouvez rencontrer une exception, par exemple **ObjectDisposedException**. C'est pourquoi nous vous recommandons de déclarer une variable supprimable dans l'instruction `using` ou avec la déclaration `using`.

spécification du langage C#

Pour plus d'informations, consultez la section L'instruction using de la spécification du langage C# et la note de proposition relative à « using basé sur un modèle » et « déclarations using ».

Voir aussi

- Informations de référence sur C#
- System.IDisposable
- System.IAsyncDisposable
- Utilisation d'objets implémentant IDisposable
- Implémenter une méthode Dispose
- Implémenter une méthode DisposeAsync
- Utiliser l'instruction « using » simple (règle de style IDE0063)
- Directive using

extern alias (référence C#)

Article • 07/04/2023

Il se peut que vous deviez référencer deux versions d'assemblys qui ont le même nom de type qualifié complet. Par exemple, vous pourriez avoir à utiliser plusieurs versions d'un assembly dans la même application. En utilisant un alias d'assembly externe, vous pouvez encapsuler les espaces de noms de chaque assembly dans des espaces de noms racines nommés par l'alias, ce qui permet de les utiliser dans le même fichier.

> ⓘ **Notes**
>
> Le mot clé **extern** est également utilisé comme modificateur de méthode, déclarant une méthode écrite en code non managé.

Pour référencer deux assemblys portant le même nom de type qualifié complet, vous devez spécifier un alias à l'invite de commandes, comme suit :

```
/r:GridV1=grid.dll
```

```
/r:GridV2=grid20.dll
```

Cela crée les alias externes `GridV1` et `GridV2`. Pour utiliser ces alias à partir d'un programme, référencez-les en utilisant le mot clé `extern`. Par exemple :

```
extern alias GridV1;
```

```
extern alias GridV2;
```

Chaque déclaration d'alias externe introduit un espace de noms racine supplémentaire qui est parallèle à l'espace de noms global (mais n'est pas compris dedans). Ainsi, vous pouvez référencer les types de chaque assembly sans ambiguïté en utilisant leur nom qualifié complet, enraciné dans l'alias d'espace de noms approprié.

Dans l'exemple précédent, `GridV1::Grid` serait le contrôle de grille de `grid.dll`, et `GridV2::Grid` serait le contrôle de grille de `grid20.dll`.

Utilisation de Visual Studio

Si vous utilisez Visual Studio, les alias peuvent être fournis de manière similaire.

Ajoutez une référence de *grid.dll* et *grid20.dll* à votre projet dans Visual Studio. Ouvrez un onglet de propriété et modifiez les alias de global en GridV1 et GridV2 respectivement.

Utilisez ces alias de la même façon ci-dessus

```
C#

 extern alias GridV1;

 extern alias GridV2;
```

Vous pouvez maintenant créer un alias pour un espace de noms ou un type avec une *directive using alias*. Pour plus d'informations, consultez Directive using.

```
C#

using Class1V1 = GridV1::Namespace.Class1;

using Class1V2 = GridV2::Namespace.Class1;
```

Spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- :: (opérateur)
- **References** (Options du compilateur C#)

new, contrainte (référence C#)

Article • 07/04/2023

La contrainte `new` spécifie que tout argument de type dans une déclaration de classe ou de méthode générique doit avoir un constructeur sans paramètre public. Pour utiliser la contrainte `new`, le type ne doit pas être abstract.

Appliquez la contrainte `new` à un paramètre de type quand une classe générique crée des instances du type, comme illustré dans l'exemple suivant :

C#

```
class ItemFactory<T> where T : new()
{
    public T GetNewItem()
    {
        return new T();
    }
}
```

Si vous utilisez la contrainte `new()` avec d'autres contraintes, spécifiez-la en dernier :

C#

```
public class ItemFactory2<T>
    where T : IComparable, new()
{  }
```

Pour plus d'informations, consultez Contraintes sur les paramètres de type.

Vous pouvez également utiliser le mot clé `new` pour créer une instance d'un type ou l'utiliser comme un modificateur de déclaration de membre.

spécification du langage C#

Pour plus d'informations, consultez la section Contraintes de paramètre de type de la spécification du langage C#.

Voir aussi

- Référence C#
- Guide de programmation C#

- Mots clés C#
- Génériques

where (contrainte de type générique) (Référence C#)

Article • 07/04/2023

La clause `where` dans une définition générique spécifie des contraintes sur les types qui sont utilisés comme arguments pour les paramètres de type d'un type générique, d'une méthode, d'un délégué ou d'une fonction locale. Les contraintes peuvent spécifier des interfaces ou des classes de base, ou nécessiter un type générique comme référence, valeur ou type non managé. Elles déclarent des fonctionnalités que l'argument de type doit avoir, et elles doivent être placées après toute classe de base déclarée ou interfaces implémentées.

Vous pouvez, par exemple, déclarer une classe générique, `AGenericClass`, de telle sorte que le paramètre de type `T` implémente l'interface IComparable<T> :

C#

```
public class AGenericClass<T> where T : IComparable<T> { }
```

> ⓘ **Notes**
>
> Pour plus d'informations sur la clause where dans une expression de requête, consultez **where, clause**.

La clause `where` peut également inclure une contrainte de classe de base. La contrainte de classe de base indique qu'un type à utiliser comme argument de type pour ce type générique a la classe spécifiée comme classe de base ou est la classe de base. Si la contrainte de classe de base est utilisée, elle doit apparaître avant toute autre contrainte sur ce paramètre de type. Certains types ne sont pas autorisés comme contrainte de classe de base : Object, Array et ValueType. L'exemple suivant montre les types qui peuvent maintenant être spécifiés comme classe de base :

C#

```
public class UsingEnum<T> where T : System.Enum { }

public class UsingDelegate<T> where T : System.Delegate { }

public class Multicaster<T> where T : System.MulticastDelegate { }
```

Dans un contexte nullable, la possibilité de nullabilité du type de classe de base est appliquée. Si la classe de base est non-nullable (par exemple `Base`), l'argument de type doit être non-nullable. Si la classe de base est nullable (par exemple `Base?`), l'argument de type peut être un type référence nullable ou non-nullable. Le compilateur émet un avertissement si l'argument de type est un type référence nullable lorsque la classe de base est non-nullable.

La clause `where` peut spécifier que le type est une `class` ou un `struct`. La contrainte `struct` supprime la nécessité de spécifier une contrainte de classe de base de `System.ValueType`. Le type `System.ValueType` ne doit pas être utilisé comme contrainte de classe de base. L'exemple suivant montre les contraintes `class` et `struct` :

C#

```
class MyClass<T, U>
    where T : class
    where U : struct
{ }
```

Dans un contexte nullable, la contrainte `class` exige qu'un type soit un type référence non-nullable. Pour autoriser les types référence nullables, utilisez la contrainte `class?`, qui autorise les types référence nullables et non-nullables.

La clause `where` peut inclure la contrainte `notnull`. La contrainte `notnull` limite le paramètre de type aux types non-nullables. Le type peut être un type valeur ou un type référence non-nullable. La contrainte `notnull` est disponible pour le code compilé dans un nullable enable contexte. Contrairement à d'autres contraintes, si un argument de type enfreint la contrainte `notnull`, le compilateur génère un avertissement au lieu d'une erreur. Les avertissements sont générés uniquement dans un contexte `nullable enable`.

L'ajout de types référence nullables introduit une ambiguïté potentielle dans la signification de `T?` dans les méthodes génériques. Si `T` est un `struct`, `T?` est identique à System.Nullable<T>. Toutefois, si `T` est un type référence, `T?` signifie que `null` est une valeur valide. L'ambiguïté survient parce que les méthodes de substitution ne peuvent pas inclure de contraintes. La nouvelle contrainte `default` résout cette ambiguïté. Vous l'ajouterez lorsqu'une classe de base ou une interface déclare deux surcharges d'une méthode, l'une qui spécifie la contrainte `struct` et l'autre qui n'a pas la contrainte `struct` ou `class` appliquée :

C#

```
public abstract class B
{
    public void M<T>(T? item) where T : struct { }
    public abstract void M<T>(T? item);

}
```

Vous utilisez la contrainte `default` pour spécifier que votre classe dérivée remplace la méthode sans la contrainte dans votre classe dérivée ou l'implémentation d'interface explicite. Elle n'est valide que sur les méthodes qui remplacent les méthodes de base ou les implémentations d'interface explicites :

C#

```
public class D : B
{
    // Without the "default" constraint, the compiler tries to override the
first method in B
    public override void M<T>(T? item) where T : default { }
}
```

> **ⓘ Important**
>
> Les déclarations génériques qui incluent la contrainte `notnull` peuvent être utilisées dans un contexte inconscient nullable, mais le compilateur n'applique pas la contrainte.

C#

```
#nullable enable
    class NotNullContainer<T>
        where T : notnull
    {
    }
#nullable restore
```

La clause `where` peut aussi inclure une contrainte `unmanaged`. La contrainte `unmanaged` limite le paramètre de type aux types connus sous le nom de types non managés. La contrainte `unmanaged` facilite l'écriture de code interop de bas niveau en C#. Cette contrainte permet des routines réutilisables sur tous les types non managés. La contrainte `unmanaged` ne peut pas être combinée avec la contrainte `class` ou `struct`. La contrainte `unmanaged` exige que le type doit être un `struct` :

C#

```
class UnManagedWrapper<T>
    where T : unmanaged
{ }
```

La clause `where` peut également inclure une contrainte de constructeur, `new()`. Cette contrainte permet de créer une instance d'un paramètre de type à l'aide de l'opérateur `new`. La contrainte new() indique au compilateur que tout argument de type fourni doit avoir un constructeur accessible sans paramètre. Par exemple :

C#

```
public class MyGenericClass<T> where T : IComparable<T>, new()
{
    // The following line is not possible without new() constraint:
    T item = new T();
}
```

La contrainte `new()` apparaît en dernier dans la clause `where`. La contrainte `new()` ne peut pas être combinée avec les contraintes `struct` ou `unmanaged`. Tous les types répondant à ces contraintes doivent avoir un constructeur sans paramètre accessible, ce qui rend la contrainte `new()` redondante.

Avec plusieurs paramètres de type, utilisez une clause `where` pour chaque paramètre de type, par exemple :

C#

```
public interface IMyInterface { }

namespace CodeExample
{
    class Dictionary<TKey, TVal>
        where TKey : IComparable<TKey>
        where TVal : IMyInterface
    {
        public void Add(TKey key, TVal val) { }
    }
}
```

Vous pouvez également joindre des contraintes aux paramètres de type des méthodes génériques, comme montré dans l'exemple suivant :

C#

```
public void MyMethod<T>(T t) where T : IMyInterface { }
```

Notez que la syntaxe décrivant les contraintes de paramètre de type sur les délégués est la même que celle des méthodes :

```
C#
```

```
delegate T MyDelegate<T>() where T : new();
```

Pour plus d'informations sur les délégués génériques, consultez Délégués génériques.

Pour plus d'informations sur la syntaxe et l'utilisation de contraintes, consultez Contraintes sur les paramètres de type.

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Introduction aux génériques
- new, contrainte
- Contraintes sur les paramètres de type

base (référence C#)

Article • 23/03/2023

Le mot clé `base` sert à accéder aux membres de la classe de base à partir d'une classe dérivée. Utilisez-le si vous souhaitez :

- Il appelle une méthode de la classe de base qui a été substituée par une autre méthode.
- Il spécifie quel constructeur de classe de base doit être appelé lors de la création d'instances de la classe dérivée.

L'accès à une classe de base est autorisé uniquement dans un constructeur, une méthode d'instance et un accesseur de propriété d'instance.

L'utilisation du mot clé `base` à partir d'une méthode statique entraîne une erreur.

La classe de base accessible est la classe de base spécifiée dans la déclaration de classe. Par exemple, si vous spécifiez `class ClassB : ClassA`, les membres de ClassA sont accessibles à partir de ClassB, quelle que soit la classe de base de ClassA.

Exemple 1

Dans cet exemple, la classe de base `Person` et la classe dérivée `Employee` ont toutes les deux une méthode nommée `GetInfo`. En utilisant le mot clé `base`, vous pouvez appeler la méthode `GetInfo` de la classe de base à partir de la classe dérivée.

C#

```
public class Person
{
    protected string ssn = "444-55-6666";
    protected string name = "John L. Malgraine";

    public virtual void GetInfo()
    {
        Console.WriteLine("Name: {0}", name);
        Console.WriteLine("SSN: {0}", ssn);
    }
}
class Employee : Person
{
    public string id = "ABC567EFG";
    public override void GetInfo()
    {
```

```
        // Calling the base class GetInfo method:
        base.GetInfo();
        Console.WriteLine("Employee ID: {0}", id);
    }
}

class TestClass
{
    static void Main()
    {
        Employee E = new Employee();
        E.GetInfo();
    }
}
/*
Output
Name: John L. Malgraine
SSN: 444-55-6666
Employee ID: ABC567EFG
*/
```

Pour obtenir d'autres exemples, consultez new, virtual et override.

Exemple 2

Cet exemple montre comment spécifier le constructeur de classe de base qui est appelé lors de la création d'instances d'une classe dérivée.

C#

```
public class BaseClass
{
    int num;

    public BaseClass()
    {
        Console.WriteLine("in BaseClass()");
    }

    public BaseClass(int i)
    {
        num = i;
        Console.WriteLine("in BaseClass(int i)");
    }

    public int GetNum()
    {
        return num;
    }
}
```

```
public class DerivedClass : BaseClass
{
    // This constructor will call BaseClass.BaseClass()
    public DerivedClass() : base()
    {
    }

    // This constructor will call BaseClass.BaseClass(int i)
    public DerivedClass(int i) : base(i)
    {
    }

    static void Main()
    {
        DerivedClass md = new DerivedClass();
        DerivedClass md1 = new DerivedClass(1);
    }
}
/*
Output:
in BaseClass()
in BaseClass(int i)
*/
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- this

this (référence C#)

Article • 07/04/2023

Le mot clé `this` fait référence à l'instance actuelle de la classe et est également utilisé comme modificateur du premier paramètre d'une méthode d'extension.

> **Notes**
>
> Cet article traite de l'utilisation de `this` avec des instances de classe. Pour plus d'informations sur son utilisation dans les méthodes d'extension, consultez **Méthodes d'extension**.

Voici quelques utilisations courantes de `this` :

- Pour qualifier des membres masqués par des noms similaires, par exemple :

```
C#

public class Employee
{
    private string alias;
    private string name;

    public Employee(string name, string alias)
    {
        // Use this to qualify the members of the class
        // instead of the constructor parameters.
        this.name = name;
        this.alias = alias;
    }
}
```

- Pour passer un objet comme paramètre à d'autres méthodes, par exemple :

```
C#

CalcTax(this);
```

- Pour déclarer des indexeurs, par exemple :

```
C#

public int this[int param]
{
    get { return array[param]; }
```

```
        set { array[param] = value; }
    }
```

Les fonctions membres statiques, car ils existent au niveau de la classe et non comme faisant partie d'un objet, n'ont pas de pointeur `this`. Une erreur consiste à faire référence à `this` dans une méthode statique.

Exemple

Dans cet exemple, `this` est utilisé pour qualifier les membres de la classe `Employee`, `name` et `alias`, qui sont masqués par des noms similaires. Il est également utilisé pour passer un objet à la méthode `CalcTax`, qui appartient à une autre classe.

C#

```
class Employee
{
    private string name;
    private string alias;
    private decimal salary = 3000.00m;

    // Constructor:
    public Employee(string name, string alias)
    {
        // Use this to qualify the fields, name and alias:
        this.name = name;
        this.alias = alias;
    }

    // Printing method:
    public void printEmployee()
    {
        Console.WriteLine("Name: {0}\nAlias: {1}", name, alias);
        // Passing the object to the CalcTax method by using this:
        Console.WriteLine("Taxes: {0:C}", Tax.CalcTax(this));
    }

    public decimal Salary
    {
        get { return salary; }
    }
}

class Tax
{
    public static decimal CalcTax(Employee E)
    {
        return 0.08m * E.Salary;
    }
```

```
}

class MainClass
{
    static void Main()
    {
        // Create objects:
        Employee E1 = new Employee("Mingda Pan", "mpan");

        // Display results:
        E1.printEmployee();
    }
}
/*
Output:
    Name: Mingda Pan
    Alias: mpan
    Taxes: $240.00
 */
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- this Préférences de style de code (IDE0003 et IDE0009)
- Référence C#
- Guide de programmation C#
- Mots clés C#
- base
- Méthodes

null (référence C#)

Article • 07/04/2023

Le mot clé `null` est un littéral qui représente une référence null, c'est-à-dire qui ne fait pas référence à un objet. `null` est la valeur par défaut des variables de type référence. Les types valeur ordinaires ne peuvent pas être Null, à l'exception des types valeur Nullable.

L'exemple suivant illustre certains comportements du mot clé `null` :

C#

```
class Program
{
    class MyClass
    {
        public void MyMethod() { }
    }

    static void Main(string[] args)
    {
        // Set a breakpoint here to see that mc = null.
        // However, the compiler considers it "unassigned."
        // and generates a compiler error if you try to
        // use the variable.
        MyClass mc;

        // Now the variable can be used, but...
        mc = null;

        // ... a method call on a null object raises
        // a run-time NullReferenceException.
        // Uncomment the following line to see for yourself.
        // mc.MyMethod();

        // Now mc has a value.
        mc = new MyClass();

        // You can call its method.
        mc.MyMethod();

        // Set mc to null again. The object it referenced
        // is no longer accessible and can now be garbage-collected.
        mc = null;

        // A null string is not the same as an empty string.
        string s = null;
        string t = String.Empty; // Logically the same as ""
```

```
        // Equals applied to any null object returns false.
        bool b = (t.Equals(s));
        Console.WriteLine(b);

        // Equality operator also returns false when one
        // operand is null.
        Console.WriteLine("Empty string {0} null string", s == t ? "equals":
"does not equal");

        // Returns true.
        Console.WriteLine("null == null is {0}", null == null);

        // A value type cannot be null
        // int i = null; // Compiler error!

        // Use a nullable value type instead:
        int? i = null;

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Informations de référence sur C#
- Mots clés C#
- Valeurs par défaut des types C#
- Nothing (Visual Basic)

bool (référence C#)

Article • 10/05/2023

Le mot clé de type `bool` est un alias pour le type de structure .NET System.Boolean qui représente une valeur booléenne, qui peut être `true` ou `false`.

Pour effectuer des opérations logiques avec des valeurs du type `bool`, utilisez des opérateurs logiques booléens. Le type `bool` est le type de résultat des opérateurs de comparaison et d'égalité. Une expression `bool` défini peut être une expression conditionnelle de contrôle dans les instructions if, do, while et for et dans l'opérateur conditionnel ?:.

La valeur par défaut du type `bool` est `false`.

Littéraux

Vous pouvez utiliser les littéraux `true` et `false` pour initialiser une variable `bool` ou transmettre une valeur `bool` :

C#

```
bool check = true;
Console.WriteLine(check ? "Checked" : "Not checked");  // output: Checked

Console.WriteLine(false ? "Checked" : "Not checked");  // output: Not
checked
```

Logique booléenne à trois valeurs

Utilisez le type Nullable `bool?` si vous voulez suivre une logique à trois valeurs, par exemple pour travailler avec des bases de données qui acceptent un type booléen à trois valeurs. Dans le cas des opérandes `bool?`, les opérateurs prédéfinis `&` et `|` prennent en charge la logique à trois valeurs. Pour plus d'informations, voir la section Opérateurs logiques booléens Nullable de l'article Opérateurs logiques booléens.

Pour plus d'informations sur les types de valeur pouvant accepter la valeur Null, consultez Types de valeur pouvant accepter la valeur Null.

Conversions

C# fournit uniquement deux conversions qui impliquent le type `bool`. Il s'agit d'une conversion implicite vers le type Nullable `bool?` correspondant et d'une conversion explicite à partir du type `bool?`. Toutefois, .NET fournit des méthodes supplémentaires que vous pouvez utiliser pour effectuer une conversion vers ou depuis le type `bool`. Pour plus d'informations, consultez la section Convertir en et à partir de valeurs booléennes de la page de référence de l'API System.Boolean.

spécification du langage C#

Pour plus d'informations, consultez la section Type bool de la spécification du langage C#.

Voir aussi

- Informations de référence sur C#
- Types de valeur
- opérateurs true et false

default (référence C#)

Article • 07/04/2023

Vous pouvez utiliser le mot clé `default` dans les contextes suivants :

- Pour spécifier la casse par défaut dans l'instruction switch.
- Comme opérateur ou littéral par défaut pour produire la valeur par défaut d'un type.
- Comme contrainte de type default sur une implémentation d'interface explicite ou de substitution de méthode générique.

Voir aussi

- Informations de référence sur C#
- Mots clés C#

add (Référence C#)

Article • 07/04/2023

Le mot clé contextuel `add` est utilisé pour définir un accesseur d'événement personnalisé qui est appelé quand le code client s'abonne à votre événement (event). Si vous fournissez un accesseur `add` personnalisé, vous devez également fournir un accesseur remove.

Exemple

L'exemple suivant illustre un événement qui a des accesseurs `add` et remove personnalisés. Pour obtenir un exemple complet, consultez Guide pratique pour implémenter des événements d'interface.

C#

```
class Events : IDrawingObject
{
    event EventHandler PreDrawEvent;

    event EventHandler IDrawingObject.OnDraw
    {
        add => PreDrawEvent += value;
        remove => PreDrawEvent -= value;
    }
}
```

En général, vous n'avez pas besoin de fournir vos propres accesseurs d'événements personnalisés. Les accesseurs générés automatiquement par le compilateur quand vous déclarez un événement sont suffisants pour la plupart des scénarios.

Voir aussi

- Événements

get (référence C#)

Article • 07/04/2023

Le mot clé `get` définit une méthode *Accessor* dans une propriété ou un indexeur qui retourne la valeur de la propriété ou l'élément de l'indexeur. Pour plus d'informations, consultez Propriétés, Propriétés implémentées automatiquement et Indexeurs.

L'exemple suivant définit un accesseur `get` et un accesseur `set` pour une propriété nommée `Seconds`. Il utilise un champ privé nommé `_seconds` pour stocker la valeur de la propriété.

C#

```
class TimePeriod
{
     private double _seconds;

     public double Seconds
     {
         get { return _seconds; }
         set { _seconds = value; }
     }
}
```

Souvent, l'accesseur `get` se compose d'une seule instruction qui retourne une valeur, comme dans l'exemple précédent. Vous pouvez implémenter l'accesseur `get` comme membre expression-bodied. L'exemple suivant implémente l'accesseur `get` et l'accesseur `set` en tant que membres expression-bodied.

C#

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get => _seconds;
        set => _seconds = value;
    }
}
```

Pour les cas simples dans lesquels les accesseurs `get` et `set` de la propriété n'effectuent aucune autre opération que la définition ou la récupération d'une valeur dans un champ de stockage privé, vous pouvez tirer parti de la prise en charge des propriétés implémentées automatiquement fournies par le compilateur C#. L'exemple suivant implémente `Hours` en tant que propriété implémentée automatiquement.

C#

```
class TimePeriod2
{
    public double Hours { get; set; }
}
```

Spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Propriétés

init (référence C#)

Article • 27/06/2023

Pour C# 9 et versions ultérieures, le mot clé `init` définit une méthode d'*accesseur* dans une propriété ou un indexeur. Un setter init-only affecte une valeur à la propriété ou à l'élément d'indexeur **uniquement** pendant la construction de l'objet. Ce processus impose l'immuabilité, de sorte qu'une fois l'objet initialisé, il ne peut plus être modifié.

Pour plus d'informations et des exemples, consultez Propriétés, Propriétés implémentées automatiquement et Indexeurs.

L'exemple suivant définit un accesseur `get` et un accesseur `init` pour une propriété nommée `YearOfBirth`. Il utilise un champ privé nommé `_yearOfBirth` pour stocker la valeur de la propriété.

C#

```
class Person_InitExample
{
     private int _yearOfBirth;

     public int YearOfBirth
     {
         get { return _yearOfBirth; }
         init { _yearOfBirth = value; }
     }
}
```

Souvent, l'accesseur `init` se compose d'une seule instruction qui assigne une valeur, comme dans l'exemple précédent. En raison d'`init`, notez que le code suivant **ne fonctionne pas** :

C#

```
var john = new Person_InitExample
{
    YearOfBirth = 1984
};

john.YearOfBirth = 1926; //Not allowed, as its value can only be set once in
the constructor
```

Un accesseur `init` ne force pas les appelants à définir la propriété. Au lieu de cela, il permet à un initialiseur d'objet de définir la valeur initiale tout en interdisant les

modifications ultérieures. Vous pouvez ajouter le modificateur `required` pour forcer les appelants à définir une propriété. L'exemple suivant montre le même comportement :

C#

```
class Person_InitExampleNullability
{
    private int? _yearOfBirth;

    public int? YearOfBirth
    {
        get => _yearOfBirth;
        init => _yearOfBirth = value;
    }
}
```

L'accesseur `init` peut être utilisé comme membre expression-bodied. Exemple :

C#

```
class Person_InitExampleExpressionBodied
{
    private int _yearOfBirth;

    public int YearOfBirth
    {
        get => _yearOfBirth;
        init => _yearOfBirth = value;
    }
}
```

L'accesseur `init` peut également être utilisé dans les propriétés implémentées automatiquement, comme le montre l'exemple de code suivant :

C#

```
class Person_InitExampleAutoProperty
{
    public int YearOfBirth { get; init; }
}
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Propriétés

partial, type (référence C#)

Article • 07/04/2023

Les définitions de type partiel permettent le fractionnement de la définition d'une classe, d'un struct, d'une interface ou d'un enregistrement entre plusieurs fichiers.

Dans *File1.cs* :

C#

```
namespace PC
{
    partial class A
    {
        int num = 0;
        void MethodA() { }
        partial void MethodC();
    }
}
```

Déclaration dans *File2.cs* :

C#

```
namespace PC
{
    partial class A
    {
        void MethodB() { }
        partial void MethodC() { }
    }
}
```

Remarques

Fractionner un type de classe, de struct ou d'interface entre plusieurs fichiers peut être utile si vous travaillez sur des projets volumineux ou des projets contenant du code généré automatiquement, comme celui fourni par le Concepteur Windows Forms. Un type partiel peut contenir une méthode partielle. Pour plus d'informations, consultez la page Classes et méthodes partielles.

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Modificateurs
- Introduction aux génériques

partial, méthode (référence C#)

Article • 10/03/2023

La signature d'une méthode partielle est définie dans une partie d'un type partiel, tandis que son implémentation est définie dans une autre partie du type. Les méthodes partielles permettent aux concepteurs de classes de fournir des hooks de méthode, semblables aux gestionnaires d'événements, que les développeurs peuvent décider ou non d'implémenter. Si le développeur ne fournit pas d'implémentation, le compilateur supprime la signature au moment de la compilation. Les méthodes partielles obéissent aux conditions suivantes :

- Les déclarations doivent commencer par le mot clé contextuel partial.
- Les signatures des deux parties du type partiel doivent correspondre.

Le mot clé `partial` n'est pas autorisé sur les constructeurs, les finaliseurs, les opérateurs surchargés, les déclarations de propriété ou les déclarations d'événement.

Une méthode partielle n'est pas nécessaire pour avoir une implémentation dans les cas suivants :

- Elle n'a aucun modificateur d'accessibilité (y compris le private par défaut).
- Elle retourne la valeur void.
- Elle n'a pas de paramètres out.
- Elle n'a aucun des modificateurs suivants : virtual, override, sealed, new ou extern.

Les méthodes qui ne sont pas conformes à toutes ces restrictions (par exemple, la méthode `public virtual partial void`) doivent fournir une implémentation.

L'exemple suivant illustre une méthode partielle définie dans deux parties d'une classe partielle :

C#

```
namespace PM
{
    partial class A
    {
        partial void OnSomethingHappened(string s);
    }

    // This part can be in a separate file.
    partial class A
```

```
    {
        // Comment out this method and the program
        // will still compile.
        partial void OnSomethingHappened(String s)
        {
            Console.WriteLine("Something happened: {0}", s);
        }
    }
}
```

Les méthodes partielles peuvent également être utiles combinées avec des générateurs de source. Par exemple, une expression régulière peut être définie à l'aide du modèle suivant :

C#

```
[GeneratedRegex("(dog|cat|fish)")]
partial bool IsPetMatch(string input);
```

Pour plus d'informations, consultez la page Classes et méthodes partielles.

Voir aussi

- Référence C#
- partial, type

remove (Référence C#)

Article • 07/04/2023

Le mot clé contextuel `remove` est utilisé pour définir un accesseur d'événement personnalisé qui est appelé quand le code client annule son abonnement à votre événement (event). Si vous fournissez un accesseur `remove` personnalisé, vous devez également fournir un accesseur add.

Exemple

L'exemple suivant illustre un événement qui a des accesseurs add et `remove` personnalisés. Pour obtenir un exemple complet, consultez Guide pratique pour implémenter des événements d'interface.

C#

```
class Events : IDrawingObject
{
    event EventHandler PreDrawEvent;

    event EventHandler IDrawingObject.OnDraw
    {
        add => PreDrawEvent += value;
        remove => PreDrawEvent -= value;
    }
}
```

En général, vous n'avez pas besoin de fournir vos propres accesseurs d'événements personnalisés. Les accesseurs générés automatiquement par le compilateur quand vous déclarez un événement sont suffisants pour la plupart des scénarios.

Voir aussi

- Événements

set (référence C#)

Article • 07/04/2023

Le mot clé `set` définit une méthode *accessor* dans une propriété ou un indexeur qui assigne une valeur à l'élément de la propriété ou de l'indexeur. Pour plus d'informations et des exemples, consultez Propriétés, Propriétés implémentées automatiquement et Indexeurs.

L'exemple suivant définit un accesseur `get` et un accesseur `set` pour une propriété nommée `Seconds`. Il utilise un champ privé nommé `_seconds` pour stocker la valeur de la propriété.

C#

```
class TimePeriod
{
      private double _seconds;

      public double Seconds
      {
          get { return _seconds; }
          set { _seconds = value; }
      }
}
```

Souvent, l'accesseur `set` se compose d'une seule instruction qui assigne une valeur, comme dans l'exemple précédent. Vous pouvez implémenter l'accesseur `set` comme membre expression-bodied. L'exemple suivant implémente l'accesseur `get` et l'accesseur `set` en tant que membres expression-bodied.

C#

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get => _seconds;
        set => _seconds = value;
    }
}
```

Pour les cas simples dans lesquels les accesseurs `get` et `set` de la propriété n'effectuent aucune autre opération que la définition ou la récupération d'une valeur dans un champ de stockage privé, vous pouvez tirer parti de la prise en charge des propriétés implémentées automatiquement fournies par le compilateur C#. L'exemple suivant implémente `Hours` en tant que propriété implémentée automatiquement.

C#

```
class TimePeriod2
{
    public double Hours { get; set; }
}
```

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#
- Propriétés

when (Référence C#)

Article • 07/04/2023

Vous pouvez utiliser le mot clé contextuel `when` pour spécifier une condition de filtre dans les contextes suivant :

- dans l'instruction `catch` d'un bloc try/catch ou try/catch/finally ;
- En tant que gardien de cas dans l'instructionswitch.
- En tant que gardien de cas dans l'expression switch.

`when` dans une instruction `catch`

Le mot clé `when` peut être utilisé dans une instruction `catch` pour spécifier une condition qui doit avoir la valeur true sur le gestionnaire pour qu'une exception spécifique s'exécute. Sa syntaxe est la suivante :

C#

```
catch (ExceptionType [e]) when (expr)
```

où *expr* est une expression qui prend une valeur booléenne. Si la valeur retournée est `true`, le gestionnaire de l'exception est exécuté ; si la valeur retournée est `false`, il n'est pas exécuté.

L'exemple suivant utilise le mot clé `when` afin d'exécuter conditionnellement des gestionnaires pour une exception HttpRequestException selon le texte du message d'exception.

C#

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

class Program
{
    static void Main()
    {
        Console.WriteLine(MakeRequest().Result);
    }

    public static async Task<string> MakeRequest()
    {
        var client = new HttpClient();
```

```
        var streamTask = client.GetStringAsync("https://localHost:10000");
        try
        {
            var responseText = await streamTask;
            return responseText;
        }
        catch (HttpRequestException e) when (e.Message.Contains("301"))
        {
            return "Site Moved";
        }
        catch (HttpRequestException e) when (e.Message.Contains("404"))
        {
            return "Page Not Found";
        }
        catch (HttpRequestException e)
        {
            return e.Message;
        }
    }
}
```

Voir aussi

- try/catch, instruction
- try/catch/finally, instruction

value (référence C#)

Article • 07/04/2023

Le mot clé contextuel `value` est utilisé dans l'accesseur dans les déclarations de setpropriété et d'indexeur. Il est semblable à un paramètre d'entrée sur une méthode. Le mot `value` fait référence à la valeur que le code client tente d'assigner à la propriété ou à l'indexeur. Dans l'exemple suivant, `MyDerivedClass` a une propriété appelée `Name` qui utilise le paramètre `value` pour assigner une nouvelle chaîne au `name` du champ de stockage. Du point de vue du code client, l'opération est écrite comme une assignation simple.

C#

```
class MyBaseClass
{
    // virtual auto-implemented property. Overrides can only
    // provide specialized behavior if they implement get and set accessors.
    public virtual string Name { get; set; }

    // ordinary virtual property with backing field
    private int _num;
    public virtual int Number
    {
        get { return _num; }
        set { _num = value; }
    }
}

class MyDerivedClass : MyBaseClass
{
    private string _name;

    // Override auto-implemented property with ordinary property
    // to provide specialized accessor behavior.
    public override string Name
    {
        get
        {
            return _name;
        }
        set
        {
            if (!string.IsNullOrEmpty(value))
            {
                _name = value;
            }
            else
            {
                _name = "Unknown";
```

```
            }
        }
    }
}
```

Pour plus d'informations, consultez les articles Propriétés et Indexeurs.

spécification du langage C#

Pour plus d'informations, voir la spécification du langage C#. La spécification du langage est la source de référence pour la syntaxe C# et son utilisation.

Voir aussi

- Référence C#
- Guide de programmation C#
- Mots clés C#

yield, instruction - fournir l'élément suivant

Article • 03/06/2023

Vous utilisez l'instruction `yield` dans un itérateur pour fournir la valeur suivante ou signaler la fin d'une itération. L'instruction `yield` a les deux formes suivantes :

- `yield return` : pour fournir la valeur suivante dans l'itération, comme le montre l'exemple suivant :

  C#

  ```
  foreach (int i in ProduceEvenNumbers(9))
  {
      Console.Write(i);
      Console.Write(" ");
  }
  // Output: 0 2 4 6 8

  IEnumerable<int> ProduceEvenNumbers(int upto)
  {
      for (int i = 0; i <= upto; i += 2)
      {
          yield return i;
      }
  }
  ```

- `yield break` : pour signaler explicitement la fin de l'itération, comme le montre l'exemple suivant :

  C#

  ```
  Console.WriteLine(string.Join(" ", TakeWhilePositive(new[] { 2, 3, 4,
  5, -1, 3, 4})));
  // Output: 2 3 4 5

  Console.WriteLine(string.Join(" ", TakeWhilePositive(new[] { 9, 8, 7
  })));
  // Output: 9 8 7

  IEnumerable<int> TakeWhilePositive(IEnumerable<int> numbers)
  {
      foreach (int n in numbers)
      {
          if (n > 0)
          {
              yield return n;
  ```

```
        }
        else
        {
            yield break;
        }
    }
}
```

L'itération se termine aussi une fois que le contrôle atteint la fin d'un itérateur.

Dans les exemples précédents, le type de retour des itérateurs est IEnumerable<T> (dans les cas non génériques, utilisez IEnumerable comme type de retour d'un itérateur). Vous pouvez aussi utiliser IAsyncEnumerable<T> comme type de retour d'un itérateur. Cela rend un itérateur asynchrone. Utilisez l' instruction await foreach pour itérer sur le résultat de l'itérateur, comme le montre l'exemple suivant :

C#

```
await foreach (int n in GenerateNumbersAsync(5))
{
    Console.Write(n);
    Console.Write(" ");
}
// Output: 0 2 4 6 8

async IAsyncEnumerable<int> GenerateNumbersAsync(int count)
{
    for (int i = 0; i < count; i++)
    {
        yield return await ProduceNumberAsync(i);
    }
}

async Task<int> ProduceNumberAsync(int seed)
{
    await Task.Delay(1000);
    return 2 * seed;
}
```

IEnumerator<T> ou IEnumerator peuvent aussi être le type de retour d'un itérateur. Cela est utile quand vous implémentez la méthode `GetEnumerator` dans les scénarios suivants :

- Vous concevez le type qui implémente l'interface IEnumerable<T> ou IEnumerable.
- Vous ajoutez une méthode d'instance ou d'extension `GetEnumerator` pour permettre l'itération sur l'instance du type avec l'instruction foreach, comme le montre l'exemple suivant :

C#

```
public static void Example()
{
    var point = new Point(1, 2, 3);
    foreach (int coordinate in point)
    {
        Console.Write(coordinate);
        Console.Write(" ");
    }
    // Output: 1 2 3
}

public readonly record struct Point(int X, int Y, int Z)
{
    public IEnumerator<int> GetEnumerator()
    {
        yield return X;
        yield return Y;
        yield return Z;
    }
}
```

Vous ne pouvez pas utiliser les instructions `yield` dans :

- les méthodes avec des paramètres in, ref ou out
- les expressions lambda et les méthodes anonymes
- les méthodes qui contiennent des blocs unsafe

Exécution d'un itérateur

L'appel d'un itérateur n'a pas pour effet de l'exécuter immédiatement, comme le montre l'exemple suivant :

C#

```
var numbers = ProduceEvenNumbers(5);
Console.WriteLine("Caller: about to iterate.");
foreach (int i in numbers)
{
    Console.WriteLine($"Caller: {i}");
}

IEnumerable<int> ProduceEvenNumbers(int upto)
{
    Console.WriteLine("Iterator: start.");
    for (int i = 0; i <= upto; i += 2)
    {
        Console.WriteLine($"Iterator: about to yield {i}");
```

```
            yield return i;
            Console.WriteLine($"Iterator: yielded {i}");
        }
        Console.WriteLine("Iterator: end.");
    }
    // Output:
    // Caller: about to iterate.
    // Iterator: start.
    // Iterator: about to yield 0
    // Caller: 0
    // Iterator: yielded 0
    // Iterator: about to yield 2
    // Caller: 2
    // Iterator: yielded 2
    // Iterator: about to yield 4
    // Caller: 4
    // Iterator: yielded 4
    // Iterator: end.
```

Comme le montre l'exemple précédent, quand vous commencez à itérer sur le résultat d'un itérateur, un itérateur s'exécute jusqu'à ce que la première instruction `yield return` soit atteinte. Ensuite, l'exécution d'un itérateur est suspendue et l'appelant obtient la première valeur d'itération et la traite. À chaque itération suivante, l'exécution d'un itérateur reprend après l'instruction `yield return` qui a provoqué la suspension précédente et continue jusqu'à ce que l'instruction `yield return` suivante soit atteinte. L'itération se termine quand le contrôle atteint la fin d'un itérateur ou une instruction `yield break`.

spécification du langage C#

Pour plus d'informations, consultez la section Instruction yield de la spécification du langage C#.

Voir aussi

- Informations de référence sur C#
- Itérateurs
- Effectuer une itération dans des collections (C#)
- foreach
- await foreach

Mots clés de requête (référence C#)

Article • 20/07/2023

Cette section contient les mots clés contextuels utilisés dans les expressions de requête.

Contenu de cette section

Clause	Description
from	Spécifie une source de données et une variable de portée (semblable à une variable d'itération).
where	Filtre des éléments sources basés sur une ou plusieurs expressions booléennes séparées par des opérateurs AND et OR logiques (`&&` ou `||`).
select	Spécifie le type et la forme que les éléments de la séquence retournée auront une fois la requête exécutée.
groupe	Regroupe les résultats de la requête d'après une valeur de clé spécifiée.
into	Fournit un identificateur qui peut servir comme référence pour les résultats d'une clause join, group ou select.
orderby	Trie des résultats de la requête par ordre croissant ou décroissant selon le comparateur par défaut du type d'élément.
join	Joint deux sources de données selon une comparaison d'égalité entre deux critères de comparaison spécifiés.
let	Introduit une variable de portée pour stocker des résultats de sous-expression dans une expression de requête.
in	Mot clé contextuel dans une clause join.
actif	Mot clé contextuel dans une clause join.
equals	Mot clé contextuel dans une clause join.
by	Mot clé contextuel dans une clause group.
ascending	Mot clé contextuel dans une clause orderby.
descending	Mot clé contextuel dans une clause orderby.

Voir aussi

- Mots clés C#

- LINQ en C#

from, clause (Référence C#)

Article • 07/04/2023

Une expression de requête doit commencer par une clause `from`. Une expression de requête peut également contenir des sous-requêtes qui commencent elles aussi par une clause `from`. La clause `from` spécifie les éléments suivants :

- La source de données sur laquelle la requête ou la sous-requête est exécutée.
- Une *variable de portée locale* qui représente chaque élément dans la séquence source.

La variable de portée et la source de données sont toutes les deux fortement typées. La source de données référencée dans la clause `from` doit avoir un type IEnumerable, IEnumerable<T> ou un type dérivé comme IQueryable<T>.

Dans l'exemple suivant, `numbers` est la source de données tandis que `num` est la variable de portée. Notez que les deux variables sont fortement typées même si le mot clé var est utilisé.

C#

```
class LowNums
{
    static void Main()
    {
        // A simple data source.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query.
        // lowNums is an IEnumerable<int>
        var lowNums = from num in numbers
            where num < 5
            select num;

        // Execute the query.
        foreach (int i in lowNums)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 4 1 3 2 0
```

Variable de portée

Le compilateur déduit le type de la variable de portée quand la source de données implémente IEnumerable<T>. Par exemple, si la source a un type `IEnumerable<Customer>`, la variable de portée est déduite comme étant `Customer`. Vous devez spécifier le type de façon explicite uniquement quand la source est un type `IEnumerable` non générique comme ArrayList. Pour plus d'informations, consultez Guide pratique pour interroger une liste de tableaux avec LINQ.

Dans l'exemple précédent, `num` est déduit comme étant de type `int`. Comme la variable de portée est fortement typée, vous pouvez appeler des méthodes sur celle-ci ou l'utiliser dans d'autres opérations. Par exemple, au lieu d'écrire `select num`, vous pouvez écrire `select num.ToString()` pour que l'expression de requête retourne une séquence de chaînes à la place d'entiers. Vous pouvez également écrire `select num + 10` pour que l'expression retourne la séquence 14, 11, 13, 12, 10. Pour plus d'informations, consultez select, clause.

La variable de portée est semblable à une variable d'itération dans une instruction foreach à une différence près, très importante : une variable de portée ne stocke jamais réellement de données de la source. Il s'agit juste d'une syntaxe pratique qui permet à la requête de décrire ce qui se produira lors de son exécution. Pour plus d'informations, consultez Introduction aux requêtes LINQ (C#).

Clauses from composées

Dans certains cas, chaque élément dans la séquence source peut être une séquence ou contenir une séquence. Par exemple, votre source de données peut être un `IEnumerable<Student>` où chaque objet Student dans la séquence contient une liste de notes d'examens. Pour accéder à la liste interne dans chaque élément `Student`, vous pouvez utiliser des clauses `from` composées. Cette technique est proche de celle consistant à utiliser des instructions foreach imbriquées. Vous pouvez ajouter des clauses where ou orderby à l'une et l'autre des clauses `from` pour filtrer les résultats. L'exemple suivant présente une séquence d'objets `Student` et chacun contient un `List` interne des entiers qui représentent des notes d'examens. Pour accéder à la liste interne, utilisez une clause `from` composée. Vous pouvez insérer des clauses entre les deux clauses `from` si nécessaire.

C#

```
class CompoundFrom
{
    // The element type of the data source.
    public class Student
    {
```

```
        public string LastName { get; set; }
        public List<int> Scores {get; set;}
    }

    static void Main()
    {

        // Use a collection initializer to create the data source. Note that
        // each element in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
           new Student {LastName="Omelchenko", Scores= new List<int> {97,
72, 81, 60}},
           new Student {LastName="O'Donnell", Scores= new List<int> {75, 84,
91, 39}},
           new Student {LastName="Mortensen", Scores= new List<int> {88, 94,
65, 85}},
           new Student {LastName="Garcia", Scores= new List<int> {97, 89,
85, 82}},
           new Student {LastName="Beebe", Scores= new List<int> {35, 72, 91,
70}}
        };

        // Use a compound from to access the inner sequence within each
element.
        // Note the similarity to a nested foreach statement.
        var scoreQuery = from student in students
                         from score in student.Scores
                            where score > 90
                            select new { Last = student.LastName, score };

        // Execute the queries.
        Console.WriteLine("scoreQuery:");
        // Rest the mouse pointer on scoreQuery in the following line to
        // see its type. The type is IEnumerable<'a>, where 'a is an
        // anonymous type defined as new {string Last, int score}. That is,
        // each instance of this anonymous type has two members, a string
        // (Last) and an int (score).
        foreach (var student in scoreQuery)
        {
            Console.WriteLine("{0} Score: {1}", student.Last,
student.score);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
scoreQuery:
Omelchenko Score: 97
O'Donnell Score: 91
Mortensen Score: 94
Garcia Score: 97
```

```
Beebe Score: 91
*/
```

Utilisation de plusieurs clauses from pour effectuer des jointures

Une clause `from` composée est utilisée pour accéder aux collections internes dans une source de données unique. Toutefois, une requête peut également contenir plusieurs clauses `from` qui génèrent des requêtes supplémentaires de sources de données indépendantes. Cette technique vous permet d'effectuer certains types d'opérations de jointure qui ne sont pas possibles en utilisant la clause join.

L'exemple suivant montre comment utiliser deux clauses `from` pour former une jointure croisée complète de deux sources de données.

C#

```
class CompoundFrom2
{
    static void Main()
    {
        char[] upperCase = { 'A', 'B', 'C' };
        char[] lowerCase = { 'x', 'y', 'z' };

        // The type of joinQuery1 is IEnumerable<'a>, where 'a
        // indicates an anonymous type. This anonymous type has two
        // members, upper and lower, both of type char.
        var joinQuery1 =
            from upper in upperCase
            from lower in lowerCase
            select new { upper, lower };

        // The type of joinQuery2 is IEnumerable<'a>, where 'a
        // indicates an anonymous type. This anonymous type has two
        // members, upper and lower, both of type char.
        var joinQuery2 =
            from lower in lowerCase
            where lower != 'x'
            from upper in upperCase
            select new { lower, upper };

        // Execute the queries.
        Console.WriteLine("Cross join:");
        // Rest the mouse pointer on joinQuery1 to verify its type.
        foreach (var pair in joinQuery1)
        {
            Console.WriteLine("{0} is matched to {1}", pair.upper,
pair.lower);
```

```
        }

        Console.WriteLine("Filtered non-equijoin:");
        // Rest the mouse pointer over joinQuery2 to verify its type.
        foreach (var pair in joinQuery2)
        {
            Console.WriteLine("{0} is matched to {1}", pair.lower,
pair.upper);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
        Cross join:
        A is matched to x
        A is matched to y
        A is matched to z
        B is matched to x
        B is matched to y
        B is matched to z
        C is matched to x
        C is matched to y
        C is matched to z
        Filtered non-equijoin:
        y is matched to A
        y is matched to B
        y is matched to C
        z is matched to A
        z is matched to B
        z is matched to C
        */
```

Pour plus d'informations sur les opérations de jointure qui utilisent plusieurs clauses `from`, consultez Effectuer des jointures externes gauches.

Voir aussi

- Mots clés de requête (LINQ)
- LINQ (Language-Integrated Query)

where, clause (Référence C#)

Article • 20/07/2023

La clause `where` est utilisée dans une expression de requête pour spécifier les éléments de la source de données qui seront retournés dans l'expression de requête. Elle applique une condition booléenne (un *prédicat*) à chaque élément source (référencé par la variable de portée) et retourne ceux pour lesquels la condition spécifiée est remplie. Une expression de requête unique peut contenir plusieurs clauses `where` et une clause unique peut contenir plusieurs sous-expressions de prédicat.

Exemple 1

Dans l'exemple suivant, la clause `where` élimine par filtrage tous les nombres excepté ceux inférieurs à cinq. Si vous supprimez la clause `where`, tous les nombres de la source de données seront retournés. L'expression `num < 5` est le prédicat qui est appliqué à chaque élément.

C#

```
class WhereSample
{
    static void Main()
    {
        // Simple data source. Arrays support IEnumerable<T>.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Simple query with one predicate in where clause.
        var queryLowNums =
            from num in numbers
            where num < 5
            select num;

        // Execute the query.
        foreach (var s in queryLowNums)
        {
            Console.Write(s.ToString() + " ");
        }
    }
}
//Output: 4 1 3 2 0
```

Exemple 2

Dans une clause `where` unique, vous pouvez spécifier autant de prédicats que nécessaire à l'aide des opérateurs && et ||. Dans l'exemple suivant, la requête spécifie deux prédicats pour sélectionner uniquement les nombres pairs inférieurs à cinq.

C#

```
class WhereSample2
{
    static void Main()
    {
        // Data source.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query with two predicates in where clause.
        var queryLowNums2 =
            from num in numbers
            where num < 5 && num % 2 == 0
            select num;

        // Execute the query
        foreach (var s in queryLowNums2)
        {
            Console.Write(s.ToString() + " ");
        }
        Console.WriteLine();

        // Create the query with two where clause.
        var queryLowNums3 =
            from num in numbers
            where num < 5
            where num % 2 == 0
            select num;

        // Execute the query
        foreach (var s in queryLowNums3)
        {
            Console.Write(s.ToString() + " ");
        }
    }
}
// Output:
// 4 2 0
// 4 2 0
```

Exemple 3

Une clause `where` peut contenir une ou plusieurs méthodes qui retournent des valeurs booléennes. Dans l'exemple suivant, la clause `where` utilise une méthode pour déterminer si la valeur actuelle de la variable de portée est paire ou impaire.

C#

```
class WhereSample3
{
    static void Main()
    {
        // Data source
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query with a method call in the where clause.
        // Note: This won't work in LINQ to SQL unless you have a
        // stored procedure that is mapped to a method by this name.
        var queryEvenNums =
            from num in numbers
            where IsEven(num)
            select num;

         // Execute the query.
        foreach (var s in queryEvenNums)
        {
            Console.Write(s.ToString() + " ");
        }
    }

    // Method may be instance method or static method.
    static bool IsEven(int i)
    {
        return i % 2 == 0;
    }
}
//Output: 4 8 6 2 0
```

Remarques

La clause `where` est un mécanisme de filtrage. Elle peut être placée à presque n'importe quel endroit d'une expression de requête, sauf qu'elle ne peut pas être la première ni la dernière clause. Une clause `where` peut apparaître avant ou après une clause group selon que vous devez filtrer les éléments sources avant ou après leur regroupement.

Si un prédicat spécifié n'est pas valide pour les éléments de la source de données, une erreur de compilation est générée. C'est l'un des avantages de la vérification de type fort fourni par LINQ.

Au moment de la compilation, le mot clé `where` est converti en appel à la méthode d'opérateur de requête standard Where.

Voir aussi

- Mots clés de requête (LINQ)
- from, clause
- select, clause
- Filtrage des données
- LINQ en C#

select, clause (Référence C#)

Article • 07/04/2023

Dans une expression de requête, la clause `select` spécifie le type de valeurs qui sont générées quand la requête est exécutée. Le résultat est basé sur l'évaluation de toutes les clauses précédentes et sur toutes les expressions de la clause `select` elle-même. Une expression de requête doit se terminer par une clause `select` ou par une clause group.

L'exemple suivant présente une clause `select` simple dans une expression de requête.

C#

```
class SelectSample1
{
    static void Main()
    {
        //Create the data source
        List<int> Scores = new List<int>() { 97, 92, 81, 60 };

        // Create the query.
        IEnumerable<int> queryHighScores =
            from score in Scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in queryHighScores)
        {
            Console.Write(i + " ");
        }
    }
}
//Output: 97 92 81
```

Le type de la séquence générée par la clause `select` détermine le type de la variable de requête `queryHighScores`. Dans le cas le plus simple, la clause `select` spécifie uniquement la variable de portée. La séquence retournée contient alors des éléments du même type que la source de données. Pour plus d'informations, consultez Relations des types dans des opérations de requête LINQ. Toutefois, la clause `select` fournit également un mécanisme puissant pour transformer (ou *projeter*) les données sources en nouveaux types. Pour plus d'informations, consultez Transformations de données avec LINQ (C#).

Exemple

L'exemple suivant affiche l'ensemble des différentes formes que peut prendre une clause `select`. Dans chaque requête, notez la relation entre la clause `select` et le type de la *variable de requête* (`studentQuery1`, `studentQuery2`, etc.).

C#

```
    class SelectSample2
    {
        // Define some classes
        public class Student
        {
            public string First { get; set; }
            public string Last { get; set; }
            public int ID { get; set; }
            public List<int> Scores;
            public ContactInfo GetContactInfo(SelectSample2 app, int id)
            {
                ContactInfo cInfo =
                    (from ci in app.contactList
                    where ci.ID == id
                    select ci)
                    .FirstOrDefault();

                return cInfo;
            }

            public override string ToString()
            {
                return First + " " + Last + ":" + ID;
            }
        }

        public class ContactInfo
        {
            public int ID { get; set; }
            public string Email { get; set; }
            public string Phone { get; set; }
            public override string ToString() { return Email + "," + Phone;
}
        }

        public class ScoreInfo
        {
            public double Average { get; set; }
            public int ID { get; set; }
        }

        // The primary data source
        List<Student> students = new List<Student>()
        {
```

```
            new Student {First="Svetlana", Last="Omelchenko", ID=111,
Scores= new List<int>() {97, 92, 81, 60}},
            new Student {First="Claire", Last="O'Donnell", ID=112, Scores=
new List<int>() {75, 84, 91, 39}},
            new Student {First="Sven", Last="Mortensen", ID=113, Scores=
new List<int>() {88, 94, 65, 91}},
            new Student {First="Cesar", Last="Garcia", ID=114, Scores= new
List<int>() {97, 89, 85, 82}},
        };

        // Separate data source for contact info.
        List<ContactInfo> contactList = new List<ContactInfo>()
        {
            new ContactInfo {ID=111, Email="SvetlanO@Contoso.com",
Phone="206-555-0108"},
            new ContactInfo {ID=112, Email="ClaireO@Contoso.com",
Phone="206-555-0298"},
            new ContactInfo {ID=113, Email="SvenMort@Contoso.com",
Phone="206-555-1130"},
            new ContactInfo {ID=114, Email="CesarGar@Contoso.com",
Phone="206-555-0521"}
        };

        static void Main(string[] args)
        {
            SelectSample2 app = new SelectSample2();

            // Produce a filtered sequence of unmodified Students.
            IEnumerable<Student> studentQuery1 =
                from student in app.students
                where student.ID > 111
                select student;

            Console.WriteLine("Query1: select range_variable");
            foreach (Student s in studentQuery1)
            {
                Console.WriteLine(s.ToString());
            }

            // Produce a filtered sequence of elements that contain
            // only one property of each Student.
            IEnumerable<String> studentQuery2 =
                from student in app.students
                where student.ID > 111
                select student.Last;

            Console.WriteLine("\r\n studentQuery2: select
range_variable.Property");
            foreach (string s in studentQuery2)
            {
                Console.WriteLine(s);
            }

            // Produce a filtered sequence of objects created by
            // a method call on each Student.
```

```
            IEnumerable<ContactInfo> studentQuery3 =
                from student in app.students
                where student.ID > 111
                select student.GetContactInfo(app, student.ID);

            Console.WriteLine("\r\n studentQuery3: select
range_variable.Method");
            foreach (ContactInfo ci in studentQuery3)
            {
                Console.WriteLine(ci.ToString());
            }

            // Produce a filtered sequence of ints from
            // the internal array inside each Student.
            IEnumerable<int> studentQuery4 =
                from student in app.students
                where student.ID > 111
                select student.Scores[0];

            Console.WriteLine("\r\n studentQuery4: select
range_variable[index]");
            foreach (int i in studentQuery4)
            {
                Console.WriteLine("First score = {0}", i);
            }

            // Produce a filtered sequence of doubles
            // that are the result of an expression.
            IEnumerable<double> studentQuery5 =
                from student in app.students
                where student.ID > 111
                select student.Scores[0] * 1.1;

            Console.WriteLine("\r\n studentQuery5: select expression");
            foreach (double d in studentQuery5)
            {
                Console.WriteLine("Adjusted first score = {0}", d);
            }

            // Produce a filtered sequence of doubles that are
            // the result of a method call.
            IEnumerable<double> studentQuery6 =
                from student in app.students
                where student.ID > 111
                select student.Scores.Average();

            Console.WriteLine("\r\n studentQuery6: select expression2");
            foreach (double d in studentQuery6)
            {
                Console.WriteLine("Average = {0}", d);
            }

            // Produce a filtered sequence of anonymous types
            // that contain only two properties from each Student.
            var studentQuery7 =
```

```
                from student in app.students
                where student.ID > 111
                select new { student.First, student.Last };

            Console.WriteLine("\r\n studentQuery7: select new anonymous
type");
            foreach (var item in studentQuery7)
            {
                Console.WriteLine("{0}, {1}", item.Last, item.First);
            }

            // Produce a filtered sequence of named objects that contain
            // a method return value and a property from each Student.
            // Use named types if you need to pass the query variable
            // across a method boundary.
            IEnumerable<ScoreInfo> studentQuery8 =
                from student in app.students
                where student.ID > 111
                select new ScoreInfo
                {
                    Average = student.Scores.Average(),
                    ID = student.ID
                };

            Console.WriteLine("\r\n studentQuery8: select new named type");
            foreach (ScoreInfo si in studentQuery8)
            {
                Console.WriteLine("ID = {0}, Average = {1}", si.ID,
si.Average);
            }

            // Produce a filtered sequence of students who appear on a
contact list
            // and whose average is greater than 85.
            IEnumerable<ContactInfo> studentQuery9 =
                from student in app.students
                where student.Scores.Average() > 85
                join ci in app.contactList on student.ID equals ci.ID
                select ci;

            Console.WriteLine("\r\n studentQuery9: select result of join
clause");
            foreach (ContactInfo ci in studentQuery9)
            {
                Console.WriteLine("ID = {0}, Email = {1}", ci.ID, ci.Email);
            }

            // Keep the console window open in debug mode
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
            }
        }
    /* Output
        Query1: select range_variable
        Claire O'Donnell:112
```

```
        Sven Mortensen:113
        Cesar Garcia:114

        studentQuery2: select range_variable.Property
        O'Donnell
        Mortensen
        Garcia

        studentQuery3: select range_variable.Method
        ClaireO@Contoso.com,206-555-0298
        SvenMort@Contoso.com,206-555-1130
        CesarGar@Contoso.com,206-555-0521

        studentQuery4: select range_variable[index]
        First score = 75
        First score = 88
        First score = 97

        studentQuery5: select expression
        Adjusted first score = 82.5
        Adjusted first score = 96.8
        Adjusted first score = 106.7

        studentQuery6: select expression2
        Average = 72.25
        Average = 84.5
        Average = 88.25

        studentQuery7: select new anonymous type
        O'Donnell, Claire
        Mortensen, Sven
        Garcia, Cesar

        studentQuery8: select new named type
        ID = 112, Average = 72.25
        ID = 113, Average = 84.5
        ID = 114, Average = 88.25

        studentQuery9: select result of join clause
        ID = 114, Email = CesarGar@Contoso.com
*/
```

Comme indiqué dans `studentQuery8` dans l'exemple précédent, vous pouvez parfois souhaiter que les éléments de la séquence retournée contiennent uniquement un sous-ensemble des propriétés des éléments sources. En limitant au maximum la séquence retournée, vous pouvez réduire les besoins en ressources mémoire et augmenter la vitesse d'exécution de la requête. Pour ce faire, créez un type anonyme dans la clause `select` et utilisez un initialiseur d'objet pour l'initialiser avec les propriétés appropriées de l'élément source. Pour obtenir un exemple de la procédure à suivre, consultez Initialiseurs d'objet et de collection.

Notes

Au moment de la compilation, la clause `select` traduite en un appel de méthode à l'opérateur de requête standard Select.

Voir aussi

- Référence C#
- Mots clés de requête (LINQ)
- from, clause
- partial, méthode (Référence C#)
- Types anonymes
- LINQ en C#
- LINQ (Language-Integrated Query)

group, clause (Référence C#)

Article • 07/04/2023

La clause `group` retourne une séquence d'objets IGrouping<TKey,TElement> qui contient zéro, un ou plusieurs éléments qui correspondent à la valeur de clé du groupe. Par exemple, vous pouvez regrouper une séquence de chaînes en fonction de la première lettre de chaque chaîne. Dans ce cas, la première lettre est la clé, elle a le type char et elle est stockée dans la propriété `Key` de chaque objet IGrouping<TKey,TElement>. Le compilateur déduit le type de la clé.

Vous pouvez terminer une expression de requête avec une clause `group`, comme illustré dans l'exemple suivant :

C#

```
// Query variable is an IEnumerable<IGrouping<char, Student>>
var studentQuery1 =
    from student in students
    group student by student.Last[0];
```

Si vous souhaitez effectuer des opérations de requête supplémentaires sur chaque groupe, vous pouvez spécifier un identificateur temporaire en utilisant le mot clé contextuel into. Quand vous utilisez `into`, vous devez continuer la requête et finalement la terminer avec une instruction `select` ou une autre clause `group`, comme illustré dans l'extrait suivant :

C#

```
// Group students by the first letter of their last name
// Query variable is an IEnumerable<IGrouping<char, Student>>
var studentQuery2 =
    from student in students
    group student by student.Last[0] into g
    orderby g.Key
    select g;
```

Des exemples d'utilisation plus complets de `group` avec et sans `into` sont fournis dans la section Exemple de cet article.

Énumération des résultats d'une requête de groupe

Étant donné que les objets IGrouping<TKey,TElement> générés par une requête `group` sont essentiellement une liste de listes, vous devez utiliser une boucle foreach imbriquée pour accéder aux éléments dans chaque groupe. La boucle externe itère les clés de groupe, et la boucle interne itère chaque élément dans le groupe proprement dit. Un groupe peut avoir une clé mais pas d'éléments. Voici la boucle `foreach` qui exécute la requête dans les exemples de code précédents :

C#

```
// Iterate group items with a nested foreach. This IGrouping encapsulates
// a sequence of Student objects, and a Key of type char.
// For convenience, var can also be used in the foreach statement.
foreach (IGrouping<char, Student> studentGroup in studentQuery2)
{
     Console.WriteLine(studentGroup.Key);
     // Explicit type for student could also be used here.
     foreach (var student in studentGroup)
     {
         Console.WriteLine("   {0}, {1}", student.Last, student.First);
     }
 }
```

Types de clés

Les clés de groupes peuvent être de tout type, comme une chaîne, un type numérique intégré, ou un type nommé ou un type anonyme défini par l'utilisateur.

Regroupement par chaîne

Les exemples de code précédents utilisaient un `char`. On aurait facilement pu spécifier une clé de chaîne à la place, par exemple le nom de famille complet :

C#

```
// Same as previous example except we use the entire last name as a key.
// Query variable is an IEnumerable<IGrouping<string, Student>>
var studentQuery3 =
    from student in students
    group student by student.Last;
```

Regroupement par valeur booléenne

L'exemple suivant illustre l'utilisation d'une valeur booléenne pour une clé afin de répartir les résultats en deux groupes. Notez que la valeur est générée par une sous-

expression dans la clause `group`.

C#

```
class GroupSample1
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
        public List<int> Scores;
    }

    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source. Note that
each element
        //  in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
           new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores=
new List<int> {97, 72, 81, 60}},
           new Student {First="Claire", Last="O'Donnell", ID=112, Scores=
new List<int> {75, 84, 91, 39}},
           new Student {First="Sven", Last="Mortensen", ID=113, Scores= new
List<int> {99, 89, 91, 95}},
           new Student {First="Cesar", Last="Garcia", ID=114, Scores= new
List<int> {72, 81, 65, 84}},
           new Student {First="Debra", Last="Garcia", ID=115, Scores= new
List<int> {97, 89, 85, 82}}
        };

        return students;
    }

    static void Main()
    {
        // Obtain the data source.
        List<Student> students = GetStudents();

        // Group by true or false.
        // Query variable is an IEnumerable<IGrouping<bool, Student>>
        var booleanGroupQuery =
            from student in students
            group student by student.Scores.Average() >= 80; //pass or fail!

        // Execute the query and access items in each group
        foreach (var studentGroup in booleanGroupQuery)
        {
            Console.WriteLine(studentGroup.Key == true ? "High averages" :
"Low averages");
            foreach (var student in studentGroup)
```

```
            {
                Console.WriteLine("   {0}, {1}:{2}", student.Last,
student.First, student.Scores.Average());
            }
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
  Low averages
   Omelchenko, Svetlana:77.5
   O'Donnell, Claire:72.25
   Garcia, Cesar:75.5
  High averages
   Mortensen, Sven:93.5
   Garcia, Debra:88.25
*/
```

Regroupement par plage numérique

L'exemple suivant utilise une expression pour créer des clés de groupes numériques qui représentent une plage de centiles. Notez l'utilisation de let comme emplacement pratique pour stocker un résultat d'appel de méthode, qui vous évite d'avoir à appeler deux fois la méthode dans la clause `group`. Pour plus d'informations sur la façon d'utiliser en toute sécurité des méthodes dans des expressions de requête, consultez Guide pratique pour gérer des exceptions dans des expressions de requête.

C#

```
class GroupSample2
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
        public List<int> Scores;
    }

    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source. Note that
each element
        //  in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
```

```
            new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores=
new List<int> {97, 72, 81, 60}},
            new Student {First="Claire", Last="O'Donnell", ID=112, Scores=
new List<int> {75, 84, 91, 39}},
            new Student {First="Sven", Last="Mortensen", ID=113, Scores= new
List<int> {99, 89, 91, 95}},
            new Student {First="Cesar", Last="Garcia", ID=114, Scores= new
List<int> {72, 81, 65, 84}},
            new Student {First="Debra", Last="Garcia", ID=115, Scores= new
List<int> {97, 89, 85, 82}}
        };

        return students;
    }

    // This method groups students into percentile ranges based on their
    // grade average. The Average method returns a double, so to produce a
whole
    // number it is necessary to cast to int before dividing by 10.
    static void Main()
    {
        // Obtain the data source.
        List<Student> students = GetStudents();

        // Write the query.
        var studentQuery =
            from student in students
            let avg = (int)student.Scores.Average()
            group student by (avg / 10) into g
            orderby g.Key
            select g;

        // Execute the query.
        foreach (var studentGroup in studentQuery)
        {
            int temp = studentGroup.Key * 10;
            Console.WriteLine("Students with an average between {0} and
{1}", temp, temp + 10);
            foreach (var student in studentGroup)
            {
                Console.WriteLine("   {0}, {1}:{2}", student.Last,
student.First, student.Scores.Average());
            }
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
     Students with an average between 70 and 80
       Omelchenko, Svetlana:77.5
       O'Donnell, Claire:72.25
       Garcia, Cesar:75.5
```

```
    Students with an average between 80 and 90
      Garcia, Debra:88.25
    Students with an average between 90 and 100
      Mortensen, Sven:93.5
*/
```

Regroupement par clés composites

Utilisez une clé composite quand vous souhaitez regrouper des éléments en fonction de plusieurs clés. Vous créez une clé composite en utilisant un type anonyme ou un type nommé pour contenir l'élément clé. Dans l'exemple suivant, supposons qu'une classe `Person` a été déclarée avec les membres nommés `surname` et `city`. La clause `group` provoque la création d'un groupe distinct pour chaque ensemble de personnes ayant le même nom de famille et la même ville.

C#

```
group person by new {name = person.surname, city = person.city};
```

Utilisez un type nommé si vous devez passer la variable de requête à une autre méthode. Créez une classe spéciale à l'aide de propriétés implémentées automatiquement pour les clés, puis substituez les méthodes Equals et GetHashCode. Vous pouvez également utiliser un struct, auquel cas vous n'êtes pas obligé de substituer ces méthodes. Pour plus d'informations, consultez Guide pratique pour implémenter une classe Lightweight avec des propriétés implémentées automatiquement et Guide pratique pour interroger des fichiers dupliqués dans une arborescence de répertoires. Ce dernier article contient un exemple de code qui montre comment utiliser une clé composite avec un type nommé.

Exemple 1

L'exemple suivant montre le modèle standard pour organiser des données sources en groupes quand aucune logique de requête supplémentaire n'est appliquée aux groupes. Il s'agit alors d'un regroupement sans continuation. Les éléments du tableau de chaînes sont regroupés en fonction de leur première lettre. Le résultat de la requête est un type IGrouping<TKey,TElement> contenant une propriété `Key` publique de type `char` et une collection IEnumerable<T> qui contient chaque élément du regroupement.

Le résultat d'une clause `group` est une séquence de séquences. Ainsi, pour accéder aux différents éléments de chaque groupe retourné, vous devez utiliser une boucle `foreach`

imbriquée à l'intérieur de la boucle qui itère les clés de groupe, comme illustré dans l'exemple suivant.

C#

```
class GroupExample1
{
    static void Main()
    {
        // Create a data source.
        string[] words = { "blueberry", "chimpanzee", "abacus", "banana",
"apple", "cheese" };

        // Create the query.
        var wordGroups =
            from w in words
            group w by w[0];

        // Execute the query.
        foreach (var wordGroup in wordGroups)
        {
            Console.WriteLine("Words that start with the letter '{0}':",
wordGroup.Key);
            foreach (var word in wordGroup)
            {
                Console.WriteLine(word);
            }
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
      Words that start with the letter 'b':
        blueberry
        banana
      Words that start with the letter 'c':
        chimpanzee
        cheese
      Words that start with the letter 'a':
        abacus
        apple
     */
```

Exemple 2

Cet exemple montre comment exécuter une logique supplémentaire sur les groupes après les avoir créés, à l'aide une *continuation* avec `into`. Pour plus d'informations,

consultez into. L'exemple suivant interroge chaque groupe pour sélectionner uniquement ceux dont la valeur de clé est une voyelle.

C#

```
class GroupClauseExample2
{
    static void Main()
    {
        // Create the data source.
        string[] words2 = { "blueberry", "chimpanzee", "abacus", "banana",
"apple", "cheese", "elephant", "umbrella", "anteater" };

        // Create the query.
        var wordGroups2 =
            from w in words2
            group w by w[0] into grps
            where (grps.Key == 'a' || grps.Key == 'e' || grps.Key == 'i'
                   || grps.Key == 'o' || grps.Key == 'u')
            select grps;

        // Execute the query.
        foreach (var wordGroup in wordGroups2)
        {
            Console.WriteLine("Groups that start with a vowel: {0}",
wordGroup.Key);
            foreach (var word in wordGroup)
            {
                Console.WriteLine("   {0}", word);
            }
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Groups that start with a vowel: a
        abacus
        apple
        anteater
    Groups that start with a vowel: e
        elephant
    Groups that start with a vowel: u
        umbrella
*/
```

Remarques

Lors de la compilation, les clauses `group` sont traduites en appels à la méthode GroupBy.

Voir aussi

- IGrouping<TKey,TElement>
- GroupBy
- ThenBy
- ThenByDescending
- Mots clés de requête
- LINQ (Language-Integrated Query)
- Créer un groupe imbriqué
- Regrouper les résultats d'une requête
- Effectuer une sous-requête sur une opération de regroupement

into (Référence C#)

Article • 07/04/2023

Le mot clé contextuel `into` permet de créer un identificateur temporaire pour stocker les résultats d'une clause group, join ou select dans un nouvel identificateur. Cet identificateur peut lui-même être un générateur pour d'autres commandes de requête. Quand il est utilisé dans une clause `group` ou `select`, le nouvel identificateur est parfois appelé *continuation*.

Exemple

L'exemple suivant illustre l'utilisation du mot clé `into` pour activer un identificateur temporaire `fruitGroup` dont le type déduit est `IGrouping`. En utilisant l'identificateur, vous pouvez appeler la méthode Count dans chaque groupe et sélectionner uniquement les groupes qui contiennent deux mots ou plus.

C#

```
class IntoSample1
{
    static void Main()
    {

        // Create a data source.
        string[] words = { "apples", "blueberries", "oranges", "bananas",
"apricots"};

        // Create the query.
        var wordGroups1 =
            from w in words
            group w by w[0] into fruitGroup
            where fruitGroup.Count() >= 2
            select new { FirstLetter = fruitGroup.Key, Words =
fruitGroup.Count() };

        // Execute the query. Note that we only iterate over the groups,
        // not the items in each group
        foreach (var item in wordGroups1)
        {
            Console.WriteLine(" {0} has {1} elements.", item.FirstLetter,
item.Words);
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
```

```
}
/* Output:
   a has 2 elements.
   b has 2 elements.
*/
```

L'utilisation de `into` dans une clause `group` n'est nécessaire que si vous voulez effectuer des opérations de requête supplémentaires dans chaque groupe. Pour plus d'informations, consultez group, clause.

Pour obtenir un exemple d'utilisation de `into` dans une clause `join`, consultez join, clause.

Voir aussi

- Mots clés de requête (LINQ)
- LINQ en C#
- group, clause

orderby, clause (Référence C#)

Article • 07/04/2023

Dans une expression de requête, la clause `orderby` a pour effet de trier la séquence ou la sous-séquence (groupe) retournée par ordre croissant ou décroissant. Il est possible de spécifier plusieurs clés de façon à effectuer une ou plusieurs opérations de tri secondaire. Le tri est effectué par le comparateur par défaut du type de l'élément. L'ordre de tri par défaut est croissant. Vous pouvez aussi spécifier un comparateur personnalisé. Cependant, sa disponibilité est soumise à l'utilisation d'une syntaxe fondée sur une méthode. Pour plus d'informations, consultez Tri des données.

Exemple 1

Dans l'exemple suivant, la première requête trie les mots par ordre alphabétique à partir de A, tandis que la deuxième requête trie les mêmes mots par ordre décroissant. (Le mot clé `ascending` est la valeur de tri par défaut et peut être omis.)

C#

```
class OrderbySample1
{
    static void Main()
    {
        // Create a delicious data source.
        string[] fruits = { "cherry", "apple", "blueberry" };

        // Query for ascending sort.
        IEnumerable<string> sortAscendingQuery =
            from fruit in fruits
            orderby fruit //"ascending" is default
            select fruit;

        // Query for descending sort.
        IEnumerable<string> sortDescendingQuery =
            from w in fruits
            orderby w descending
            select w;

        // Execute the query.
        Console.WriteLine("Ascending:");
        foreach (string s in sortAscendingQuery)
        {
            Console.WriteLine(s);
        }

        // Execute the query.
        Console.WriteLine(Environment.NewLine + "Descending:");
```

```
        foreach (string s in sortDescendingQuery)
        {
            Console.WriteLine(s);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
Ascending:
apple
blueberry
cherry

Descending:
cherry
blueberry
apple
*/
```

Exemple 2

L'exemple suivant effectue un tri principal sur les noms des étudiants, puis un tri secondaire sur leurs prénoms.

C#

```
class OrderbySample2
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
    }

    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source. Note that
each element
        //  in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
           new Student {First="Svetlana", Last="Omelchenko", ID=111},
           new Student {First="Claire", Last="O'Donnell", ID=112},
           new Student {First="Sven", Last="Mortensen", ID=113},
           new Student {First="Cesar", Last="Garcia", ID=114},
           new Student {First="Debra", Last="Garcia", ID=115}
```

```
        };

        return students;
    }
    static void Main(string[] args)
    {
        // Create the data source.
        List<Student> students = GetStudents();

        // Create the query.
        IEnumerable<Student> sortedStudents =
            from student in students
            orderby student.Last ascending, student.First ascending
            select student;

        // Execute the query.
        Console.WriteLine("sortedStudents:");
        foreach (Student student in sortedStudents)
            Console.WriteLine(student.Last + " " + student.First);

        // Now create groups and sort the groups. The query first sorts the
names
        // of all students so that they will be in alphabetical order after
they are
        // grouped. The second orderby sorts the group keys in alpha order.
        var sortedGroups =
            from student in students
            orderby student.Last, student.First
            group student by student.Last[0] into newGroup
            orderby newGroup.Key
            select newGroup;

        // Execute the query.
        Console.WriteLine(Environment.NewLine + "sortedGroups:");
        foreach (var studentGroup in sortedGroups)
        {
            Console.WriteLine(studentGroup.Key);
            foreach (var student in studentGroup)
            {
                Console.WriteLine("   {0}, {1}", student.Last,
student.First);
            }
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
sortedStudents:
Garcia Cesar
Garcia Debra
Mortensen Sven
O'Donnell Claire
```

```
Omelchenko Svetlana

sortedGroups:
G
   Garcia, Cesar
   Garcia, Debra
M
   Mortensen, Sven
O
   O'Donnell, Claire
   Omelchenko, Svetlana
*/
```

Notes

Au moment de la compilation, la clause `orderby` est traduite en appel à la méthode OrderBy. Les différentes clés présentes dans la clause `orderby` sont traduites en appels à la méthode ThenBy.

Voir aussi

- Référence C#
- Mots clés de requête (LINQ)
- LINQ en C#
- group, clause
- LINQ (Language-Integrated Query)

join, clause (référence C#)

Article • 07/04/2023

La clause `join` est utile pour associer des éléments de différentes séquences sources qui n'ont pas de relation directe dans le modèle objet. Le seul impératif est que les éléments de chaque source partagent une valeur dont l'égalité peut être comparée. Par exemple, un distributeur de produits alimentaires peut avoir une liste de fournisseurs d'un certain produit et une liste d'acheteurs. Une clause `join` peut être utilisée par exemple pour créer une liste des fournisseurs et des acheteurs de ce produit qui se trouvent tous dans la même région spécifiée.

Une clause `join` prend deux séquences sources comme entrée. Les éléments de chaque séquence doivent être ou contenir une propriété qui peut être comparée à une propriété correspondante dans l'autre séquence. La clause `join` utilise le mot clé spécial `equals` pour comparer les clés spécifiées et déterminer si elles sont égales. Toutes les jointures effectuées par la clause `join` sont des équijointures. La forme de la sortie d'une clause `join` dépend du type spécifique de jointure que vous effectuez. Les trois types de jointure les plus courants sont les suivants :

- Jointure interne
- Jointure groupée
- Jointure externe gauche

Jointure interne

L'exemple suivant montre une équijointure interne simple. Cette requête produit une séquence plate de paires « nom de produit / catégorie ». La même chaîne de catégorie apparaît dans plusieurs éléments. Si un élément de `categories` n'a pas de `products` correspondant, cette catégorie n'apparaît pas dans les résultats.

C#

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name };
//produces flat sequence
```

Pour plus d'informations, consultez Effectuer des jointures internes.

Jointure groupée

Une clause `join` avec une expression `into` est appelée une jointure groupée.

```
C#
```

```
var innerGroupJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID into
prodGroup
    select new { CategoryName = category.Name, Products = prodGroup };
```

Une jointure groupée produit une séquence de résultats hiérarchique, qui associe des éléments de la séquence source de gauche à un ou plusieurs éléments correspondants de la séquence source de droite. Une jointure groupée n'a pas d'équivalent en termes relationnels ; il s'agit en fait d'une séquence de tableaux d'objets.

Si aucun élément de la séquence source de droite n'est trouvé en correspondance avec un élément de la source de gauche, la clause `join` produit un tableau vide pour cet élément. Par conséquent, la jointure groupée est fondamentalement une équijointure interne, excepté que la séquence de résultats est organisée en groupes.

Si vous sélectionnez les résultats d'une jointure groupée, vous pouvez accéder aux éléments, mais vous ne pouvez pas identifier leur clé de correspondance. Par conséquent, il est généralement plus utile de sélectionner les résultats de la jointure groupée dans un nouveau type qui a aussi le nom de clé, comme illustré dans l'exemple précédent.

Bien sûr, vous pouvez aussi utiliser le résultat d'une jointure groupée comme générateur d'une autre sous-requête :

```
C#
```

```
var innerGroupJoinQuery2 =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID into
prodGroup
    from prod2 in prodGroup
    where prod2.UnitPrice > 2.50M
    select prod2;
```

Pour plus d'informations, consultez Effectuer des jointures groupées.

Jointure externe gauche

Dans une jointure externe gauche, tous les éléments de la séquence source de gauche sont retournés, même si aucun élément correspondant ne se trouve dans la séquence de droite. Pour effectuer une jointure externe gauche dans LINQ, utilisez la méthode `DefaultIfEmpty` en combinaison avec une jointure groupée pour spécifier un élément droit par défaut à créer si un élément de gauche n'a pas de correspondance. Vous pouvez utiliser `null` comme valeur par défaut pour tous les types référence ou vous pouvez spécifier un type par défaut défini par l'utilisateur. L'exemple suivant montre un type par défaut défini par l'utilisateur :

C#

```
var leftOuterJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID into
prodGroup
    from item in prodGroup.DefaultIfEmpty(new Product { Name = String.Empty,
CategoryID = 0 })
    select new { CatName = category.Name, ProdName = item.Name };
```

Pour plus d'informations, consultez Effectuer des jointures externes gauches.

Opérateur Égal

Une clause `join` effectue une équijointure. En d'autres termes, vous pouvez baser les correspondances seulement sur l'égalité de deux clés. Les autres types de comparaisons, comme « supérieur à » ou « différent de », ne sont pas pris en charge. Pour indiquer clairement que toutes les jointures sont des équijointures, la clause `join` utilise le mot clé `equals` au lieu de l'opérateur `==`. Le mot clé `equals` ne peut être utilisé que dans une clause `join` et il diffère de l'opérateur `==` sur certains points importants. Lors de la comparaison de chaînes, `equals` présente une surcharge à comparer par valeur et l'opérateur `==` utilise l'égalité de référence. Lorsque les deux côtés de la comparaison présentent des variables de chaîne identiques, `equals` et `==` atteignent le même résultat : true. En effet, lorsqu'un programme déclare au moins deux variables de chaîne équivalentes, le compilateur les stocke toutes au même emplacement. C'est ce qu'on appelle la *centralisation*. La comparaison des valeurs Null constitue une autre différence importante : `null equals null` est évalué comme false avec l'opérateur `equals`, alors que l'opérateur `==` l'évalue comme true. Enfin, le comportement lié à l'étendue est différent : avec `equals`, la clé de gauche consomme la séquence source externe et la clé de droite consomme la source interne. La source externe est seulement dans l'étendue du côté gauche de `equals` et la séquence source interne est seulement dans l'étendue du côté droit.

Non-équijointures

Vous pouvez effectuer des non-équijointures, des jointures croisées et d'autres opérations de jointure personnalisées en utilisant plusieurs clauses `from` pour introduire indépendamment de nouvelles séquences dans une requête. Pour plus d'informations, consultez Effectuer des opérations de jointure personnalisées.

Jointures sur des collections d'objets et sur des tables relationnelles

Dans une expression de requête LINQ, les opérations de jointure sont effectuées sur les collections d'objets. Les collections d'objets ne peuvent pas être « jointes » exactement de la même façon que deux tables relationnelles. Dans LINQ, les clauses `join` explicites sont uniquement nécessaires lorsque deux séquences sources ne sont liées par aucune relation. Lorsque vous utilisez LINQ to SQL, les tables de clés étrangères sont représentées dans le modèle objet en tant que propriétés de la table principale. Par exemple, dans la base de données Northwind, la table Customer a une relation de clé étrangère avec la table Orders. Quand vous mappez les tables au modèle objet, la classe Customer a une propriété Orders qui contient la collection de commandes associées à ce client. En réalité, la jointure a déjà été effectuée pour vous.

Pour plus d'informations sur l'interrogation de tables liées dans le contexte de LINQ to SQL, consultez Guide pratique pour mapper des relations de base de données.

Clés composites

Vous pouvez tester l'égalité de plusieurs valeurs en utilisant une clé composite. Pour plus d'informations, consultez Effectuer des opérations de jointure à l'aide de clés composites. Vous pouvez aussi utiliser des clés composites dans une clause `group`.

Exemple

L'exemple suivant compare les résultats d'une jointure interne, d'une jointure groupée et d'une jointure externe gauche sur les mêmes sources de données en utilisant les mêmes clés de correspondance. Du code supplémentaire est ajouté à ces exemples pour clarifier les résultats dans l'affichage de la console.

```
C#
```

```
class JoinDemonstration
{
    #region Data

    class Product
    {
        public string Name { get; set; }
        public int CategoryID { get; set; }
    }

    class Category
    {
        public string Name { get; set; }
        public int ID { get; set; }
    }

    // Specify the first data source.
    List<Category> categories = new List<Category>()
    {
        new Category {Name="Beverages", ID=001},
        new Category {Name="Condiments", ID=002},
        new Category {Name="Vegetables", ID=003},
        new Category {Name="Grains", ID=004},
        new Category {Name="Fruit", ID=005}
    };

    // Specify the second data source.
    List<Product> products = new List<Product>()
   {
      new Product {Name="Cola",  CategoryID=001},
      new Product {Name="Tea",  CategoryID=001},
      new Product {Name="Mustard", CategoryID=002},
      new Product {Name="Pickles", CategoryID=002},
      new Product {Name="Carrots", CategoryID=003},
      new Product {Name="Bok Choy", CategoryID=003},
      new Product {Name="Peaches", CategoryID=005},
      new Product {Name="Melons", CategoryID=005},
    };
    #endregion

    static void Main(string[] args)
    {
        JoinDemonstration app = new JoinDemonstration();

        app.InnerJoin();
        app.GroupJoin();
        app.GroupInnerJoin();
        app.GroupJoin3();
        app.LeftOuterJoin();
        app.LeftOuterJoin2();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
```

```
    }

    void InnerJoin()
    {
        // Create the query that selects
        // a property from each element.
        var innerJoinQuery =
           from category in categories
           join prod in products on category.ID equals prod.CategoryID
           select new { Category = category.ID, Product = prod.Name };

        Console.WriteLine("InnerJoin:");
        // Execute the query. Access results
        // with a simple foreach statement.
        foreach (var item in innerJoinQuery)
        {
            Console.WriteLine("{0,-10}{1}", item.Product, item.Category);
        }
        Console.WriteLine("InnerJoin: {0} items in 1 group.",
innerJoinQuery.Count());
        Console.WriteLine(System.Environment.NewLine);
    }

    void GroupJoin()
    {
        // This is a demonstration query to show the output
        // of a "raw" group join. A more typical group join
        // is shown in the GroupInnerJoin method.
        var groupJoinQuery =
           from category in categories
           join prod in products on category.ID equals prod.CategoryID into
prodGroup
           select prodGroup;

        // Store the count of total items (for demonstration only).
        int totalItems = 0;

        Console.WriteLine("Simple GroupJoin:");

        // A nested foreach statement is required to access group items.
        foreach (var prodGrouping in groupJoinQuery)
        {
            Console.WriteLine("Group:");
            foreach (var item in prodGrouping)
            {
                totalItems++;
                Console.WriteLine("   {0,-10}{1}", item.Name,
item.CategoryID);
            }
        }
        Console.WriteLine("Unshaped GroupJoin: {0} items in {1} unnamed
groups", totalItems, groupJoinQuery.Count());
        Console.WriteLine(System.Environment.NewLine);
    }
```

```
    void GroupInnerJoin()
    {
        var groupJoinQuery2 =
            from category in categories
            orderby category.ID
            join prod in products on category.ID equals prod.CategoryID into
prodGroup
            select new
            {
                Category = category.Name,
                Products = from prod2 in prodGroup
                           orderby prod2.Name
                           select prod2
            };

        //Console.WriteLine("GroupInnerJoin:");
        int totalItems = 0;

        Console.WriteLine("GroupInnerJoin:");
        foreach (var productGroup in groupJoinQuery2)
        {
            Console.WriteLine(productGroup.Category);
            foreach (var prodItem in productGroup.Products)
            {
                totalItems++;
                Console.WriteLine("  {0,-10} {1}", prodItem.Name,
prodItem.CategoryID);
            }
        }
        Console.WriteLine("GroupInnerJoin: {0} items in {1} named groups",
totalItems, groupJoinQuery2.Count());
        Console.WriteLine(System.Environment.NewLine);
    }

    void GroupJoin3()
    {

        var groupJoinQuery3 =
            from category in categories
            join product in products on category.ID equals
product.CategoryID into prodGroup
            from prod in prodGroup
            orderby prod.CategoryID
            select new { Category = prod.CategoryID, ProductName = prod.Name
};

        //Console.WriteLine("GroupInnerJoin:");
        int totalItems = 0;

        Console.WriteLine("GroupJoin3:");
        foreach (var item in groupJoinQuery3)
        {
            totalItems++;
            Console.WriteLine("   {0}:{1}", item.ProductName,
item.Category);
```

```
        }

        Console.WriteLine("GroupJoin3: {0} items in 1 group", totalItems);
        Console.WriteLine(System.Environment.NewLine);
    }

    void LeftOuterJoin()
    {
        // Create the query.
        var leftOuterQuery =
           from category in categories
           join prod in products on category.ID equals prod.CategoryID into
prodGroup
           select prodGroup.DefaultIfEmpty(new Product() { Name =
"Nothing!", CategoryID = category.ID });

        // Store the count of total items (for demonstration only).
        int totalItems = 0;

        Console.WriteLine("Left Outer Join:");

        // A nested foreach statement  is required to access group items
        foreach (var prodGrouping in leftOuterQuery)
        {
            Console.WriteLine("Group:");
            foreach (var item in prodGrouping)
            {
                totalItems++;
                Console.WriteLine("  {0,-10}{1}", item.Name,
item.CategoryID);
            }
        }
        Console.WriteLine("LeftOuterJoin: {0} items in {1} groups",
totalItems, leftOuterQuery.Count());
        Console.WriteLine(System.Environment.NewLine);
    }

    void LeftOuterJoin2()
    {
        // Create the query.
        var leftOuterQuery2 =
           from category in categories
           join prod in products on category.ID equals prod.CategoryID into
prodGroup
           from item in prodGroup.DefaultIfEmpty()
           select new { Name = item == null ? "Nothing!" : item.Name,
CategoryID = category.ID };

        Console.WriteLine("LeftOuterJoin2: {0} items in 1 group",
leftOuterQuery2.Count());
        // Store the count of total items
        int totalItems = 0;

        Console.WriteLine("Left Outer Join 2:");
```

```
        // Groups have been flattened.
        foreach (var item in leftOuterQuery2)
        {
            totalItems++;
            Console.WriteLine("{0,-10}{1}", item.Name, item.CategoryID);
        }
        Console.WriteLine("LeftOuterJoin2: {0} items in 1 group",
totalItems);
    }
}
/*Output:

InnerJoin:
Cola      1
Tea       1
Mustard   2
Pickles   2
Carrots   3
Bok Choy  3
Peaches   5
Melons    5
InnerJoin: 8 items in 1 group.


Unshaped GroupJoin:
Group:
    Cola      1
    Tea       1
Group:
    Mustard   2
    Pickles   2
Group:
    Carrots   3
    Bok Choy  3
Group:
Group:
    Peaches   5
    Melons    5
Unshaped GroupJoin: 8 items in 5 unnamed groups


GroupInnerJoin:
Beverages
    Cola       1
    Tea        1
Condiments
    Mustard    2
    Pickles    2
Vegetables
    Bok Choy   3
    Carrots    3
Grains
Fruit
    Melons     5
    Peaches    5
```

```
GroupInnerJoin: 8 items in 5 named groups


GroupJoin3:
    Cola:1
    Tea:1
    Mustard:2
    Pickles:2
    Carrots:3
    Bok Choy:3
    Peaches:5
    Melons:5
GroupJoin3: 8 items in 1 group


Left Outer Join:
Group:
    Cola      1
    Tea       1
Group:
    Mustard   2
    Pickles   2
Group:
    Carrots   3
    Bok Choy  3
Group:
    Nothing!  4
Group:
    Peaches   5
    Melons    5
LeftOuterJoin: 9 items in 5 groups


LeftOuterJoin2: 9 items in 1 group
Left Outer Join 2:
Cola      1
Tea       1
Mustard   2
Pickles   2
Carrots   3
Bok Choy  3
Nothing!  4
Peaches   5
Melons    5
LeftOuterJoin2: 9 items in 1 group
Press any key to exit.
*/
```

Remarques

Une clause `join` qui n'est pas suivie de `into` se traduit par l'appel de la méthode Join.
Une clause `join` qui n'est pas suivie de `into` se traduit par l'appel de la méthode GroupJoin.

Voir aussi

- Mots clés de requête (LINQ)
- LINQ (Language-Integrated Query)
- Opérations de jointure
- group, clause
- Effectuer des jointures externes gauches
- Effectuer des jointures internes
- Effectuer des jointures groupées
- Classer les résultats d'une clause join
- Effectuer des jointures à l'aide de clés composites
- Systèmes de base de données compatibles pour Visual Studio

let, clause (Référence C#)

Article • 07/04/2023

Dans une expression de requête, il est parfois utile de stocker le résultat d'une sous-expression pour pouvoir l'utiliser dans des clauses ultérieures. Pour cela, vous pouvez utiliser le mot clé `let`, qui crée une variable de portée et l'initialise avec le résultat de l'expression que vous fournissez. Une fois initialisée avec une valeur, la variable de portée ne peut pas être utilisée pour stocker une autre valeur. Cependant, si la variable de portée contient un type requêtable, elle peut être interrogée.

Exemple

Dans l'exemple suivant, la clause `let` est utilisée de deux façons différentes :

1. Pour créer un type énumérable qui peut lui-même être interrogé.
2. Pour permettre à la requête d'appeler `ToLower` une seule fois sur la variable de portée `word`. Si vous n'utilisez pas `let`, vous devez appeler `ToLower` dans chaque prédicat de la clause `where`.

C#

```
class LetSample1
{
    static void Main()
    {
        string[] strings =
        {
            "A penny saved is a penny earned.",
            "The early bird catches the worm.",
            "The pen is mightier than the sword."
        };

        // Split the sentence into an array of words
        // and select those whose first letter is a vowel.
        var earlyBirdQuery =
            from sentence in strings
            let words = sentence.Split(' ')
            from word in words
            let w = word.ToLower()
            where w[0] == 'a' || w[0] == 'e'
                || w[0] == 'i' || w[0] == 'o'
                || w[0] == 'u'
            select word;

        // Execute the query.
```

```
        foreach (var v in earlyBirdQuery)
        {
            Console.WriteLine("\"{0}\" starts with a vowel", v);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    "A" starts with a vowel
    "is" starts with a vowel
    "a" starts with a vowel
    "earned." starts with a vowel
    "early" starts with a vowel
    "is" starts with a vowel
*/
```

Voir aussi

- Référence C#
- Mots clés de requête (LINQ)
- LINQ en C#
- LINQ (Language-Integrated Query)
- Gérer des exceptions dans des expressions de requête

ascending (Référence C#)

Article • 07/04/2023

Le mot clé contextuel `ascending` est utilisé dans la clause orderby dans les expressions de requête afin de spécifier que l'ordre de tri est du plus petit au plus grand. `ascending` étant l'ordre de tri par défaut, vous n'avez pas à le spécifier.

Exemple

L'exemple suivant illustre l'utilisation de `ascending` dans une clause orderby.

C#

```
IEnumerable<string> sortAscendingQuery =
    from vegetable in vegetables
    orderby vegetable ascending
    select vegetable;
```

Voir aussi

- Référence C#
- LINQ en C#
- descending

descending (Référence C#)

Article • 07/04/2023

Le mot clé contextuel `descending` est utilisé dans la clause orderby dans les expressions de requête afin de spécifier que l'ordre de tri est du plus grand au plus petit.

Exemple

L'exemple suivant illustre l'utilisation de `descending` dans une clause orderby.

C#

```
IEnumerable<string> sortDescendingQuery =
    from vegetable in vegetables
    orderby vegetable descending
    select vegetable;
```

Voir aussi

- Référence C#
- LINQ en C#
- ascending

on (Référence C#)

Article • 07/04/2023

Le mot clé contextuel `on` est utilisée dans la clause join d'une expression de requête pour spécifier la condition de jointure.

Exemple

L'exemple suivant illustre l'utilisation de `on` dans une clause `join`.

C#

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name };
```

Voir aussi

- Référence C#
- LINQ (Language-Integrated Query)

equals (Référence C#)

Article • 07/04/2023

Le mot clé contextuel `equals` est utilisé dans une clause `join` dans une expression de requête pour comparer les éléments de deux séquences. Pour plus d'informations, consultez join, clause.

Exemple

L'exemple suivant illustre l'utilisation du mot clé `equals` dans une clause `join`.

C#

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name };
```

Voir aussi

- LINQ (Language-Integrated Query)

by (Référence C#)

Article • 07/04/2023

Le mot clé contextuel `by` est utilisé dans la clause `group` dans une expression de requête pour spécifier la façon dont les éléments retournés doivent être groupés. Pour plus d'informations, consultez group, clause.

Exemple

L'exemple suivant illustre l'utilisation du mot clé contextuel `by` dans une clause `group` pour spécifier que les étudiants doivent être groupés en fonction de la première lettre de leur nom.

C#

```
var query = from student in students
            group student by student.LastName[0];
```

Voir aussi

- LINQ en C#

in (Référence C#)

Article • 07/04/2023

Le mot clé `in` est utilisé dans les contextes suivants :

- paramètre de type générique dans les interfaces et les délégués génériques.
- Comme modificateur de paramètre, qui vous permet de passer un argument à une méthode par référence plutôt que par valeur.
- Instructions foreach.
- Clauses from dans les expressions de requête LINQ.
- Clauses join dans les expressions de requête LINQ.

Voir aussi

- Mots clés C#
- Référence C#

Opérateurs et expressions C# (référence C#)

Article • 25/02/2023

C# fournit un certain nombre d'opérateurs. Beaucoup d'entre eux sont pris en charge par les types intégrés et vous permettent d'effectuer des opérations de base avec des valeurs de ces types. Ces opérateurs incluent les groupes suivants :

- Opérateurs arithmétiques qui effectuent des opérations arithmétiques avec des opérandes numériques
- Opérateurs de comparaison qui comparent des opérandes numériques
- Opérateurs logiques booléens qui effectuent des opérations logiques avec des opérandes bool
- Opérateurs de bits et de décalage qui effectuent des opérations logiques de décalage ou au niveau du bit avec des opérandes des types intégraux
- Opérateurs d'égalité qui vérifient si leurs opérandes sont égaux ou non

En règle générale, vous pouvez surcharger ces opérateurs, c'est-à-dire spécifier le comportement de l'opérateur pour les opérandes d'un type défini par l'utilisateur.

Les expressions C# les plus simples sont les littéraux (par exemple, les nombres entiers et réels) et les noms des variables. Vous pouvez les combiner en expressions complexes à l'aide d'opérateurs. La priorité et l'associativité de l'opérateur déterminent l'ordre dans lequel les opérations sont exécutées dans une expressions. Vous pouvez utiliser des parenthèses pour changer l'ordre d'évaluation imposé par la priorité et l'associativité de l'opérateur.

Dans le code suivant, des exemples d'expressions se trouvent à droite des affectations :

C#

```
int a, b, c;
a = 7;
b = a;
c = b++;
b = a + b * c;
c = a >= 100 ? b : c / 10;
a = (int)Math.Sqrt(b * b + c * c);

string s = "String literal";
char l = s[s.Length - 1];

var numbers = new List<int>(new[] { 1, 2, 3 });
b = numbers.FindLast(n => n > 1);
```

En règle générale, une expression produit un résultat et peut être incluse dans une autre expression. Un appel de méthode void est un exemple d'expression qui ne produit pas de résultat. Elle ne peut être utilisée qu'en tant qu'instruction, comme le montre l'exemple suivant :

C#

```
Console.WriteLine("Hello, world!");
```

Voici d'autres types d'expressions que C# fournit :

- Expressions de chaîne interpolée qui fournissent une syntaxe pratique pour créer des chaînes mises en forme :

  C#

  ```
  var r = 2.3;
  var message = $"The area of a circle with radius {r} is {Math.PI * r * r:F3}.";
  Console.WriteLine(message);
  // Output:
  // The area of a circle with radius 2.3 is 16.619.
  ```

- Expressions lambda qui vous permettent de créer des fonctions anonymes :

  C#

  ```
  int[] numbers = { 2, 3, 4, 5 };
  var maximumSquare = numbers.Max(x => x * x);
  Console.WriteLine(maximumSquare);
  // Output:
  // 25
  ```

- Expressions de requête qui vous permettent d'utiliser les fonctionnalités de requête directement en C# :

  C#

  ```
  var scores = new[] { 90, 97, 78, 68, 85 };
  IEnumerable<int> highScoresQuery =
      from score in scores
      where score > 80
      orderby score descending
      select score;
  Console.WriteLine(string.Join(" ", highScoresQuery));
  ```

```
// Output:
// 97 90 85
```

Vous pouvez utiliser une définition de corps d'expression pour fournir une définition concise pour une méthode, un constructeur, une propriété, un indexeur ou un finaliseur.

Priorité des opérateurs

Dans une expression avec plusieurs opérateurs, les opérateurs avec une priorité plus élevée sont évalués avant les opérateurs avec une priorité moins élevée. Dans l'exemple suivant, la multiplication est effectuée en premier, car elle a une priorité plus élevée que l'addition :

C#

```
var a = 2 + 2 * 2;
Console.WriteLine(a); //  output: 6
```

Utilisez des parenthèses pour changer l'ordre d'évaluation imposé par la précédence des opérateurs :

C#

```
var a = (2 + 2) * 2;
Console.WriteLine(a); //  output: 8
```

La table suivante répertorie les opérateurs C# de la priorité la plus élevée à la plus basse. Les opérateurs dans chaque ligne sont de priorité égale.

Opérateurs	**Catégorie ou nom**
x.y, f(x), a[i], x?.y, x?[y], x++, x--, x!, new, typeof, checked, unchecked, default, nameof, delegate, sizeof, stackalloc, x->y	Principal
+x, -x, !x, ~x, ++x, --x, ^x, (T)x, await, &x, *x, true and false	Unaire
x..y	Plage
switch, avec	expressions `switch` et `with`
x * y, x / y, x % y	Multiplicatif
x + y, x – y	Additive

Opérateurs	Catégorie ou nom
x << y, x >> y	Shift
x < y, x > y, x <= y, x >= y, is, as	Relations et test de type
x == y, x != y	Égalité
`x & y`	Logique booléen AND ou logique bitwise AND
`x ^ y`	Logique booléen XOR ou logique bitwise XOR
`x | y`	Logique booléen OR ou logique bitwise OR
x && y	AND conditionnel
x || y	OR conditionnel
x ?? y	Opérateur de fusion de Null
c ? t : f	Opérateur conditionnel
x = y, x += y, x -= y, x *= y, x /= y, x %= y, x &= y, x |= y, x ^= y, x <<= y, x >>= y, x ??= y, =>	Affectation et déclaration lambda

Associativité des opérateurs

Quand les opérateurs sont de même priorité, l'associativité des opérateurs détermine l'ordre dans lequel les opérations sont effectuées :

- Les opérateurs *associatifs sur leur gauche* sont évalués dans l'ordre, de gauche à droite. À l'exception des opérateurs d'assignation et de l'opérateur de fusion Null, tous les opérateurs binaires sont associatifs sur leur gauche. Par exemple, `a + b - c` est évalué comme étant `(a + b) - c`.
- Les opérateurs *associatifs sur leur droite* sont évalués dans l'ordre, de droite à gauche. Les opérateurs d'assignation, l'opérateur de fusion Null, lambda et l'opérateur conditionnel?: sont associatifs sur leur droite. Par exemple, `x = y = z` est évalué comme étant `x = (y = z)`.

Utilisez des parenthèses pour changer l'ordre d'évaluation imposé par l'associativité des opérateurs :

```
C#
```

```
int a = 13 / 5 / 2;
int b = 13 / (5 / 2);
Console.WriteLine($"a = {a}, b = {b}");  // output: a = 1, b = 6
```

Évaluation des opérandes

Sans rapport avec la priorité et l'associativité des opérateurs, les opérandes dans une expression sont évalués de gauche à droite. Les exemples suivants illustrent l'ordre dans lequel les opérateurs et les opérandes sont évalués :

Expression	Ordre d'évaluation
`a + b`	a, b, +
`a + b * c`	a, b, c, *, +
`a / b + c * d`	a, b, /, c, d, *, +
`a / (b + c) * d`	a, b, c, +, /, d, *

En règle générale, tous les opérandes d'opérateur sont évalués. Toutefois, certains opérateurs évaluent les opérandes de manière conditionnelle. Autrement dit, la valeur de l'opérande le plus à gauche d'un opérateur de ce type définit si les autres opérandes doivent être évalués. Ces opérateurs sont les opérateurs conditionnels logiques AND (&&) et OR (||), l'opérateur de fusion Null ?? et ??=, les opérateurs conditionnels Null?. et ?[], et l'opérateur conditionnel ?:. Pour plus d'informations, consultez la description de chaque opérateur.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Expressions
- Opérateurs

Voir aussi

- Informations de référence sur C#
- Surcharge d'opérateur
- Arborescences d'expressions

Opérateurs arithmétiques (référence C#)

Article • 10/05/2023

Les opérateurs suivants effectuent des opérations arithmétiques avec des opérandes de types numériques :

- Opérateurs unaires ++ (incrément), -- (décrément), + (plus) et - (moins)
- Opérateurs binaires * (multiplication), / (division), % (reste), + (addition) et - (soustraction)

Ces opérateurs sont pris en charge par tous les types numériques intégraux et à virgule flottante.

Dans le cas des types intégraux, ces opérateurs (à l'exception des opérateurs `++` et `--`) sont définis pour les types `int`, `uint`, `long` et `ulong`. Lorsque les opérandes sont d'autres types intégraux (`sbyte`, `byte`, `short`, `ushort` ou `char`), leurs valeurs sont converties en valeurs de type `int`, qui est également le type de résultat d'une opération. Lorsque des opérandes sont de types intégraux ou à virgule flottante différents, leurs valeurs sont converties en type contenant le plus proche, s'il existe un tel type. Pour plus d'informations, consultez la section Promotions numériques de la spécification du langage C#. Les opérateurs `++` et `--` sont définis pour tous les types numériques intégraux et à virgule flottante, ainsi que pour le type char . Le type de résultat d'une expression d'affectation composée est le type de l'opérande de gauche.

Opérateur d'incrémentation ++

L'opérateur d'incrémentation unaire `++` incrémente son opérande de 1. L'opérande doit être une variable, un accès propriété ou un accès indexeur.

L'opérateur d'incrémentation est pris en charge sous deux formes : l'opérateur d'incrémentation suffixé, `x++`, et l'opérateur d'incrémentation préfixé, `++x`.

Opérateur d'incrément suffixé

Le résultat de `x++` est la valeur de `x` *avant* l'opération, comme le montre l'exemple suivant :

C#

```
int i = 3;
Console.WriteLine(i);   // output: 3
```

```
Console.WriteLine(i++); // output: 3
Console.WriteLine(i);   // output: 4
```

Opérateur d'incrémentation préfixé

Le résultat de `++x` est la valeur de `x` *après* l'opération, comme le montre l'exemple suivant :

C#

```
double a = 1.5;
Console.WriteLine(a);   // output: 1.5
Console.WriteLine(++a); // output: 2.5
Console.WriteLine(a);   // output: 2.5
```

Opérateur de décrémentation --

L'opérateur de décrémentation unaire `--` décrémente son opérande de 1. L'opérande doit être une variable, un accès propriété ou un accès indexeur.

L'opérateur de décrémentation est pris en charge sous deux formes : l'opérateur de décrémentation suffixé, `x--`, et l'opérateur de décrémentation préfixé, `--x`.

Opérateur de décrémentation suffixé

Le résultat de `x--` est la valeur de `x` *avant* l'opération, comme le montre l'exemple suivant :

C#

```
int i = 3;
Console.WriteLine(i);   // output: 3
Console.WriteLine(i--); // output: 3
Console.WriteLine(i);   // output: 2
```

Opérateur de décrémentation préfixé

Le résultat de `--x` est la valeur de `x` *après* l'opération, comme le montre l'exemple suivant :

C#

```
double a = 1.5;
Console.WriteLine(a);   // output: 1.5
Console.WriteLine(--a); // output: 0.5
Console.WriteLine(a);   // output: 0.5
```

Opérateurs plus et moins unaires

L'opérateur unaire `+` retourne la valeur de son opérande. L'opérateur unaire `-` calcule la négation numérique de son opérande.

C#

```
Console.WriteLine(+4);     // output: 4

Console.WriteLine(-4);     // output: -4
Console.WriteLine(-(-4));  // output: 4

uint a = 5;
var b = -a;
Console.WriteLine(b);            // output: -5
Console.WriteLine(b.GetType());  // output: System.Int64

Console.WriteLine(-double.NaN);  // output: NaN
```

Le type ulong ne prend pas en charge l'opérateur unaire `-`.

Opérateur de multiplication *

L'opérateur de multiplication `*` calcule le produit de ses opérandes :

C#

```
Console.WriteLine(5 * 2);         // output: 10
Console.WriteLine(0.5 * 2.5);     // output: 1.25
Console.WriteLine(0.1m * 23.4m);  // output: 2.34
```

L'opérateur unaire `*` est l'opérateur d'indirection de pointeur.

Opérateur de division /

L'opérateur de division `/` divise son opérande de partie gauche par son opérande de partie droite.

Division d'entier

Pour les opérandes des types entiers, le résultat de l'opérateur `/` est de type entier et égal au quotient de deux opérandes arrondis vers le zéro :

C#

```
Console.WriteLine(13 / 5);    // output: 2
Console.WriteLine(-13 / 5);   // output: -2
Console.WriteLine(13 / -5);   // output: -2
Console.WriteLine(-13 / -5);  // output: 2
```

Pour obtenir le quotient de deux opérandes comme un nombre à virgule flottante, utilisez le type `float`, `double` ou `decimal` :

C#

```
Console.WriteLine(13 / 5.0);       // output: 2.6

int a = 13;
int b = 5;
Console.WriteLine((double)a / b);  // output: 2.6
```

Division à virgule flottante

Pour les types `float`, `double` et `decimal`, le résultat de l'opérateur `/` est le quotient de deux opérandes :

C#

```
Console.WriteLine(16.8f / 4.1f);   // output: 4.097561
Console.WriteLine(16.8d / 4.1d);   // output: 4.09756097560976
Console.WriteLine(16.8m / 4.1m);   // output: 4.0975609756097560975609756098
```

Si l'un des opérandes est `decimal`, un autre opérande ne peut être `float` ni `double`, car ni `float` ni `double` ne sont implicitement convertibles en `decimal`. Vous devez convertir explicitement l'opérande `float` ou `double` en type `decimal`. Pour plus d'informations sur les conversions entre des types numériques, consultez Conversions numériques intégrées.

Opérateur de reste %

L'opérateur restant `%` calcule le reste après la division de son opérande de partie gauche par son opérande de partie droite.

Reste entier

Pour les opérandes des types entiers, le résultat de `a % b` est la valeur produite par `a - (a / b) * b`. Le signe du reste non zéro est le même que celui du signe de l'opérande de partie gauche, comme l'indique l'exemple suivant :

```
C#
```

```
Console.WriteLine(5 % 4);   // output: 1
Console.WriteLine(5 % -4);  // output: 1
Console.WriteLine(-5 % 4);  // output: -1
Console.WriteLine(-5 % -4); // output: -1
```

Utilisez la méthode Math.DivRem pour calculer à la fois la division entière et les résultats du reste.

Reste à virgule flottante

En ce qui concerne les opérandes `float` et `double`, le résultat de `x % y` pour le `x` et le `y` finis est la valeur `z` afin que

- Le signe de `z`, s'il est différent de zéro, soit identique au signe de `x`.
- La valeur absolue de `z` soit la valeur produite par `|x| - n * |y|`, où `n` représente le plus grand entier possible inférieur ou égal à `|x| / |y|`, et où `|x|` et `|y|` sont les valeurs absolues de `x` et `y`, respectivement.

> ⓘ **Notes**
>
> Cette méthode de calcul du reste est analogue à celle utilisée pour les opérandes entiers. Toutefois, elle ne suit pas la spécification IEEE 754. Si l'opération de reste doit être conforme à la spécification IEEE 754, utilisez la méthode **Math.IEEERemainder**.

Pour plus d'informations sur le comportement de l'opérateur `%` avec des opérandes non finis, consultez la section Opérateur de reste dans la Spécification du langage C#.

Pour les opérandes `decimal`, l'opérateur de reste `%` équivaut à l'opérateur de reste de type System.Decimal.

L'exemple suivant illustre le comportement de l'opérateur de reste pour les opérandes à virgule flottante :

```
C#
```

```
Console.WriteLine(-5.2f % 2.0f); // output: -1.2
Console.WriteLine(5.9 % 3.1);    // output: 2.8
Console.WriteLine(5.9m % 3.1m);  // output: 2.8
```

Opérateur d'addition +

L'opérateur d'addition `+` calcule la somme de ses opérandes :

```
C#
```

```
Console.WriteLine(5 + 4);       // output: 9
Console.WriteLine(5 + 4.3);     // output: 9.3
Console.WriteLine(5.1m + 4.2m); // output: 9.3
```

Vous pouvez également utiliser l'opérateur `+` pour la concaténation de chaînes et la combinaison de délégués. Pour plus d'informations, consultez l'article + et +=, opérateurs.

Opérateur de soustraction -

L'opérateur de soustraction `-` soustrait son opérande de partie droite de son opérande de partie gauche :

```
C#
```

```
Console.WriteLine(47 - 3);      // output: 44
Console.WriteLine(5 - 4.3);     // output: 0.7
Console.WriteLine(7.5m - 2.3m); // output: 5.2
```

Vous pouvez également utiliser l'opérateur `-` pour la suppression de délégués. Pour plus d'informations, consultez l'article - et -=, opérateurs.

Assignation composée

Pour un opérateur binaire `op`, une expression d'assignation composée du formulaire

```
C#
```

```
x op= y
```

équivaut à :

```
C#

x = x op y
```

sauf que `x` n'est évalué qu'une seule fois.

L'exemple suivant illustre l'utilisation de l'assignation composée avec des opérateurs arithmétiques :

```
C#

int a = 5;
a += 9;
Console.WriteLine(a);  // output: 14

a -= 4;
Console.WriteLine(a);  // output: 10

a *= 2;
Console.WriteLine(a);  // output: 20

a /= 4;
Console.WriteLine(a);  // output: 5

a %= 3;
Console.WriteLine(a);  // output: 2
```

En raison des promotions numériques, le résultat de l'opération `op` risque de ne pas être implicitement convertible en type `T` de `x`. Dans ce cas, si `op` est un opérateur prédéfini et que le résultat de l'opération est explicitement convertible en type `T` de `x`, une expression d'assignation composée de la forme `x op= y` équivaut à `x = (T)(x op y)`, sauf que `x` n'est évalué qu'une seule fois. L'exemple suivant illustre ce comportement :

```
C#

byte a = 200;
byte b = 100;

var c = a + b;
Console.WriteLine(c.GetType());  // output: System.Int32
Console.WriteLine(c);  // output: 300
```

```
a += b;
Console.WriteLine(a);  // output: 44
```

Dans l'exemple précédent, la valeur `44` est le résultat de la conversion de la valeur `300` en type `byte`.

> ⓘ **Notes**
>
> Dans le **contexte de vérification de dépassement de capacité vérifié**, l'exemple précédent lève un **OverflowException**. Pour plus d'informations, consultez la section **Dépassement arithmétique entier**.

Vous utilisez également les opérateurs `+=` et `-=` pour vous abonner d'un événements puis vous désabonner, respectivement. Pour plus d'informations, consultez Guide pratique pour s'abonner et annuler l'abonnement à des événements.

Priorité des opérateurs et associativité

La liste suivante présente les opérateurs arithmétiques de la priorité la plus élevée à la plus basse :

- Opérateurs suffixés d'incrémentation `x++` et de décrémentation `x--`
- Opérateurs préfixés d'incrémentation `++x` et de décrémentation `--x` et opérateurs unaires `+` et `-`
- Opérateurs multiplicatifs `*`, `/` et `%`
- Opérateurs additifs `+` et `-`

Les opérateurs arithmétiques binaires sont associatifs sur leur gauche. Autrement dit, les opérateurs de même niveau de priorité sont évalués de gauche à droite.

Utilisez des parenthèses, `()`, pour modifier l'ordre d'évaluation imposé par la priorité et l'associativité de l'opérateur.

C#

```
Console.WriteLine(2 + 2 * 2);   // output: 6
Console.WriteLine((2 + 2) * 2); // output: 8

Console.WriteLine(9 / 5 / 2);   // output: 0
Console.WriteLine(9 / (5 / 2)); // output: 4
```

Pour obtenir la liste complète des opérateurs C# classés par niveau de priorité, consultez la section Priorité des opérateurs de l'article Opérateurs C#.

Débordement arithmétique et division par zéro

Quand le résultat d'une opération arithmétique n'est pas compris dans la plage de valeurs finies possibles du type numérique impliqué, le comportement d'un opérateur arithmétique varie selon le type de ses opérandes.

Débordement arithmétique entier

La division d'un entier par zéro lève toujours une exception DivideByZeroException.

Si un dépassement arithmétique entier se produit, un contexte de vérification de débordement, qui peut être vérifié ou non vérifié, contrôle le comportement qui en résulte :

- Dans un contexte vérifié, si le débordement se produit dans une expression constante, une erreur de compilation se produit. Sinon, un OverflowException est levé quand l'opération est effectuée au moment de l'exécution.
- Dans un contexte non vérifié, le résultat est tronqué en supprimant tous les bits de poids fort qui ne tiennent pas dans le type destinataire.

Avec les instructions vérifiées et non vérifiées, vous pouvez utiliser les opérateurs `checked` et `unchecked` pour contrôler le contexte de vérification de dépassement, dans lequel une expression est évaluée :

C#

```
int a = int.MaxValue;
int b = 3;

Console.WriteLine(unchecked(a + b));  // output: -2147483646
try
{
    int d = checked(a + b);
}
catch(OverflowException)
{
    Console.WriteLine($"Overflow occurred when adding {a} to {b}.");
}
```

Par défaut, les opérations arithmétiques se produisent dans un contexte *unchecked*.

Débordement arithmétique à virgule flottante

Les opérations arithmétiques avec les types `float` et `double` ne lèvent jamais d'exceptions. Le résultat des opérations arithmétiques avec ces types peut être une des valeurs spéciales qui représentent l'infini et non un nombre :

C#

```
double a = 1.0 / 0.0;
Console.WriteLine(a);                    // output: Infinity
Console.WriteLine(double.IsInfinity(a)); // output: True

Console.WriteLine(double.MaxValue + double.MaxValue); // output: Infinity

double b = 0.0 / 0.0;
Console.WriteLine(b);                 // output: NaN
Console.WriteLine(double.IsNaN(b));  // output: True
```

Pour les opérandes du type `decimal`, le dépassement arithmétique lève toujours un OverflowException. La division d'un nombre par zéro lève toujours une exception DivideByZeroException.

Erreurs d'arrondi

En raison des limitations générales de la représentation à virgule flottante des nombres réels et de l'arithmétique à virgule flottante, des erreurs d'arrondi peuvent se produire dans les calculs avec les types à virgule flottante. Autrement dit, le résultat d'une expression peut différer du résultat mathématique attendu. L'exemple suivant illustre plusieurs cas de ce genre :

C#

```
Console.WriteLine(.41f % .2f); // output: 0.00999999

double a = 0.1;
double b = 3 * a;
Console.WriteLine(b == 0.3);   // output: False
Console.WriteLine(b - 0.3);    // output: 5.55111512312578E-17

decimal c = 1 / 3.0m;
decimal d = 3 * c;
Console.WriteLine(d == 1.0m);  // output: False
Console.WriteLine(d);          // output: 0.99999999999999999999999999
```

Pour plus d'informations, consultez les remarques dans les pages de référence System.Double, System.Single ou System.Decimal.

Capacité de surcharge de l'opérateur

Un type défini par l'utilisateur peut surcharger les opérateurs arithmétiques unaires (`++`, `--`, `+` et `-`) et binaires (`*`, `/`, `%`, `+` et `-`). Quand un opérateur binaire est surchargé, l'opérateur d'assignation composée correspondant est aussi implicitement surchargé. Un type défini par l'utilisateur ne peut pas surcharger explicitement un opérateur d'assignation composée.

Opérateurs vérifiés définis par l'utilisateur

À compter de C# 11, lorsque vous surchargez un opérateur arithmétique, vous pouvez utiliser le mot clé `checked` pour définir la version *vérifiée* de cet opérateur. L'exemple suivant montre comment effectuer cette opération :

C#

```
public record struct Point(int X, int Y)
{
    public static Point operator checked +(Point left, Point right)
    {
        checked
        {
            return new Point(left.X + right.X, left.Y + right.Y);
        }
    }

    public static Point operator +(Point left, Point right)
    {
        return new Point(left.X + right.X, left.Y + right.Y);
    }
}
```

Lorsque vous définissez un opérateur vérifié, vous devez également définir l'opérateur correspondant sans le modificateur `checked`. L'opérateur vérifié est appelé dans un contexte vérifié; l'opérateur sans le `checked` modificateur est appelé dans un contexte non vérifié. Si vous fournissez uniquement l'opérateur sans le modificateur `checked`, il est appelé à la fois dans un contexte `checked` et `unchecked`.

Lorsque vous définissez les deux versions d'un opérateur, il est attendu que leur comportement diffère uniquement lorsque le résultat d'une opération est trop grand pour être représenté dans le type de résultat comme suit :

- Un opérateur vérifié lève une OverflowException.

- Un opérateur sans modificateur `checked` retourne une instance représentant un résultat *tronqué*.

Pour plus d'informations sur la différence de comportement des opérateurs arithmétiques intégrés, consultez la section Dépassement arithmétique et division par zéro.

Vous pouvez utiliser le modificateur `checked` uniquement lorsque vous surchargez l'un des opérateurs suivants :

- Opérateurs unaires `++`, `--` et `-`
- Opérateurs binaires `*`, `/`, `+` et `-`
- Opérateurs de conversion explicite

> ⓘ **Notes**
>
> Le contexte de contrôle de dépassement dans le corps d'un opérateur vérifié n'est pas affecté par la présence du modificateur `checked`. Le contexte par défaut est défini par la valeur de l'option de compilateur **CheckForOverflowUnderflow**. Utilisez les **instructions uncheckedet checked** pour spécifier explicitement le contexte de vérification de dépassement de capacité, comme le montre l'exemple au début de cette section.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Opérateurs suffixés d'incrémentation et de décrémentation
- Opérateurs préfixés d'incrémentation et de décrémentation
- Opérateur plus unaire
- Opérateur moins unaire
- Opérateur de multiplication
- Opérateur de division
- Opérateur de reste
- Opérateur d'addition
- Opérateur de soustraction
- Assignation composée
- Opérateurs vérifiés et non vérifiés
- Promotions numériques

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- System.Math
- System.MathF
- Valeurs numériques dans .NET

Opérateurs logiques booléens - AND, OR, NOT, XOR

Article • 07/04/2023

Les opérateurs booléens logiques effectuent des opérations logiques avec des opérandes bool. Les opérateurs incluent la négation logique unaire (`!`), la logique binaire AND (`&`), OR (`|`) et OR exclusif (`^`) et la logique binaire AND (`&&`) et OR (`||`).

- opérateur unaire ! (négation logique) ;
- opérateurs binaires & (AND logique), | (OR logique) et ^ (OR exclusif logique), qui évaluent toujours les deux opérandes ;
- opérateurs binaires && (AND logique conditionnel) et || (OR logique conditionnel), Ces opérateurs n'évaluent l'opérande de partie droite que si nécessaire.

En ce qui concerne les opérandes de type numérique intégral, les opérateurs `&`, `|` et `^` effectuent des opérations logiques binaires. Pour plus d'informations, consultez Opérateurs de bits et de décalage.

L'opérateur de négation logique !

L'opérateur unaire préfixe `!` calcule la négation logique de son opérande. Autrement dit, il produit `true` si l'opérande donne `false` et `false` si l'opérande donne `true` :

C#

```
bool passed = false;
Console.WriteLine(!passed);  // output: True
Console.WriteLine(!true);    // output: False
```

L'opérateur postfix `!` unaire est l'opérateur null-forgiving.

L'opérateur AND logique &

L'opérateur `&` calcule le AND logique de ses opérandes. Le résultat de `x & y` est `true` si `x` et `y` prennent la valeur `true`. Sinon, le résultat est `false`.

L'opérateur `&` évalue les deux opérandes, même si l'opérande de gauche est évalué à `false`. Le résultat doit donc être `false` quelle que soit la valeur de l'opérande de droite.

Dans l'exemple suivant, l'opérande de partie droite de l'opérateur `&` est un appel de méthode, effectué indépendamment de la valeur de l'opérande de partie gauche :

C#

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = false & SecondOperand();
Console.WriteLine(a);
// Output:
// Second operand is evaluated.
// False

bool b = true & SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

L'opérateur AND logique conditionnel `&&` calcule également le AND logique de ses opérandes, mais n'évalue pas l'opérande de partie droite si l'opérande de partie gauche donne la valeur de `false`.

Dans le cas des opérandes de types numériques intégraux, `&` l'opérateur calcule le AND logique binaire de ses opérandes. L'opérateur unaire `&` est l'opérateur address-of.

L'opérateur OR exclusif logique ^

L'opérateur `^` calcule le OR exclusif logique, également appelé XOR logique, de ses opérandes. Le résultat de `x ^ y` est `true` si `x` donne `true` et `y` donne `false`, ou `x` donne `false` et `y` donne `true`. Sinon, le résultat est `false`. Autrement dit, pour les opérandes `bool`, l'opérateur `^` calcule le même résultat que l'opérateur d'inégalité `!=`.

C#

```
Console.WriteLine(true ^ true);    // output: False
Console.WriteLine(true ^ false);   // output: True
Console.WriteLine(false ^ true);   // output: True
Console.WriteLine(false ^ false);  // output: False
```

Dans le cas des opérandes de types numériques intégraux, `^` l'opérateur calcule l'opération logique exclusive OU binaire de ses opérandes.

L'opérateur OU logique |

L'opérateur `|` calcule le OR logique de ses opérandes. Le résultat de `x | y` est `true` si `x` ou `y` prend la valeur `true`. Sinon, le résultat est `false`.

L'opérateur `|` évalue les deux opérandes, même si l'opérande de gauche est évalué à `true`. Le résultat doit donc être `true` quelle que soit la valeur de l'opérande de droite.

Dans l'exemple suivant, l'opérande de partie droite de l'opérateur `|` est un appel de méthode, effectué indépendamment de la valeur de l'opérande de partie gauche :

C#

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = true | SecondOperand();
Console.WriteLine(a);
// Output:
// Second operand is evaluated.
// True

bool b = false | SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

L'opérateur OR logique conditionnel `||` calcule également le OR logique de ses opérandes, mais n'évalue pas l'opérande de partie droite si l'opérande de partie gauche donne la valeur de `true`.

Dans le cas des opérandes de types numériques intégraux, `|` l'opérateur calcule le OR logique binaire de ses opérandes.

Opérateur AND logique conditionnel &&

L'opérateur AND logique conditionnel `&&`, également appelé opérateur AND logique de « court-circuit », calcule le AND logique de ses opérandes. Le résultat de `x && y` est

`true` si `x` et `y` prennent la valeur `true`. Sinon, le résultat est `false`. Si `x` est évalué à `false`, `y` n'est pas évalué.

Dans l'exemple suivant, l'opérande de partie droite de l'opérateur `&&` est un appel de méthode, non effectué si l'opérande de partie gauche donne la valeur de `false` :

C#

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = false && SecondOperand();
Console.WriteLine(a);
// Output:
// False

bool b = true && SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

L'opérateur AND logique `&` calcule également le AND logique de ses opérandes, mais évalue toujours les deux opérandes.

L'opérateur OR logique conditionnel ||

L'opérateur OR logique conditionnel `||`, également appelé opérateur OR logique de « court-circuit », calcule le OR logique de ses opérandes. Le résultat de `x || y` est `true` si `x` ou `y` prend la valeur `true`. Sinon, le résultat est `false`. Si `x` est évalué à `true`, `y` n'est pas évalué.

Dans l'exemple suivant, l'opérande de partie droite de l'opérateur `||` est un appel de méthode, non effectué si l'opérande de partie gauche donne la valeur de `true` :

C#

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = true || SecondOperand();
```

```
Console.WriteLine(a);
// Output:
// True

bool b = false || SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

L'opérateur OR logique `|` calcule également le OR logique de ses opérandes, mais évalue toujours les deux opérandes.

Les opérateurs logiques booléens Nullable

Pour les opérandes `bool?`, les opérateurs &(AND logique) et | (OR logique) prennent en charge la logique à trois valeurs comme suit :

- L'opérateur `&` produit `true` uniquement si les deux opérandes sont évalués à `true`. Si l'une ou l'autre de `x` ou `y` est évaluée à `false`, `x & y` produit `false` (même si un autre opérande est évalué à `null`). Sinon, le résultat de `x & y` est `null`.

- L'opérateur `|` produit `false` uniquement si les deux opérandes sont évalués à `false`. Si l'une ou l'autre de `x` ou `y` est évaluée à `true`, `x | y` produit `true` (même si un autre opérande est évalué à `null`). Sinon, le résultat de `x | y` est `null`.

Le tableau suivant présente cette sémantique :

x	y	x&y	x|y
true	true	true	true
true	false	false	true
true	null	null	true
false	true	false	true
false	false	false	false
false	null	false	null
null	true	null	true

x	y	x&y	x|y
null	false	false	null
null	null	null	null

Le comportement de ces opérateurs diffère du comportement classique des opérateurs avec des types valeur Nullable. En règle générale, un opérateur défini pour les opérandes d'un type valeur peut être également utilisé avec des opérandes du type valeur pouvant accepter la valeur Null correspondante. Un tel opérateur produit `null` si l'un de ses opérandes est `null`. Toutefois, les opérateurs `&` et `|` peuvent produire une valeur non null même si l'un des opérandes est évalué à `null`. Pour plus d'informations sur le comportement des opérateurs avec des types valeur pouvant accepter la valeur Null, voir la section Opérateurs de l'article Utiliser des types pouvant accepter la valeur Null.

Vous pouvez également utiliser les opérateurs `!` et `^` avec les opérandes `bool?`, comme le montre l'exemple suivant :

C#

```
bool? test = null;
Display(!test);         // output: null
Display(test ^ false);  // output: null
Display(test ^ null);   // output: null
Display(true ^ null);   // output: null

void Display(bool? b) => Console.WriteLine(b is null ? "null" :
b.Value.ToString());
```

Les opérateurs logiques conditionnels `&&` et `||` ne prennent pas en charge les opérandes `bool?`.

Assignation composée

Pour un opérateur binaire `op`, une expression d'assignation composée du formulaire

C#

```
x op= y
```

équivaut à :

C#

```
x = x op y
```

sauf que `x` n'est évalué qu'une seule fois.

Les opérateurs `&`, `|` et `^` prennent en charge l'assignation composée, comme le montre l'exemple suivant :

C#

```
bool test = true;
test &= false;
Console.WriteLine(test);  // output: False

test |= true;
Console.WriteLine(test);  // output: True

test ^= false;
Console.WriteLine(test);  // output: True
```

> ⓘ **Notes**
>
> Les opérateurs logiques conditionnels `&&` et `||` ne prennent pas en charge l'assignation composée.

Priorité des opérateurs

La liste suivante présente les opérateurs logiques par ordre de précédence, de la plus élevée à la plus basse :

- Opérateur de négation logique `!`
- L'opérateur AND logique `&`
- Opérateur OR exclusif logique `^`
- Opérateur OR logique `|`
- Opérateur AND logique conditionnel `&&`
- Opérateur OR logique conditionnel `||`

Utilisez des parenthèses, `()`, pour modifier l'ordre d'évaluation imposé par la précédence des opérateurs :

C#

```
Console.WriteLine(true | true & false);   // output: True
Console.WriteLine((true | true) & false); // output: False

bool Operand(string name, bool value)
{
    Console.WriteLine($"Operand {name} is evaluated.");
    return value;
}

var byDefaultPrecedence = Operand("A", true) || Operand("B", true) &&
Operand("C", false);
Console.WriteLine(byDefaultPrecedence);
// Output:
// Operand A is evaluated.
// True

var changedOrder = (Operand("A", true) || Operand("B", true)) &&
Operand("C", false);
Console.WriteLine(changedOrder);
// Output:
// Operand A is evaluated.
// Operand C is evaluated.
// False
```

Pour obtenir la liste complète des opérateurs C# classés par niveau de priorité, consultez la section Priorité des opérateurs de l'article Opérateurs C#.

Capacité de surcharge de l'opérateur

Un type défini par l'utilisateur peut surcharger les opérateurs `!`, `&`, `|` et `^`. Quand un opérateur binaire est surchargé, l'opérateur d'assignation composée correspondant est aussi implicitement surchargé. Un type défini par l'utilisateur ne peut pas surcharger explicitement un opérateur d'assignation composée.

Un type défini par l'utilisateur ne peut pas surcharger les opérateurs logiques conditionnels `&&` et `||`. Toutefois, si un type défini par l'utilisateur surcharge les opérateurs true et false et l'opérateur `&` ou `|` d'une certaine manière, l'opération `&&` ou `||` peut être évaluée respectivement pour les opérandes de ce type. Pour plus d'informations, consultez la section Opérateurs logiques conditionnels définis par l'utilisateur de la spécification du langage C#.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Opérateur de négation logique
- Opérateurs logiques
- Opérateurs logiques conditionnels
- Assignation composée

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- Opérateurs de bits et de décalage

Opérateurs au niveau du bit et opérateurs de décalage (référence C#)

Article • 03/04/2023

Les opérateurs au niveau du bit et de décalage incluent le complément de bits unaire, les décalages binaires vers la gauche et la droite, le décalage vers la droite non signé, et les opérateurs logiques binaires AND, OR et OR exclusif. Ces opérandes prennent des opérandes des types numériques intégraux ou du type char.

- Opérateur unaire ~ (complément de bits)
- Opérateurs binaires << (décalage vers la gauche), >> (décalage vers la droite) et >>> (décalage vers la droite non signé)
- Opérateurs binaires & (AND logique), | (OR logique) et ^ (OR exclusif logique)

Ces opérateurs sont définis pour les types `int`, `uint`, `long` et `ulong`. Lorsque les deux opérandes sont d'autres types intégraux (`sbyte`, `byte`, `short`, `ushort` ou `char`), leurs valeurs sont converties en valeurs de type `int`, qui est également le type de résultat d'une opération. Lorsque les opérandes sont de différents types intégraux, leurs valeurs sont converties vers le type intégral le plus proche. Pour plus d'informations, consultez la section Promotions numériques de la spécification du langage C#. Les opérateurs composés (tels que `>>=`) ne convertissent pas leurs arguments en `int` et n'ont pas le type de résultat `int`.

Les opérateurs `&`, `|` et `^` sont également définis pour les opérandes du type `bool`. Pour plus d'informations, consultez Opérateurs logiques booléens.

Les opérations de décalage et au niveau du bit ne provoquent jamais de dépassements de capacité et donnent les mêmes résultats dans des contextes checked et unchecked.

Opérateur de complément de bits ~

L'opérateur `~` produit un complément de bits de son opérande en inversant chaque bit :

C#

```
uint a = 0b_0000_1111_0000_1111_0000_1111_0000_1100;
uint b = ~a;
Console.WriteLine(Convert.ToString(b, toBase: 2));
// Output:
// 11110000111100001111000011110011
```

Vous pouvez également utiliser le symbole `~` pour déclarer des finaliseurs. Pour plus d'informations, consultez Finaliseurs.

Opérateur de décalage gauche <<

L'opérateur `<<` décale son opérande de partie gauche vers le nombre de bits spécifié par son opérande de partie droite. Pour obtenir des informations sur la façon dont l'opérande droit définit la valeur de décalage, consultez la section Valeur de décalage des opérateurs de décalage.

L'opération de décalage gauche supprime les bits d'ordre supérieur qui sont en dehors de la plage du type de résultat, et définit les positions de bits vides d'ordre inférieur sur zéro, comme le montre l'exemple suivant :

C#

```
uint x = 0b_1100_1001_0000_0000_0000_0000_0001_0001;
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2)}");

uint y = x << 4;
Console.WriteLine($"After:  {Convert.ToString(y, toBase: 2)}");
// Output:
// Before: 11001001000000000000000000010001
// After:  10010000000000000000000100010000
```

Étant donné que les opérateurs de décalage sont définis uniquement pour les types `int`, `uint`, `long` et `ulong`, le résultat d'une opération contient toujours au moins 32 bits. Si l'opérande de partie gauche est d'un autre type intégral (`sbyte`, `byte`, `short`, `ushort` ou `char`), sa valeur est convertie en type `int`, comme l'indique l'exemple suivant :

C#

```
byte a = 0b_1111_0001;

var b = a << 8;
Console.WriteLine(b.GetType());
Console.WriteLine($"Shifted byte: {Convert.ToString(b, toBase: 2)}");
// Output:
// System.Int32
// Shifted byte: 1111000100000000
```

Opérateur de décalage vers la droite >>

L'opérateur `>>` décale son opérande de partie gauche vers le nombre de bits défini par son opérande de partie droite. Pour obtenir des informations sur la façon dont l'opérande droit définit la valeur de décalage, consultez la section Valeur de décalage des opérateurs de décalage.

L'opération de décalage vers la droite ignore les bits d'ordre inférieur, comme le montre l'exemple suivant :

C#

```
uint x = 0b_1001;
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2), 4}");

uint y = x >> 2;
Console.WriteLine($"After:  {Convert.ToString(y, toBase: 2).PadLeft(4, '0'),
4}");
// Output:
// Before: 1001
// After:  0010
```

Les positions de bits vides d'ordre supérieur sont définies en fonction du type de l'opérande de partie gauche comme suit :

- Si l'opérande gauche est de type `int` ou `long`, l'opérateur de décalage vers la droite effectue un décalage *arithmétique* : la valeur du bit le plus significatif (le bit de signe) de l'opérande gauche est propagée vers les positions des bits vides de poids fort. Autrement dit, les positions de bits vides d'ordre supérieur sont définies sur zéro si l'opérande de partir gauche n'est pas négatif et sur un s'il est négatif.

  C#

  ```
  int a = int.MinValue;
  Console.WriteLine($"Before: {Convert.ToString(a, toBase: 2)}");

  int b = a >> 3;
  Console.WriteLine($"After:  {Convert.ToString(b, toBase: 2)}");
  // Output:
  // Before: 10000000000000000000000000000000
  // After:  11110000000000000000000000000000
  ```

- Si l'opérande gauche est de type `uint` ou `ulong`, l'opérateur de décalage vers la droite effectue un décalage *logique* : les positions des bits vides de poids fort sont toujours définies sur zéro.

  C#

```
uint c = 0b_1000_0000_0000_0000_0000_0000_0000_0000;
Console.WriteLine($"Before: {Convert.ToString(c, toBase: 2), 32}");

uint d = c >> 3;
Console.WriteLine($"After:  {Convert.ToString(d, toBase: 2).PadLeft(32,
'0'), 32}");
// Output:
// Before: 10000000000000000000000000000000
// After:  00010000000000000000000000000000
```

> ⓘ Notes
>
> Utilisez l'**opérateur de décalage vers la droite non signé** pour effectuer un décalage *logique* sur les opérandes de types entiers signés. Cela est préférable au cast d'un opérande gauche en type non signé, puis au cast du résultat d'une opération de décalage en type signé.

Opérateur de décalage vers la droite non signé >>>

Disponible en C# 11 et versions ultérieures, l'opérateur `>>>` décale son opérande gauche vers la droite du nombre de bits défini par son opérande droit. Pour obtenir des informations sur la façon dont l'opérande droit définit la valeur de décalage, consultez la section Valeur de décalage des opérateurs de décalage.

L'opérateur `>>>` effectue toujours un décalage *logique*. Autrement dit, les positions de bits vides de poids fort sont toujours définies sur zéro, quel que soit le type de l'opérande gauche. L'opérateur `>>` effectue un décalage *arithmétique* (autrement dit, la valeur du bit le plus significatif est propagée aux positions de bits vides de poids fort) si l'opérande gauche est d'un type signé. L'exemple suivant illustre la différence entre les opérateurs `>>` et `>>>` pour un opérande gauche négatif :

C#

```
int x = -8;
Console.WriteLine($"Before:    {x,11}, hex: {x,8:x}, binary:
{Convert.ToString(x, toBase: 2), 32}");

int y = x >> 2;
Console.WriteLine($"After  >>: {y,11}, hex: {y,8:x}, binary:
{Convert.ToString(y, toBase: 2), 32}");

int z = x >>> 2;
```

```
Console.WriteLine($"After >>>: {z,11}, hex: {z,8:x}, binary:
{Convert.ToString(z, toBase: 2).PadLeft(32, '0'), 32}");
// Output:
// Before:             -8, hex: fffffff8, binary:
11111111111111111111111111111000
// After  >>:          -2, hex: fffffffe, binary:
11111111111111111111111111111110
// After >>>:  1073741822, hex: 3ffffffe, binary:
00111111111111111111111111111110
```

L'opérateur AND logique &

L'opérateur `&` calcule le AND logique au niveau du bit de ses opérandes intégraux :

C#

```
uint a = 0b_1111_1000;
uint b = 0b_1001_1101;
uint c = a & b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 10011000
```

Pour les opérandes `bool`, l'opérateur `&` calcule le AND logique de ses opérandes. L'opérateur unaire `&` est l'opérateur address-of.

L'opérateur OR exclusif logique ^

L'opérateur `^` calcule le OR exclusif logique au niveau du bit, également appelé XOR logique au niveau du bit, de ses opérandes intégraux :

C#

```
uint a = 0b_1111_1000;
uint b = 0b_0001_1100;
uint c = a ^ b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 11100100
```

Pour les opérandes `bool`, l'opérateur `^` calcule le OR exclusif logique de ses opérandes.

L'opérateur OU logique |

L'opérateur `|` calcule le OR logique au niveau du bit de ses opérandes intégraux :

```
C#

uint a = 0b_1010_0000;
uint b = 0b_1001_0001;
uint c = a | b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 10110001
```

Pour les opérandes `bool`, l'opérateur `|` calcule le OR logique de ses opérandes.

Assignation composée

Pour un opérateur binaire `op`, une expression d'assignation composée du formulaire

```
C#

x op= y
```

équivaut à :

```
C#

x = x op y
```

sauf que `x` n'est évalué qu'une seule fois.

L'exemple suivant montre l'utilisation de l'assignation composée avec des opérateurs de décalage et des opérateurs au niveau du bit :

```
C#

uint INITIAL_VALUE = 0b_1111_1000;

uint a = INITIAL_VALUE;
a &= 0b_1001_1101;
Display(a);  // output: 10011000

a = INITIAL_VALUE;
a |= 0b_0011_0001;
Display(a);  // output: 11111001

a = INITIAL_VALUE;
a ^= 0b_1000_0000;
```

```
Display(a);  // output: 01111000

a = INITIAL_VALUE;
a <<= 2;
Display(a);  // output: 1111100000

a = INITIAL_VALUE;
a >>= 4;
Display(a);  // output: 00001111

a = INITIAL_VALUE;
a >>>= 4;
Display(a);  // output: 00001111

void Display(uint x) => Console.WriteLine($"{Convert.ToString(x, toBase:
2).PadLeft(8, '0'), 8}");
```

En raison des promotions numériques, le résultat de l'opération `op` risque de ne pas être implicitement convertible en type `T` de `x`. Dans ce cas, si `op` est un opérateur prédéfini et que le résultat de l'opération est explicitement convertible en type `T` de `x`, une expression d'assignation composée de la forme `x op= y` équivaut à `x = (T)(x op y)`, sauf que `x` n'est évalué qu'une seule fois. L'exemple suivant illustre ce comportement :

C#

```
byte x = 0b_1111_0001;

int b = x << 8;
Console.WriteLine($"{Convert.ToString(b, toBase: 2)}");  // output:
1111000100000000

x <<= 8;
Console.WriteLine(x);  // output: 0
```

Priorité des opérateurs

La liste suivante présente les opérateurs au niveau du bit et les opérateurs de décalage par ordre de précédence, de la plus élevée à la plus basse :

- Opérateur de complément de bits `~`
- Opérateurs de décalage `<<`, `>>` et `>>>`
- L'opérateur AND logique `&`
- Opérateur OR exclusif logique `^`
- Opérateur OR logique `|`

Utilisez des parenthèses, `()`, pour modifier l'ordre d'évaluation imposé par la précédence des opérateurs :

```
C#
```

```
uint a = 0b_1101;
uint b = 0b_1001;
uint c = 0b_1010;

uint d1 = a | b & c;
Display(d1);  // output: 1101

uint d2 = (a | b) & c;
Display(d2);  // output: 1000

void Display(uint x) => Console.WriteLine($"{Convert.ToString(x, toBase: 2),
4}");
```

Pour obtenir la liste complète des opérateurs C# classés par niveau de priorité, consultez la section Priorité des opérateurs de l'article Opérateurs C#.

Valeur de décalage des opérateurs de décalage

Pour les opérateurs de décalage intégrés `<<`, `>>` et `>>>`, le type de l'opérande droit doit être `int` ou un type ayant une conversion numérique implicite prédéfinie sur `int`.

Pour les expressions `x << count`, `x >> count` et `x >>> count`, la valeur de décalage réelle dépend du type de `x`, comme suit :

- Si le type de `x` est `int` ou `uint`, la valeur de décalage est définie par les *cinq* bits de poids faible de l'opérande droit. La valeur de décalage est donc calculée à partir de `count & 0x1F` (ou de `count & 0b_1_1111`).
- Si le type de `x` est `long` ou `ulong`, la valeur de décalage est définie par les *six* bits de poids faible de l'opérande droit. La valeur de décalage est donc calculée à partir de `count & 0x3F` (ou de `count & 0b_11_1111`).

L'exemple suivant illustre ce comportement :

```
C#
```

```
int count1 = 0b_0000_0001;
int count2 = 0b_1110_0001;

int a = 0b_0001;
Console.WriteLine($"{a} << {count1} is {a << count1}; {a} << {count2} is {a
```

```
<< count2}");
// Output:
// 1 << 1 is 2; 1 << 225 is 2

int b = 0b_0100;
Console.WriteLine($"{b} >> {count1} is {b >> count1}; {b} >> {count2} is {b
>> count2}");
// Output:
// 4 >> 1 is 2; 4 >> 225 is 2

int count = -31;
int c = 0b_0001;
Console.WriteLine($"{c} << {count} is {c << count}");
// Output:
// 1 << -31 is 2
```

> ⓘ **Notes**
>
> Comme le montre l'exemple précédent, le résultat d'une opération de décalage peut être différent de zéro, même si la valeur de l'opérande droit est supérieure au nombre de bits dans l'opérande gauche.

Opérateurs logiques d'énumération

Les opérateurs `~`, `&`, `|` et `^` sont également pris en charge par tout type d'énumération. Pour les opérandes du même type énumération, une opération logique est effectuée sur les valeurs correspondantes du type intégral sous-jacent. Par exemple, pour tout `x` et `y` d'un type énumération `T` avec un type sous-jacent `U`, l'expression `x & y` produit le même résultat que l'expression `(T)((U)x & (U)y)`.

Vous utilisez généralement des opérateurs logiques au niveau du bit avec un type énumération qui est défini à l'aide de l'attribut Flags. Pour plus d'informations, consultez la section Types énumération comme indicateurs binaires de l'articleTypes énumération.

Capacité de surcharge de l'opérateur

Un type défini par l'utilisateur peut surcharger les opérateurs `~`, `<<`, `>>`, `>>>`, `&`, `|` et `^`. Quand un opérateur binaire est surchargé, l'opérateur d'assignation composée correspondant est aussi implicitement surchargé. Un type défini par l'utilisateur ne peut pas surcharger explicitement un opérateur d'assignation composée.

Si un type défini par l'utilisateur `T` surcharge l'opérateur `<<`, `>>` ou `>>>`, le type de l'opérande gauche doit être `T`. En C# 10 et versions antérieures, le type de l'opérande

droit doit être `int`. Depuis C# 11, l'opérande droit d'un opérateur de décalage surchargé peut être d'un type quelconque.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Opérateur de complément de bits
- Opérateurs de décalage
- Opérateurs logiques
- Assignation composée
- Promotions numériques
- C# 11 – Exigences de décalage assouplies
- C# 11 – Opérateur logique de décalage vers la droite

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- Opérateurs logiques booléens

Opérateurs d'égalité : testent si deux objets sont égaux ou non

Article • 07/04/2023

Les opérateurs == (égalité) et != (inégalité) vérifient si leurs opérandes sont égaux ou non. Les types de valeurs sont égaux lorsque leur contenu est égal. Les types de référence sont égaux lorsque les deux variables font référence au même stockage.

Opérateur d'égalité ==

L'opérateur d'égalité `==` retourne `true` si ses opérandes sont égaux, `false` dans le cas contraire.

Égalité de types valeur

Les opérandes des types valeur intégrés sont égaux si leurs valeurs sont égales :

```C#
int a = 1 + 2 + 3;
int b = 6;
Console.WriteLine(a == b);  // output: True

char c1 = 'a';
char c2 = 'A';
Console.WriteLine(c1 == c2);  // output: False
Console.WriteLine(c1 == char.ToLower(c2));  // output: True
```

> ⓘ **Notes**
>
> Pour les opérateurs `==`, `<`, `>`, `<=` et `>=`, si un des opérandes n'est pas un nombre (**Double.NaN** ou **Single.NaN**), le résultat de l'opération est `false`. Cela signifie que la valeur `NaN` n'est ni supérieure à, ni inférieure à, ni égale à n'importe quelle autre valeur `double` (ou `float`), y compris `NaN`. Pour plus d'informations et des exemples, consultez l'article de référence **Double.NaN** ou **Single.NaN**.

Deux opérandes du même type enum sont égaux si les valeurs correspondantes du type intégral sous-jacent sont égales.

Par défaut, les types struct définis par l'utilisateur ne prennent pas en charge l'opérateur `==`. Pour prendre en charge l'opérateur `==`, un struct défini par l'utilisateur doit le surcharger.

Les opérateurs `==` et `!=` sont pris en charge par les tuples C#. Pour plus d'informations, consultez la section Égalité des tuples de l'article Types de tuples.

Égalité de types référence

Par défaut, deux opérandes de type référence non-enregistrement sont égaux s'ils font référence au même objet :

C#

```
public class ReferenceTypesEquality
{
    public class MyClass
    {
        private int id;

        public MyClass(int id) => this.id = id;
    }

    public static void Main()
    {
        var a = new MyClass(1);
        var b = new MyClass(1);
        var c = a;
        Console.WriteLine(a == b);  // output: False
        Console.WriteLine(a == c);  // output: True
    }
}
```

Comme le montre l'exemple, les types référence définis par l'utilisateur prennent en charge l'opérateur `==` par défaut. Un type référence peut toutefois surcharger l'opérateur `==`. Si un type référence surcharge l'opérateur `==`, utilisez la méthode Object.ReferenceEquals pour vérifier si deux références de ce type font référence au même objet.

Égalité des types d'enregistrements

Disponibles dans C# 9.0 et versions ultérieures, les types d'enregistrements prennent en charge les opérateurs `==` et `!=` qui fournissent par défaut une sémantique d'égalité des valeurs. Autrement dit, deux opérandes d'enregistrement sont égaux quand les deux

sont `null` ou de valeurs correspondantes pour tous les champs et que les propriétés implémentées automatiquement sont égales.

C#

```
public class RecordTypesEquality
{
    public record Point(int X, int Y, string Name);
    public record TaggedNumber(int Number, List<string> Tags);

    public static void Main()
    {
        var p1 = new Point(2, 3, "A");
        var p2 = new Point(1, 3, "B");
        var p3 = new Point(2, 3, "A");

        Console.WriteLine(p1 == p2);  // output: False
        Console.WriteLine(p1 == p3);  // output: True

        var n1 = new TaggedNumber(2, new List<string>() { "A" });
        var n2 = new TaggedNumber(2, new List<string>() { "A" });
        Console.WriteLine(n1 == n2);  // output: False
    }
}
```

Comme le montre l'exemple précédent, pour les membres de type référence non-enregistrement, leurs valeurs de référence sont comparées, et non les instances référencées.

Égalité de chaîne

Deux opérandes string sont égaux lorsque les deux sont `null` ou lorsque les deux instances de chaîne sont de même longueur et ont des caractères identiques dans chaque position de caractère :

C#

```
string s1 = "hello!";
string s2 = "HeLLo!";
Console.WriteLine(s1 == s2.ToLower());  // output: True

string s3 = "Hello!";
Console.WriteLine(s1 == s3);  // output: False
```

Les comparaisons d'égalité de chaîne sont des comparaisons ordinales respectant la casse. Pour plus d'informations sur la comparaison de chaînes, consultez Comment comparer des chaînes en C#.

Égalité de délégué

Deux opérandes délégués du même type à l'exécution sont égaux si les deux sont `null` ou si leurs listes d'appel sont de même longueur et ont des entrées égales dans chaque position :

C#

```
Action a = () => Console.WriteLine("a");

Action b = a + a;
Action c = a + a;
Console.WriteLine(object.ReferenceEquals(b, c));  // output: False
Console.WriteLine(b == c);  // output: True
```

Pour plus d'informations, consultez la section Opérateurs d'égalité de délégué de la spécification du langage C#.

Les délégués qui sont produits à partir de l'évaluation d'expressions lambda identiques sémantiquement ne sont pas égaux, comme l'indique l'exemple suivant :

C#

```
Action a = () => Console.WriteLine("a");
Action b = () => Console.WriteLine("a");

Console.WriteLine(a == b);  // output: False
Console.WriteLine(a + b == a + b);  // output: True
Console.WriteLine(b + a == a + b);  // output: False
```

Opérateur d'inégalité !=

L'opérateur d'inégalité `!=` retourne `true` si ses opérandes ne sont pas égaux, `false` dans le cas contraire. Pour les opérandes des types intégrés, l'expression `x != y` produit le même résultat que l'expression `!(x == y)`. Pour plus d'informations sur l'égalité des types, consultez la section Opérateur d'égalité.

L'exemple suivant illustre l'utilisation de l'opérateur `!=` :

C#

```
int a = 1 + 1 + 2 + 3;
int b = 6;
Console.WriteLine(a != b);  // output: True
```

```
string s1 = "Hello";
string s2 = "Hello";
Console.WriteLine(s1 != s2);  // output: False

object o1 = 1;
object o2 = 1;
Console.WriteLine(o1 != o2);  // output: True
```

Capacité de surcharge de l'opérateur

Un type défini par l'utilisateur peut surcharger les opérateurs `==` et `!=`. Si un type surcharge un des deux opérateurs, il doit aussi en surcharger un autre.

Un type d'enregistrement ne peut pas surcharger explicitement les opérateurs `==` et `!=`. Si vous devez modifier le comportement des opérateurs `==` et `!=` pour le type d'enregistrement `T`, implémentez la méthode IEquatable<T>.Equals avec la signature suivante :

C#

```
public virtual bool Equals(T? other);
```

spécification du langage C#

Pour plus d'informations, consultez la section Opérateurs relationnels et de test de type de la spécification du langage C#.

Pour plus d'informations sur l'égalité des types d'enregistrements, consultez la section Membres d'égalité de la note de proposition de fonctionnalité d'enregistrement.

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- System.IEquatable<T>
- Object.Equals
- Object.ReferenceEquals
- Comparaisons d'égalité
- Opérateurs de comparaison

Opérateurs de comparaison (référence C#)

Article • 07/04/2023

Les opérateurs de comparaison, également appelés relationnels, < (inférieur à), > (supérieur à), <= (inférieur ou égal à), et >= (supérieur ou égal à) comparent leurs opérandes. Ces opérateurs sont pris en charge par tous les types numériques intégraux et à virgule flottante.

> ⓘ **Notes**
>
> Pour les opérateurs `==`, `<`, `>`, `<=` et `>=`, si un des opérandes n'est pas un nombre (**Double.NaN** ou **Single.NaN**), le résultat de l'opération est `false`. Cela signifie que la valeur `NaN` n'est ni supérieure à, ni inférieure à, ni égale à n'importe quelle autre valeur `double` (ou `float`), y compris `NaN`. Pour plus d'informations et des exemples, consultez l'article de référence **Double.NaN** ou **Single.NaN**.

Le type char prend également en charge les opérateurs de comparaison. Dans le cas des opérandes `char`, les codes de caractères correspondants sont comparés.

Les types d'énumération prennent également en charge les opérateurs de comparaison. Pour les opérandes du même type enum, les valeurs correspondantes du type intégral sous-jacent sont comparées.

Les opérateurs == et != vérifient si leurs opérandes sont égaux ou non.

Opérateur Inférieur à <

L'opérateur `<` retourne `true` si son opérande de partie gauche est inférieur à son opérande de partie droite, `false` dans le cas contraire :

C#

```
Console.WriteLine(7.0 < 5.1);   // output: False
Console.WriteLine(5.1 < 5.1);   // output: False
Console.WriteLine(0.0 < 5.1);   // output: True

Console.WriteLine(double.NaN < 5.1);   // output: False
Console.WriteLine(double.NaN >= 5.1);  // output: False
```

Opérateur Supérieur à >

L'opérateur `>` retourne `true` si son opérande de partie gauche est supérieur à son opérande de partie droite, `false` dans le cas contraire :

C#

```
Console.WriteLine(7.0 > 5.1);   // output: True
Console.WriteLine(5.1 > 5.1);   // output: False
Console.WriteLine(0.0 > 5.1);   // output: False

Console.WriteLine(double.NaN > 5.1);   // output: False
Console.WriteLine(double.NaN <= 5.1);  // output: False
```

Opérateur Inférieur ou égal à <=

L'opérateur `<=` retourne `true` si son opérande de partie gauche est inférieur ou égal à son opérande de partie droite, `false` dans le cas contraire :

C#

```
Console.WriteLine(7.0 <= 5.1);   // output: False
Console.WriteLine(5.1 <= 5.1);   // output: True
Console.WriteLine(0.0 <= 5.1);   // output: True

Console.WriteLine(double.NaN > 5.1);   // output: False
Console.WriteLine(double.NaN <= 5.1);  // output: False
```

Opérateur Supérieur ou égal à >=

L'opérateur `>=` retourne `true` si son opérande de partie gauche est supérieur ou égal à son opérande de partie droite, `false` dans le cas contraire :

C#

```
Console.WriteLine(7.0 >= 5.1);   // output: True
Console.WriteLine(5.1 >= 5.1);   // output: True
Console.WriteLine(0.0 >= 5.1);   // output: False

Console.WriteLine(double.NaN < 5.1);   // output: False
Console.WriteLine(double.NaN >= 5.1);  // output: False
```

Capacité de surcharge de l'opérateur

Un type défini par l'utilisateur peut surcharger les opérateurs `<`, `>`, `<=` et `>=`.

Si un type surcharge un des opérateurs `<` ou `>`, il doit surcharger à la fois `<` et `>`. Si un type surcharge un des opérateurs `<=` ou `>=`, il doit surcharger à la fois `<=` et `>=`.

spécification du langage C#

Pour plus d'informations, consultez la section Opérateurs relationnels et de test de type de la spécification du langage C#.

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- System.IComparable<T>
- Opérateurs d'égalité

Opérateurs et expressions d'accès aux membres : opérateurs de point, d'indexeur et d'appel.

Article • 16/03/2023

Vous utilisez plusieurs opérateurs et expressions pour accéder à un membre de type. Ces opérateurs incluent l'accès aux membres (`.`), l'élément de tableau ou l'accès à l'indexeur (`[]`), l'index de fin (`^`), la plage (`..`), les opérateurs conditionnels Null (`?.` et `?[]`) et l'appel de méthode (`()`). Il s'agit notamment des opérateurs d'accès au membre *conditionnel Null* (`?.`) et d'indexeur (`?[]`).

- . (accès aux membres) : accédez à un membre d'un espace de noms ou d'un type
- [] (accès à un indexeur ou à un élément de tableau) : pour accéder à un élément de tableau ou à un indexeur de type
- ?. et ?[] (opérateurs conditionnels Null) : pour effectuer une opération d'accès à un membre ou élément uniquement si un opérande est non Null
- () (appel) : pour appeler une méthode sollicitée ou appeler un délégué
- ^ (index de fin) : pour indiquer que la position de l'élément provient de la fin d'une séquence
- .. (plage) : pour spécifier une plage d'index que vous pouvez utiliser pour obtenir une plage d'éléments de séquence

Expression d'accès au membre `.`

Le jeton `.` sert à accéder à l'un des membres d'un espace de noms ou d'un type, comme le montrent les exemples suivants :

- Utilisez `.` pour accéder à un espace de noms imbriqué dans un autre, comme le montre l'exemple suivant avec la directive using :

C#

```
using System.Collections.Generic;
```

- Utilisez `.` pour former un *nom qualifié* permettant d'accéder à un type dans un espace de noms, comme le montre le code suivant :

C#

```
System.Collections.Generic.IEnumerable<int> numbers = new int[] { 1, 2, 3 };
```

Utilisez une directive using pour rendre facultative l'utilisation de noms qualifiés.

- Utilisez `.` pour accéder aux membres de type, statiques et non statiques, comme le montre le code suivant :

C#

```
var constants = new List<double>();
constants.Add(Math.PI);
constants.Add(Math.E);
Console.WriteLine($"{constants.Count} values to show:");
Console.WriteLine(string.Join(", ", constants));
// Output:
// 2 values to show:
// 3.14159265358979, 2.71828182845905
```

Vous pouvez également utiliser `.` pour accéder à une méthode d'extension.

Opérateur d'indexeur []

Les crochets, `[]`, sont généralement utilisés pour l'accès à un élément tableau, indexeur ou pointeur.

Accès aux tableaux

L'exemple suivant montre comment accéder à des éléments tableau :

C#

```
int[] fib = new int[10];
fib[0] = fib[1] = 1;
for (int i = 2; i < fib.Length; i++)
{
    fib[i] = fib[i - 1] + fib[i - 2];
}
Console.WriteLine(fib[fib.Length - 1]);  // output: 55

double[,] matrix = new double[2,2];
matrix[0,0] = 1.0;
matrix[0,1] = 2.0;
matrix[1,0] = matrix[1,1] = 3.0;
var determinant = matrix[0,0] * matrix[1,1] - matrix[1,0] * matrix[0,1];
Console.WriteLine(determinant);  // output: -3
```

Si un index de tableau est en dehors des limites de la dimension correspondante d'un tableau, une IndexOutOfRangeException est levée.

Comme le montre l'exemple précédent, vous utilisez également des crochets quand vous déclarez un type tableau ou instanciez une instance de tableau.

Pour plus d'informations sur les tableaux, consultez Tableaux.

Accès aux indexeurs

L'exemple suivant utilise le type .NET Dictionary<TKey,TValue> afin d'illustrer l'accès aux indexeurs :

C#

```
var dict = new Dictionary<string, double>();
dict["one"] = 1;
dict["pi"] = Math.PI;
Console.WriteLine(dict["one"] + dict["pi"]);  // output: 4.14159265358979
```

Les indexeurs vous permettent d'indexer des instances d'un type défini par l'utilisateur en procédant de la même façon que pour l'indexation de tableau. Contrairement aux index de tableau, qui doivent être des entiers, les paramètres d'indexeur peuvent être déclarés comme étant de n'importe quel type.

Pour plus d'informations sur les indexeurs, consultez Indexeurs.

Autres utilisations de []

Pour plus d'informations concernant l'accès à l'élément de pointeur, consultez la section Opérateur d'accès à l'élément de pointeur [] de l'article Opérateurs associés au pointeur.

Vous utilisez également des crochets pour spécifier des attributs :

C#

```
[System.Diagnostics.Conditional("DEBUG")]
void TraceMethod() {}
```

Opérateurs conditionnels Null `?.` et `?[]`

Un opérateur conditionnel Null n'applique une opération d'accès à un membre, `?.` ou d'accès à un élément, `?[]`, à son opérande que si cet opérande a la valeur non Null ;

sinon, il retourne la valeur `null`. Plus précisément :

- Si `a` renvoie la valeur `null`, le résultat de `a?.x` ou `a?[x]` est `null`.
- Si `a` renvoie la valeur non Null, le résultat de `a?.x` ou `a?[x]` est le même que le résultat de `a.x` ou `a[x]`, respectivement.

> ⓘ **Notes**
>
> Si `a.x` ou `a[x]` lève une exception, `a?.x` ou `a?[x]` lève la même exception pour la valeur non Null `a`. Par exemple, si `a` est une instance de tableau non Null et que `x` est en dehors des limites de `a`, `a?[x]` lève une **IndexOutOfRangeException**.

Les opérateurs conditionnels Null ont un effet de court-circuit. Autrement dit, si une opération dans une chaîne d'opérations d'accès au membre ou à l'élément conditionnelles retourne une valeur `null`, le reste de la chaîne ne s'exécute pas. Dans l'exemple suivant, `B` n'est pas évalué si `A` renvoie la valeur `null` et `C` n'est pas évalué si `A` ou `B` renvoie la valeur `null` :

```
C#

A?.B?.Do(C);
A?.B?[C];
```

Si `A` peut être Null, mais que `B` et `C` ne peuvent être Null si A n'est pas Null, vous devez uniquement appliquer l'opérateur conditionnel Null à `A` :

```
C#

A?.B.C();
```

Dans l'exemple précédent, `B` n'est pas évalué et `C()` n'est pas appelé si `A` est Null. Toutefois, si l'accès au membre chaîné est interrompu, par exemple par des parenthèses comme dans `(A?.B).C()`, il n'y a pas d'effet de court-circuit.

Les exemples suivants illustrent l'utilisation des opérateurs `?.` et `?[]` :

```
C#

double SumNumbers(List<double[]> setsOfNumbers, int indexOfSetToSum)
{
    return setsOfNumbers?[indexOfSetToSum]?.Sum() ?? double.NaN;
```

```
}

var sum1 = SumNumbers(null, 0);
Console.WriteLine(sum1);  // output: NaN

var numberSets = new List<double[]>
{
    new[] { 1.0, 2.0, 3.0 },
    null
};

var sum2 = SumNumbers(numberSets, 0);
Console.WriteLine(sum2);  // output: 6

var sum3 = SumNumbers(numberSets, 1);
Console.WriteLine(sum3);  // output: NaN
```

C#

```
namespace MemberAccessOperators2;

public static class NullConditionalShortCircuiting
{
    public static void Main()
    {
        Person person = null;
        person?.Name.Write(); // no output: Write() is not called due to
short-circuit.
        try
        {
            (person?.Name).Write();
        }
        catch (NullReferenceException)
        {
            Console.WriteLine("NullReferenceException");
        }; // output: NullReferenceException
    }
}

public class Person
{
    public FullName Name { get; set; }
}

public class FullName
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public void Write()
    {
        Console.WriteLine($"{FirstName} {LastName}");
    }
}
```

Le premier des deux exemples précédents utilise également l'opérateur de coalescence nulle?? pour spécifier une autre expression à évaluer si le résultat d'une opération conditionnelle nulle est `null`.

Si `a.x` ou `a[x]` correspond à un type de valeur est d'un type de valeur pouvant accepter la valeur Null `T`, `a?.x` ou `a?[x]` est le type de valeur pouvant accepter la valeur Null correspondante `T?`. Si vous avez besoin d'une expression de type `T`, appliquez l'opérateur de coalescence nulle `??` à une expression conditionnelle Null, comme l'illustre l'exemple suivant :

C#

```
int GetSumOfFirstTwoOrDefault(int[] numbers)
{
    if ((numbers?.Length ?? 0) < 2)
    {
        return 0;
    }
    return numbers[0] + numbers[1];
}

Console.WriteLine(GetSumOfFirstTwoOrDefault(null));  // output: 0
Console.WriteLine(GetSumOfFirstTwoOrDefault(new int[0]));  // output: 0
Console.WriteLine(GetSumOfFirstTwoOrDefault(new[] { 3, 4, 5 }));  // output:
7
```

Dans l'exemple précédent, si vous n'utilisez pas l'opérateur `??`, `numbers?.Length < 2` renvoie la valeur `false` lorsque `numbers` est `null`.

> **Notes**
>
> L'opérateur `?.` évalue son opérande de gauche au maximum une fois, ce qui garantit qu'il ne peut pas être remplacé par `null` après avoir été vérifié comme étant non Null.

L'opérateur d'accès aux membres conditionnels null `?.` est également appelé l'opérateur Elvis.

Appel de délégué thread-safe

Utilisez l'opérateur `?.` pour vérifier si un délégué est non Null et l'appeler de manière thread-safe (par exemple, quand vous déclenchez un événement), comme illustré dans le code suivant :

```
C#

PropertyChanged?.Invoke(…)
```

Ce code équivaut au code suivant :

```
C#

var handler = this.PropertyChanged;
if (handler != null)
{
    handler(…);
}
```

L'exemple précédent est un moyen thread-safe de s'assurer que seule une valeur non Null `handler` est appelée. Les instances de délégué étant immuables, aucun thread ne peut modifier l'objet référencé par la variable locale `handler`. Plus particulièrement, si le code exécuté par un autre thread se désabonne de l'événement `PropertyChanged` et que `PropertyChanged` devient `null` avant l'appel de `handler`, l'objet référencé par `handler` reste inchangé.

Expression d'appel ()

Utilisez des parenthèses, `()`, pour appeler une méthode ou un délégué.

L'exemple suivant montre comment appeler une méthode, avec ou sans arguments, et un délégué :

```
C#

Action<int> display = s => Console.WriteLine(s);

var numbers = new List<int>();
numbers.Add(10);
numbers.Add(17);
display(numbers.Count);   // output: 2

numbers.Clear();
display(numbers.Count);   // output: 0
```

Vous utilisez également des parenthèses quand vous appelez un constructeur avec l'opérateur new.

Autres utilisations de ()

Vous utilisez également des parenthèses pour ajuster l'ordre dans lequel évaluer les opérations dans une expression. Pour plus d'informations, consultez Opérateurs C#.

Les expressions cast, qui effectuent des conversions de type explicites, utilisent aussi des parenthèses.

Index à partir de l'opérateur de fin ^

L'opérateur `^` indique la position de l'élément à partir de la fin d'une séquence. Pour une séquence de longueur `length`, `^n` pointe vers l'élément avec un décalage `length - n` à partir du début d'une séquence. Par exemple, `^1` pointe vers le dernier élément d'une séquence et `^length` pointe vers le premier élément d'une séquence.

C#

```
int[] xs = new[] { 0, 10, 20, 30, 40 };
int last = xs[^1];
Console.WriteLine(last);  // output: 40

var lines = new List<string> { "one", "two", "three", "four" };
string prelast = lines[^2];
Console.WriteLine(prelast);  // output: three

string word = "Twenty";
Index toFirst = ^word.Length;
char first = word[toFirst];
Console.WriteLine(first);  // output: T
```

Comme l'illustre l'exemple précédent, l'expression `^e` est de type System.Index. Dans l'expression `^e`, le résultat de `e` doit être implicitement convertible en `int`.

Vous pouvez également utiliser l'opérateur `^` avec l'opérateur de plage pour créer une plage d'index. Pour plus d'informations, consultez Index et plages.

Opérateur de plage `..`

L'opérateur `..` spécifie le début et la fin d'une plage d'index en tant qu'opérandes. L'opérande de gauche correspond à un début *inclusif* de plage. L'opérande de droite correspond à une fin *exclusive* de plage. L'un des opérandes peut être un index à partir du début ou de la fin d'une séquence, comme l'illustre l'exemple suivant :

C#

```
int[] numbers = new[] { 0, 10, 20, 30, 40, 50 };
int start = 1;
int amountToTake = 3;
int[] subset = numbers[start..(start + amountToTake)];
Display(subset);  // output: 10 20 30

int margin = 1;
int[] inner = numbers[margin..^margin];
Display(inner);  // output: 10 20 30 40

string line = "one two three";
int amountToTakeFromEnd = 5;
Range endIndices = ^amountToTakeFromEnd..^0;
string end = line[endIndices];
Console.WriteLine(end);  // output: three

void Display<T>(IEnumerable<T> xs) => Console.WriteLine(string.Join(" ",
xs));
```

Comme l'illustre l'exemple précédent, l'expression `a..b` est de type System.Range. Dans l'expression `a..b`, les résultats de `a` et `b` doivent être implicitement convertibles en Int32 ou Index.

> ⓘ **Important**
>
> Les conversions implicites de `int` à `Index` lèvent **ArgumentOutOfRangeException** lorsque la valeur est négative.

Vous pouvez omettre l'un des opérandes de l'opérateur `..` pour obtenir une plage ouverte :

- `a..` équivaut à `a..^0`
- `..b` équivaut à `0..b`
- `..` équivaut à `0..^0`

C#

```
int[] numbers = new[] { 0, 10, 20, 30, 40, 50 };
int amountToDrop = numbers.Length / 2;

int[] rightHalf = numbers[amountToDrop..];
Display(rightHalf);  // output: 30 40 50

int[] leftHalf = numbers[..^amountToDrop];
Display(leftHalf);  // output: 0 10 20

int[] all = numbers[..];
```

```
Display(all);  // output: 0 10 20 30 40 50

void Display<T>(IEnumerable<T> xs) => Console.WriteLine(string.Join(" ",
xs));
```

Le tableau suivant montre différentes façons d'exprimer des plages de collection :

Expression d'opérateur de plage	Description
`..`	Toutes les valeurs de la collection.
`..end`	Valeurs comprises du début à la `end` exclusivement.
`start..`	Valeurs comprises du `start` inclusivement à la fin.
`start..end`	Valeurs comprises du `start` inclusivement à la `end` exclusivement.
`^start..`	Valeurs comprises du `start` inclusivement à la fin en comptant à partir de la fin.
`..^end`	Valeurs comprises du début à la `end` exclusivement en comptant à partir de la fin.
`start..^end`	Valeurs comprises du `start` inclusivement à la `end` exclusivement en comptant à partir de la fin.
`^start..^end`	Valeurs comprises du `start` inclusivement à la `end` exclusivement en comptant à partir de la fin.

L'exemple suivant illustre l'effet de l'utilisation de toutes les plages présentées dans le tableau précédent :

C#

```
int[] oneThroughTen =
{
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10
};

Write(oneThroughTen, ..);
Write(oneThroughTen, ..3);
Write(oneThroughTen, 2..);
Write(oneThroughTen, 3..5);
Write(oneThroughTen, ^2..);
Write(oneThroughTen, ..^3);
Write(oneThroughTen, 3..^4);
Write(oneThroughTen, ^4..^2);

static void Write(int[] values, Range range) =>
    Console.WriteLine($"{range}:\t{string.Join(", ", values[range])}");
```

```
// Sample output:
//      0..^0:      1, 2, 3, 4, 5, 6, 7, 8, 9, 10
//      0..3:       1, 2, 3
//      2..^0:      3, 4, 5, 6, 7, 8, 9, 10
//      3..5:       4, 5
//      ^2..^0:     9, 10
//      0..^3:      1, 2, 3, 4, 5, 6, 7
//      3..^4:      4, 5, 6
//      ^4..^2:     7, 8
```

Pour plus d'informations, consultez Index et plages.

Capacité de surcharge de l'opérateur

Les opérateurs `.`, `()`, `^` et `..` ne peuvent pas être surchargés. L'opérateur `[]` est également considéré comme un opérateur non surchargeable. Utilisez des indexeurs pour prendre en charge l'indexation avec des types définis par l'utilisateur.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Accès aux membres
- Accès aux éléments
- Accès au membre conditionnel Null
- Expressions d'appels

Pour plus d'informations sur les index et les plages, consultez la note de proposition de fonctionnalité.

Voir aussi

- Utiliser l'opérateur d'index (règle de style IDE0056)
- Utiliser l'opérateur de plage (règle de style IDE0057)
- Utiliser l'appel délégué conditionnel (règle de style IDE1005)
- Informations de référence sur C#
- Opérateurs et expressions C#
- ?? (opérateur de coalescence nulle)
- :: opérateur

Opérateurs de test de type et expressions cast : `is`, `as` `typeof` et casts

Article • 10/05/2023

Ces opérateurs et expressions effectuent un contrôle ou une conversion de type. L'`is` opérateur vérifie si le type d'exécution d'une expression est compatible avec un type donné. L'`as` opérateur convertit explicitement une expression en un type donné si son type d'exécution est compatible avec ce type. Les expressions cast effectuent une conversion explicite en un type cible. L'`typeof` opérateur obtient l'instance System.Type d'un type.

Opérateur is

L'opérateur `is` vérifie si le type d'un résultat d'expression est compatible avec un type donné. L'opérateur `is` teste également un résultat d'expression par rapport à un modèle.

L'expression avec l'opérateur de test de type `is` a la forme suivante

```
C#

E is T
```

où `E` est une expression qui retourne une valeur et `T` est le nom d'un type ou un d'un paramètre de type. `E` ne peut pas être une méthode anonyme ou une expression lambda.

L'opérateur `is` renvoie `true` lorsqu'un résultat d'expression n'est pas nul et que l'une des conditions suivantes est remplie :

- Le type d'exécution d'un résultat d'expression est `T`.
- Le type d'exécution d'un résultat d'expression dérive du type `T`, implémente l'interface `T`, ou une autre conversion de référence implicite en `T` existe à partir de celui-ci.
- Le type d'exécution d'un résultat d'expression est un type valeur pouvant accepter la valeur Null avec le type sous-jacent `T`, et Nullable<T>.HasValue est `true`.

- Il existe une conversion boxing ou unboxing en type `T` à partir du type d'exécution d'un résultat d'expression.

L'opérateur `is` ne considère pas les conversions définies par l'utilisateur.

L'exemple suivant montre que l'opérateur `is` renvoie `true` si le type d'exécution d'un résultat d'expression dérive d'un type donné, autrement dit, s'il existe une conversion de référence entre les types :

C#

```
public class Base { }

public class Derived : Base { }

public static class IsOperatorExample
{
    public static void Main()
    {
        object b = new Base();
        Console.WriteLine(b is Base);  // output: True
        Console.WriteLine(b is Derived);  // output: False

        object d = new Derived();
        Console.WriteLine(d is Base);  // output: True
        Console.WriteLine(d is Derived); // output: True
    }
}
```

L'exemple suivant montre que l'opérateur `is` prend en compte les conversions boxing et unboxing mais ne tient pas compte des conversions numériques :

C#

```
int i = 27;
Console.WriteLine(i is System.IFormattable);  // output: True

object iBoxed = i;
Console.WriteLine(iBoxed is int);  // output: True
Console.WriteLine(iBoxed is long);  // output: False
```

Pour plus d'informations sur les conversions C#, consultez le chapitre Conversions de la Spécification du langage C#.

Test de type avec des critères spéciaux

L'opérateur `is` teste également un résultat d'expression par rapport à un modèle. L'exemple suivant montre comment utiliser un modèle de déclaration pour vérifier le type d'exécution d'une expression :

```
C#
```

```
int i = 23;
object iBoxed = i;
int? jNullable = 7;
if (iBoxed is int a && jNullable is int b)
{
    Console.WriteLine(a + b);  // output 30
}
```

Pour plus d'informations sur les modèles pris en charge, consultez Modèles.

opérateur as

L'opérateur `as` convertit explicitement le résultat d'une expression en un type de valeur de référence ou nullable. Si la conversion est impossible, l'opérateur `as` renvoie `null`. Contrairement à une expression cast, l'opérateur `as` ne lève jamais d'exception.

L'expression de la forme

```
C#
```

```
E as T
```

où `E` est une expression qui retourne une valeur et `T` est le nom d'un type ou un d'un paramètre de type, avec le même résultat que

```
C#
```

```
E is T ? (T)(E) : (T)null
```

sauf que `E` n'est évalué qu'une seule fois.

L'opérateur `as` envisage uniquement les conversions de référence, nullable, boxing et unboxing. Vous ne pouvez pas utiliser l'opérateur `as` pour effectuer une conversion définie par l'utilisateur. Pour ce faire, utilisez une expression cast.

L'exemple suivant illustre l'utilisation de l'opérateur `as` :

```
C#
```

```
IEnumerable<int> numbers = new[] { 10, 20, 30 };
IList<int> indexable = numbers as IList<int>;
if (indexable != null)
{
    Console.WriteLine(indexable[0] + indexable[indexable.Count - 1]);  //
output: 40
}
```

> ⓘ Notes
>
> Comme le montre l'exemple précédent, vous devez comparer le résultat de l'expression `as` avec `null` pour vérifier si la conversion a réussi. Vous pouvez utiliser **l'opérateur is** pour tester si la conversion réussit et, si tel est le cas, pour affecter son résultat à une nouvelle variable.

Expression Cast

Une expression cast de la forme `(T)E` effectue une conversion explicite du résultat de l'expression `E` en type `T`. S'il n'existe aucune conversion explicite du type de `E` en type `T`, une erreur de compilation se produit. Au moment de l'exécution, une conversion explicite peut ne pas réussir, et une expression cast peut lever une exception.

L'exemple suivant montre des conversions numériques et de référence explicites :

C#

```
double x = 1234.7;
int a = (int)x;
Console.WriteLine(a);   // output: 1234

IEnumerable<int> numbers = new int[] { 10, 20, 30 };
IList<int> list = (IList<int>)numbers;
Console.WriteLine(list.Count);  // output: 3
Console.WriteLine(list[1]);  // output: 20
```

Pour plus d'informations sur les conversions explicites prises en charge, consultez la section Conversions explicites de la Spécification du langage C#. Pour plus d'informations sur la façon de définir une conversion de type explicite ou implicite personnalisée, consultez Opérateurs de conversion définie par l'utilisateur.

Autres utilisations de ()

Vous pouvez aussi utiliser des parenthèses pour appeler une méthode ou un délégué.

Une autre utilisation des parenthèses est d'ajuster l'ordre dans lequel évaluer les opérations dans une expression. Pour plus d'informations, consultez Opérateurs C#.

Opérateur typeof

L'opérateur `typeof` obtient l'instance System.Type pour un type. L'argument de l'opérateur `typeof` doit avoir le nom d'un type ou d'un paramètre de type, comme le montre l'exemple suivant :

C#

```
void PrintType<T>() => Console.WriteLine(typeof(T));

Console.WriteLine(typeof(List<string>));
PrintType<int>();
PrintType<System.Int32>();
PrintType<Dictionary<int, char>>();
// Output:
// System.Collections.Generic.List`1[System.String]
// System.Int32
// System.Int32
// System.Collections.Generic.Dictionary`2[System.Int32,System.Char]
```

L'argument ne doit pas être un type qui nécessite des annotations de métadonnées. Les types suivants en sont des exemples :

- `dynamic`
- `string?` (ou tout type référence Nullable)

Ces types ne sont pas directement représentés dans les métadonnées. Les types comprennent des attributs qui décrivent le type sous-jacent. Dans les deux cas, vous pouvez utiliser le type sous-jacent. Au lieu de `dynamic`, vous pouvez utiliser `object`. Au lieu de `string?`, vous pouvez utiliser `string`.

Vous pouvez également utiliser l'opérateur `typeof` avec des types génériques indépendants. Le nom d'un type générique indépendant doit contenir le nombre approprié de virgules, à savoir une de moins que le nombre de paramètres de type. L'exemple suivant illustre l'utilisation de l'opérateur `typeof` avec un type générique indépendant :

C#

```
Console.WriteLine(typeof(Dictionary<,>));
// Output:
// System.Collections.Generic.Dictionary`2[TKey,TValue]
```

Une expression ne peut pas être un argument de l'opérateur `typeof`. Pour obtenir l'instance System.Type du type d'exécution d'un résultat d'expression, utilisez la méthode Object.GetType.

Test de type avec l'opérateur `typeof`

Utilisez l'opérateur `typeof` pour vérifier si le type d'exécution du résultat de l'expression correspond exactement à un type donné. L'exemple suivant illustre la différence entre les contrôles de type effectués avec l'opérateur `typeof` et l'opérateur is :

C#

```
public class Animal { }

public class Giraffe : Animal { }

public static class TypeOfExample
{
    public static void Main()
    {
        object b = new Giraffe();
        Console.WriteLine(b is Animal);  // output: True
        Console.WriteLine(b.GetType() == typeof(Animal));  // output: False

        Console.WriteLine(b is Giraffe);  // output: True
        Console.WriteLine(b.GetType() == typeof(Giraffe));  // output: True
    }
}
```

Capacité de surcharge de l'opérateur

Les opérateurs `is`, `as` et `typeof` ne peuvent pas être surchargés.

Un type défini par l'utilisateur ne peut pas surcharger l'opérateur `()`, mais vous pouvez définir des conversions de type personnalisées qui peuvent être effectuées par une expression cast. Pour plus d'informations, consultez Opérateurs de conversion définie par l'utilisateur.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- L'opérateur is
- L'opérateur as
- Expressions cast
- L'opérateur typeof

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- Guide pratique pour caster de manière sécurisée avec les critères spéciaux, ainsi que les opérateurs is et as
- Génériques en .NET

Opérateurs de conversion explicites et implicites définis par l'utilisateur

Article • 07/04/2023

Un type défini par l'utilisateur peut définir une conversion implicite ou explicite personnalisée depuis ou vers un autre type. Les conversions implicites ne nécessitent pas une syntaxe spéciale à appeler et peuvent se produire dans diverses situations, par exemple, dans les appels de méthodes et les affectations. Les conversions implicites C# prédéfinies réussissent toujours et ne lèvent jamais d'exception. Les conversions implicites définies par l'utilisateur doivent aussi se comporter de cette façon. Si une conversion personnalisée peut lever une exception ou perdre des informations, définissez-la comme conversion explicite.

Les conversions définies par l'utilisateur ne sont pas prises en compte par les opérateurs is et as. Utilisez l'expression de cast pour appeler une conversion explicite définie par l'utilisateur.

Utilisez les mots clés `operator` et `implicit` ou `explicit` pour définir une conversion implicite ou explicite, respectivement. Le type qui définit une conversion doit être un type source ou un type cible de cette conversion. Une conversion entre deux types définis par l'utilisateur peut être définie dans l'un ou l'autre des deux types.

L'exemple suivant montre comment définir une conversion implicite et explicite :

C#

```
using System;

public readonly struct Digit
{
    private readonly byte digit;

    public Digit(byte digit)
    {
        if (digit > 9)
        {
            throw new ArgumentOutOfRangeException(nameof(digit), "Digit
cannot be greater than nine.");
        }
        this.digit = digit;
    }

    public static implicit operator byte(Digit d) => d.digit;
    public static explicit operator Digit(byte b) => new Digit(b);
```

```
    public override string ToString() => $"{digit}";
}

public static class UserDefinedConversions
{
    public static void Main()
    {
        var d = new Digit(7);

        byte number = d;
        Console.WriteLine(number);  // output: 7

        Digit digit = (Digit)number;
        Console.WriteLine(digit);  // output: 7
    }
}
```

À compter de C# 11, vous pouvez définir des opérateurs de conversion explicite *vérifiés*. Pour plus d'informations, consultez la section Opérateurs vérifiés définis par l'utilisateur de l'article Opérateurs arithmétiques.

Utilisez également le mot clé `operator` pour surcharger un opérateur C# prédéfini. Pour plus d'informations, consultez Surcharge d'opérateur.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Opérateurs de conversion
- Conversions définies par l'utilisateur
- Conversions implicites
- Conversions explicites

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- Surcharge d'opérateur
- Opérateurs de conversion et de test de type
- Cast et conversion de types
- Instructions de conception - Opérateurs de conversion
- Conversions explicites définies par l'utilisateur chaînées en C#

Opérateurs liés aux pointeurs : permettent de prendre l'adresse des variables, de déréférencer des emplacements de stockage et d'accéder à des emplacements de la mémoire.

Article • 07/04/2023

Les opérateurs liés aux pointeurs vous permettent de prendre l'adresse d'une variable (`&`), de déréférencer un pointeur (`*`), de comparer les valeurs des pointeurs, et d'ajouter ou de soustraire des pointeurs et des entiers.

Vous utilisez les opérateurs suivants avec les pointeurs :

- Opérateur unaire & (address-of) : pour obtenir l'adresse d'une variable
- Opérateur unaire * (indirection du pointeur) : pour obtenir la variable pointée par un pointeur
- Opérateurs -> (accès aux membres) et [] (accès aux éléments)
- Opérateurs arithmétiques +, -, ++ et --
- Opérateurs de comparaison ==, !=, <, >, <= et >=

Pour plus d'informations sur les types de pointeurs, consultez Types pointeur.

> ⓘ **Notes**
>
> Toutes les opérations impliquant des pointeurs nécessitent un contexte **unsafe**. Le code qui contient des blocs unsafe doit être compilé avec l'option de compilateur **AllowUnsafeBlocks**.

Opérateur address-of &

L'opérateur unaire `&` retourne l'adresse de son opérande :

C#

```
unsafe
{
    int number = 27;
    int* pointerToNumber = &number;
```

```
    Console.WriteLine($"Value of the variable: {number}");
    Console.WriteLine($"Address of the variable:
{(long)pointerToNumber:X}");
}
// Output is similar to:
// Value of the variable: 27
// Address of the variable: 6C1457DBD4
```

L'opérande de l'opérateur `&` doit être une variable fixe. Les variables *fixes* se trouvent dans des emplacements de stockage qui ne sont pas affectés par le récupérateur de mémoire. Dans l'exemple précédent, la variable locale `number` est une variable fixe, car elle se trouve dans la pile. Les variables qui se trouvent dans des emplacements de stockage pouvant être affectés par le récupérateur de mémoire (par exemple, en étant déplacés) sont appelées variables *déplaçables*. Les champs d'objet et les éléments de tableau sont des exemples de variables déplaçables. Vous pouvez obtenir l'adresse d'une variable déplaçable en la fixant (ou en l'épinglant) à l'aide d'une instruction fixed. L'adresse obtenue est uniquement valide à l'intérieur du bloc d'une instruction `fixed`. L'exemple suivant montre comment utiliser une instruction `fixed` et l'opérateur `&` :

C#

```
unsafe
{
    byte[] bytes = { 1, 2, 3 };
    fixed (byte* pointerToFirst = &bytes[0])
    {
        // The address stored in pointerToFirst
        // is valid only inside this fixed statement block.
    }
}
```

Vous ne pouvez pas obtenir l'adresse d'une constante ou d'une valeur.

Pour plus d'informations sur les variables fixes et déplaçables, consultez la section Variables fixes et déplaçables de la spécification du langage C#.

L'opérateur binaire `&` calcule la logique AND de ses opérandes booléens, ou la logique AND au niveau du bit de ses opérandes de type intégral.

Opérateur d'indirection de pointeur *

L'opérateur unaire d'indirection de pointeur `*` permet d'obtenir la variable vers laquelle pointe son opérande. Il est également appelé « opérateur de déréférence ». L'opérande de l'opérateur `*` doit être un type de pointeur.

C#

```
unsafe
{
    char letter = 'A';
    char* pointerToLetter = &letter;
    Console.WriteLine($"Value of the `letter` variable: {letter}");
    Console.WriteLine($"Address of the `letter` variable:
{(long)pointerToLetter:X}");

    *pointerToLetter = 'Z';
    Console.WriteLine($"Value of the `letter` variable after update:
{letter}");
}
// Output is similar to:
// Value of the `letter` variable: A
// Address of the `letter` variable: DCB977DDF4
// Value of the `letter` variable after update: Z
```

Vous ne pouvez pas appliquer l'opérateur `*` à une expression de type `void*`.

L'opérateur binaire `*` calcule le produit de ses opérandes numériques.

Opérateur d'accès aux membres de pointeur ->

L'opérateur `->` associe l'indirection de pointeur à l'accès aux membres. C'est-à-dire que si `x` est un pointeur de type `T*` et que `y` est un membre accessible de `T`, une expression au format

C#

```
x->y
```

équivaut à :

C#

```
(*x).y
```

L'exemple suivant illustre l'utilisation de l'opérateur `->` :

C#

```
public struct Coords
{
    public int X;
```

```
    public int Y;
    public override string ToString() => $"({X}, {Y})";
}

public class PointerMemberAccessExample
{
    public static unsafe void Main()
    {
        Coords coords;
        Coords* p = &coords;
        p->X = 3;
        p->Y = 4;
        Console.WriteLine(p->ToString());  // output: (3, 4)
    }
}
```

Vous ne pouvez pas appliquer l'opérateur `->` à une expression de type `void*`.

Opérateur d'accès aux éléments de pointeur []

Pour une expression `p` d'un type pointeur, l'accès à un élément de pointeur au format `p[n]` est évalué comme `*(p + n)`, où `n` doit être d'un type implicitement convertible en `int`, `uint`, `long` ou `ulong`. Pour plus d'informations sur le comportement de l'opérateur `+` avec les pointeurs, consultez la section Addition ou soustraction d'une valeur intégrale dans un pointeur.

L'exemple suivant montre comment accéder à des éléments tableau avec un pointeur et l'opérateur `[]` :

C#

```
unsafe
{
    char* pointerToChars = stackalloc char[123];

    for (int i = 65; i < 123; i++)
    {
        pointerToChars[i] = (char)i;
    }

    Console.Write("Uppercase letters: ");
    for (int i = 65; i < 91; i++)
    {
        Console.Write(pointerToChars[i]);
    }
}
// Output:
// Uppercase letters: ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Dans l'exemple précédent, une expression stackalloc alloue un bloc de mémoire sur la pile.

> ⓘ **Notes**
>
> L'opérateur d'accès aux éléments de pointeur ne recherche pas les erreurs de dépassement des limites.

Vous ne pouvez pas utiliser `[]` pour l'accès aux éléments de pointeur avec une expression de type `void*`.

Vous pouvez également utiliser l'opérateur `[]` pour l'accès aux éléments de tableau ou à l'indexeur.

Opérateurs arithmétiques de pointeur

Vous pouvez effectuer les opérations arithmétiques suivantes avec des pointeurs :

- Ajouter ou soustraire une valeur intégrale dans un pointeur
- Soustraire deux pointeurs
- Incrémenter ou décrémenter un pointeur

Vous ne pouvez pas effectuer ces opérations avec des pointeurs de type `void*`.

Pour plus d'informations sur les opérations arithmétiques prises en charge avec les types numériques, consultez Opérateurs arithmétiques.

Ajout ou soustraction d'une valeur intégrale dans un pointeur

Pour un pointeur `p` de type `T*` et une expression `n` d'un type implicitement convertible en `int`, `uint`, `long` ou `ulong`, l'addition et la soustraction sont définies de la façon suivante :

- Les expressions `p + n` et `n + p` produisent un pointeur de type `T*` qui est obtenu en ajoutant `n * sizeof(T)` à l'adresse fournie par `p`.
- L'expression `p - n` produit un pointeur de type `T*` qui est obtenu en soustrayant `n * sizeof(T)` de l'adresse fournie par `p`.

L'opérateur sizeof permet d'obtenir la taille d'un type en octets.

L'exemple suivant illustre l'utilisation de l'opérateur `+` avec un pointeur :

C#

```
unsafe
{
    const int Count = 3;
    int[] numbers = new int[Count] { 10, 20, 30 };
    fixed (int* pointerToFirst = &numbers[0])
    {
        int* pointerToLast = pointerToFirst + (Count - 1);

        Console.WriteLine($"Value {*pointerToFirst} at address
{(long)pointerToFirst}");
        Console.WriteLine($"Value {*pointerToLast} at address
{(long)pointerToLast}");
    }
}
// Output is similar to:
// Value 10 at address 1818345918136
// Value 30 at address 1818345918144
```

Soustraction de pointeur

Pour deux pointeurs `p1` et `p2` de type `T*`, l'expression `p1 - p2` produit la différence entre les adresses fournies par `p1` et `p2`, divisée par `sizeof(T)`. Le type du résultat est `long`. C'est -à-dire que `p1 - p2`est calculé en tant que `((long)(p1) - (long)(p2)) / sizeof(T)`.

L'exemple suivant montre la soustraction d'un pointeur :

C#

```
unsafe
{
    int* numbers = stackalloc int[] { 0, 1, 2, 3, 4, 5 };
    int* p1 = &numbers[1];
    int* p2 = &numbers[5];
    Console.WriteLine(p2 - p1);  // output: 4
}
```

Incrémenter et décrémenter des pointeurs

L'opérateur d'incrémentation `++`ajoute 1 à son opérande de pointeur. L'opérateur de décrémentation `--`soustrait 1 de son opérande de pointeur.

Les deux opérateurs sont pris en charge sous deux formes : suffixée (`p++` et `p--`) et préfixée (`++p` et `--p`). Le résultat de `p++` et `p--` correspond à la valeur de `p` *avant* l'opération. Le résultat de `++p` et `--p` correspond à la valeur de `p` *après* l'opération.

L'exemple suivant montre le comportement des opérateurs d'incrémentation suffixés et préfixés :

C#

```
unsafe
{
    int* numbers = stackalloc int[] { 0, 1, 2 };
    int* p1 = &numbers[0];
    int* p2 = p1;
    Console.WriteLine($"Before operation: p1 - {(long)p1}, p2 -
{(long)p2}");
    Console.WriteLine($"Postfix increment of p1: {(long)(p1++)}");
    Console.WriteLine($"Prefix increment of p2: {(long)(++p2)}");
    Console.WriteLine($"After operation: p1 - {(long)p1}, p2 - {(long)p2}");
}
// Output is similar to
// Before operation: p1 - 816489946512, p2 - 816489946512
// Postfix increment of p1: 816489946512
// Prefix increment of p2: 816489946516
// After operation: p1 - 816489946516, p2 - 816489946516
```

Opérateurs de comparaison de pointeur

Vous pouvez utiliser les opérateurs `==`, `!=`, `<`, `>`, `<=` et `>=` pour comparer des opérandes de tout type de pointeur, y compris `void*`. Ces opérateurs comparent les adresses fournies par les deux opérandes comme s'il s'agissait d'entiers non signés.

Pour plus d'informations sur le comportement de ces opérateurs pour les opérandes d'autres types, consultez les articles Opérateurs d'égalité et Opérateurs de comparaison.

Priorité des opérateurs

La liste suivante présente les opérateurs relatifs aux pointeurs par ordre de précédence, de la plus élevée à la plus basse :

- Opérateurs suffixés d'incrémentation `x++` et de décrémentation `x--`, ainsi que les opérateurs `->` et `[]`
- Opérateurs préfixés d'incrémentation `++x` et de décrémentation `--x`, ainsi que les opérateurs `&` et `*`

- Opérateurs additifs `+` et `-`
- Opérateurs de comparaison `<`, `>`, `<=` et `>=`
- Opérateurs d'égalité `==` et `!=`

Utilisez des parenthèses (`()`) pour modifier l'ordre d'évaluation imposé par la précédence des opérateurs.

Pour obtenir la liste complète des opérateurs C# classés par niveau de priorité, consultez la section Priorité des opérateurs de l'article Opérateurs C#.

Capacité de surcharge de l'opérateur

Un type défini par l'utilisateur ne peut pas surcharger les opérateurs liés aux pointeurs `&`, `*`, `->` et `[]`.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Variables fixes et déplaçables
- address-of, opérateur
- Indirection de pointeur
- Accès aux membres de pointeur
- Accès aux éléments de pointeur
- Arithmétique des pointeurs
- Incrémentation et décrémentation des pointeurs
- Comparaison de pointeurs

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- Code, types de pointeurs et pointeurs de fonction non sécurisés
- mot clé unsafe
- fixed, instruction
- stackalloc, expression
- Opérateur sizeof

Opérateurs d'affectation (référence C#)

Article • 20/07/2023

L'opérateur d'affectation `=` affecte la *valeur* de son opérande droit à une variable, une propriété ou un élément indexeur donné par son opérande gauche. Le résultat d'une expression d'assignation est la valeur assignée à l'opérande de gauche. L'opérande de droite doit être du même type que l'opérande de gauche, ou implicitement convertible vers le type de l'opérande de gauche.

L'opérateur d'affectation `=` est associatif droit, c'est-à-dire une expression du formulaire

C#

```
a = b = c
```

est évaluée comme étant

C#

```
a = (b = c)
```

L'exemple suivant illustre l'utilisation de l'opérateur d'assignation avec une variable locale, une propriété et un élément d'indexeur comme opérande de gauche :

C#

```
var numbers = new List<double>() { 1.0, 2.0, 3.0 };

Console.WriteLine(numbers.Capacity);
numbers.Capacity = 100;
Console.WriteLine(numbers.Capacity);
// Output:
// 4
// 100

int newFirstElement;
double originalFirstElement = numbers[0];
newFirstElement = 5;
numbers[0] = newFirstElement;
Console.WriteLine(originalFirstElement);
Console.WriteLine(numbers[0]);
// Output:
// 1
// 5
```

L'opérande gauche d'une affectation reçoit la *valeur* de l'opérande droit. Lorsque les opérandes sont de types valeur, l'affectation copie le contenu de l'opérande droit. Lorsque les opérandes sont de types de référence, l'affectation copie la référence à l'objet.

Il s'agit de *l'affectation de valeur* : la valeur est affectée.

affectation ref

Affectation ref `= ref` fait de son opérande de gauche un alias de l'opérande de droite, comme le montre l'exemple suivant :

C#

```
void Display(double[] s) => Console.WriteLine(string.Join(" ", s));

double[] arr = { 0.0, 0.0, 0.0 };
Display(arr);

ref double arrayElement = ref arr[0];
arrayElement = 3.0;
Display(arr);

arrayElement = ref arr[arr.Length - 1];
arrayElement = 5.0;
Display(arr);
// Output:
// 0 0 0
// 3 0 0
// 3 0 5
```

Dans l'exemple précédent, la variable de référence locale `arrayElement` est initialisée en tant qu'alias au premier élément du tableau. Ensuite, elle est `ref` réassignée pour faire référence au dernier élément de tableau. Comme il s'agit d'un alias, lorsque vous mettez à jour sa valeur avec un opérateur d'affectation ordinaire `=`, l'élément de tableau correspondant est également mis à jour.

L'opérande de gauche de l'affectation `ref` peut être une variable de référence locale, un ref champ et un paramètre de la méthode ref, out ou in. Les deux opérandes doivent être du même type.

Assignation composée

Pour un opérateur binaire `op`, une expression d'assignation composée du formulaire

```C#
x op= y
```

équivaut à :

```C#
x = x op y
```

sauf que `x` n'est évalué qu'une seule fois.

L'assignation composée est prise en charge par les opérateurs arithmétiques, logiques booléens et logiques au niveau du bit et du décalage.

Attribution de fusion Null

Vous pouvez utiliser l'opérateur d'affectation null-coalescing `??=` pour affecter la valeur de l'opérande droit à l'opérande gauche uniquement si l'opérande gauche a la valeur `null`. Pour plus d'informations, consultez l'article ?? et ?? = opérateurs.

Capacité de surcharge de l'opérateur

Un type défini par l'utilisateur ne peut pas surcharger l'opérateur d'affectation. Toutefois, il peut définir une conversion implicite vers un autre type. Ainsi, la valeur d'un type défini par l'utilisateur peut être assignée à une variable, une propriété ou un élément d'indexeur d'un autre type. Pour plus d'informations, consultez Opérateurs de conversion définie par l'utilisateur.

Un type défini par l'utilisateur ne peut pas surcharger explicitement un opérateur d'assignation composée. Toutefois, si un type défini par l'utilisateur surcharge un opérateur binaire `op`, l'opérateur `op=`, s'il existe, est également implicitement surchargé.

spécification du langage C#

Pour plus d'informations, voir la section Opérateurs d'assignation de la spécification du langage C#.

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- ref, mot clé
- Utiliser l'attribution composée (règles de style IDE0054 et IDE0074)

Expressions lambda et fonctions anonymes

Article • 03/06/2023

Vous utilisez une *expression lambda* pour créer une fonction anonyme. Utilisez l'opérateur de déclaration lambda `=>` pour séparer la liste des paramètres de l'expression lambda de son corps. Une expression lambda peut être de l'une des deux formes suivantes :

- Expression lambda qui a une expression comme corps :

  ```
  C#

  (input-parameters) => expression
  ```

- Instruction lambda qui a un bloc d'instructions comme corps :

  ```
  C#

  (input-parameters) => { <sequence-of-statements> }
  ```

Pour créer une expression lambda, vous spécifiez des paramètres d'entrée (le cas échéant) à gauche de l'opérateur lambda et une expression ou un bloc d'instructions de l'autre côté.

Toute expression lambda peut être convertie en type délégué. Le type délégué vers lequel une expression lambda peut être convertie est défini par les types de ses paramètres et de sa valeur de retour. Si une expression lambda ne retourne pas de valeur, elle peut être convertie en l'un des types délégués `Action` ; sinon, elle peut être convertie en l'un des types délégués `Func`. Par exemple, une expression lambda qui a deux paramètres et qui ne retourne aucune valeur peut être convertie en un délégué Action<T1,T2>. Une expression lambda qui a un paramètre et qui retourne une valeur peut être convertie en un délégué Func<T,TResult>. Dans l'exemple suivant, l'expression lambda `x => x * x`, qui spécifie un paramètre nommé `x` et retourne la valeur du carré de `x`, est assignée à une variable d'un type délégué :

```
C#

Func<int, int> square = x => x * x;
Console.WriteLine(square(5));
// Output:
// 25
```

Les expressions lambda peuvent également être converties en types d'arborescence d'expression, comme le montre l'exemple suivant :

C#

```
System.Linq.Expressions.Expression<Func<int, int>> e = x => x * x;
Console.WriteLine(e);
// Output:
// x => (x * x)
```

Vous pouvez utiliser des expressions lambda dans tout code qui requiert des instances de types délégués ou d'arborescences d'expressions, par exemple en tant qu'argument de la méthode Task.Run(Action) pour passer le code qui doit être exécuté en arrière-plan. Vous pouvez également utiliser des expressions lambda lorsque vous écrivez LINQ en C#, comme l'illustre l'exemple suivant :

C#

```
int[] numbers = { 2, 3, 4, 5 };
var squaredNumbers = numbers.Select(x => x * x);
Console.WriteLine(string.Join(" ", squaredNumbers));
// Output:
// 4 9 16 25
```

Lorsque la méthode Enumerable.Select est appelée avec une syntaxe fondée sur une méthode dans la classe System.Linq.Enumerable (par exemple, dans LINQ to Objects et LINQ to XML), le paramètre est un type délégué System.Func<T,TResult>. Si la méthode Queryable.Select est appelée dans la classe System.Linq.Queryable (par exemple, dans LINQ to SQL), le paramètre est un type d'arborescence d'expression Expression<Func<TSource,TResult>>. Dans les deux cas, vous pouvez utiliser la même expression lambda pour spécifier la valeur de paramètre. Les deux appels `Select` semblent ainsi semblables alors qu'en fait le type des objets créés à partir des lambdas est différent.

Expressions lambdas

Une expression lambda comportant une expression à droite de l'opérateur `=>` est appelée *expression lambda*. Une expression lambda retourne le résultat de l'expression et prend la forme de base suivante :

C#

```
(input-parameters) => expression
```

Le corps d'une expression lambda peut se composer d'un appel de méthode. Cependant, si vous créez des arborescences d'expressions évaluées en dehors du contexte du Common Language Runtime (CLR) .NET, comme dans SQL Server, vous ne devez pas utiliser d'appels de méthode dans les expressions lambda. Les méthodes n'ont aucune signification en dehors du contexte du CLR (Common Language Runtime) .NET.

Instructions lambda

Une instruction lambda ressemble à une expression lambda, mais l'instruction ou les instructions sont mises entre accolades :

C#

```
(input-parameters) => { <sequence-of-statements> }
```

Le corps d'une instruction lambda peut se composer d'un nombre illimité d'instructions ; toutefois, en pratique, leur nombre est généralement de deux ou trois.

C#

```
Action<string> greet = name =>
{
    string greeting = $"Hello {name}!";
    Console.WriteLine(greeting);
};
greet("World");
// Output:
// Hello World!
```

Vous ne pouvez pas utiliser des instructions lambda pour créer des arborescences d'expressions.

Paramètres d'entrée d'une expression lambda

Vous placez les paramètres d'entrée d'une expression lambda entre parenthèses. Spécifiez des paramètres d'entrée de zéro avec des parenthèses vides :

C#

```
Action line = () => Console.WriteLine();
```

Si une expression lambda n'a qu'un seul paramètre d'entrée, les parenthèses sont facultatives :

C#

```
Func<double, double> cube = x => x * x * x;
```

Deux paramètres d'entrée ou plus sont séparés par des virgules :

C#

```
Func<int, int, bool> testForEquality = (x, y) => x == y;
```

Parfois, le compilateur ne peut pas déduire les types de paramètres d'entrée. Vous pouvez spécifier les types explicitement comme dans l'exemple suivant :

C#

```
Func<int, string, bool> isTooLong = (int x, string s) => s.Length > x;
```

Les types de paramètres d'entrée doivent être tous explicites ou tous implicites ; sinon, une erreur de compilateur CS0748 se produit.

À compter de C# 9.0, vous pouvez utiliser des abandons pour spécifier deux paramètres d'entrée ou plus d'une expression lambda qui ne sont pas utilisés dans l'expression :

C#

```
Func<int, int, int> constant = (_, _) => 42;
```

Les paramètres d'abandon lambda peuvent être utiles lorsque vous utilisez une expression lambda pour fournir un gestionnaire d'événements.

> ⓘ **Notes**
>
> Pour la compatibilité descendante, si un seul paramètre d'entrée est nommé `_`, puis, dans une expression lambda, `_` est traité comme le nom de ce paramètre.

À partir de C# 12, vous pouvez fournir des *valeurs par défaut* pour les paramètres des expressions lambda. La syntaxe et les restrictions relatives aux valeurs de paramètre par défaut sont les mêmes que pour les méthodes et fonctions locales. L'exemple suivant déclare une expression lambda avec un paramètre par défaut, puis l'appelle une fois à l'aide de la valeur par défaut et une fois avec deux paramètres explicites :

C#

```
var IncrementBy = (int source, int increment = 1) => source + increment;

Console.WriteLine(IncrementBy(5)); // 6
Console.WriteLine(IncrementBy(5, 2)); // 7
```

Vous pouvez également déclarer des expressions lambda avec des tableaux `params` en tant que paramètres :

C#

```
var sum = (params int[] values) =>
{
    int sum = 0;
    foreach (var value in values)
        sum += value;

    return sum;
};

var empty = sum();
Console.WriteLine(empty); // 0

var sequence = new[] { 1, 2, 3, 4, 5 };
var total = sum(sequence);
Console.WriteLine(total); // 15
```

Dans le cadre de ces mises à jour, quand un groupe de méthodes qui a un paramètre par défaut est affecté à une expression lambda, cette expression lambda a également le même paramètre par défaut. Un groupe de méthodes avec un paramètre de tableau `params` peut également être affecté à une expression lambda.

Les expressions lambda avec des paramètres par défaut ou des tableaux `params` en tant que paramètres n'ont pas de types naturels qui correspondent aux types `Func<>` ou `Action<>`. Toutefois, vous pouvez définir des types délégués qui incluent des valeurs de paramètre par défaut :

C#

```
delegate int IncrementByDelegate(int source, int increment = 1);
delegate int SumDelegate(params int[] values);
```

Vous pouvez également utiliser des variables implicitement typées avec des déclarations `var` pour définir le type délégué. Le compilateur synthétise le type délégué approprié.

Pour plus d'informations, consultez la spécification de fonctionnalités des paramètres par défaut pour les expressions lambda.

Lambdas asynchrones

Vous pouvez facilement créer des expressions et des instructions lambda qui incorporent le traitement asynchrone à l'aide des mots clés async et await . Par exemple, l'exemple Windows Forms suivant contient un gestionnaire d'événements qui appelle et attend une méthode async `ExampleMethodAsync`.

C#

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += button1_Click;
    }

    private async void button1_Click(object sender, EventArgs e)
    {
        await ExampleMethodAsync();
        textBox1.Text += "\r\nControl returned to Click event handler.\n";
    }

    private async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous
process.
        await Task.Delay(1000);
    }
}
```

Vous pouvez ajouter le même gestionnaire d'événements en utilisant une expression lambda asynchrone. Pour ajouter ce gestionnaire, ajoutez un modificateur `async` avant la liste des paramètres lambda, comme dans l'exemple suivant :

C#

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += async (sender, e) =>
        {
            await ExampleMethodAsync();
            textBox1.Text += "\r\nControl returned to Click event
handler.\n";
        };
    }

    private async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous
process.
        await Task.Delay(1000);
    }
}
```

Pour plus d'informations sur la création et l'utilisation des méthodes asyncrones, consultez Programmation asynchrone avec async et await.

Expressions lambda et tuples

Le langage C# fournit une prise en charge intégrée des tuples. Vous pouvez fournir un tuple comme argument à une expression lambda, et votre expression lambda peut aussi retourner un tuple. Dans certains cas, le compilateur C# utilise l'inférence de type pour déterminer les types des composants du tuple.

Vous définissez un tuple en plaçant entre des parenthèses une liste de ses composants avec des virgules comme séparateur. L'exemple suivant utilise un tuple à trois composants pour passer une séquence de nombres à une expression lambda, qui double chaque valeur et retourne un tuple à trois composants contenant le résultat des multiplications.

C#

```
Func<(int, int, int), (int, int, int)> doubleThem = ns => (2 * ns.Item1, 2 *
ns.Item2, 2 * ns.Item3);
var numbers = (2, 3, 4);
var doubledNumbers = doubleThem(numbers);
Console.WriteLine($"The set {numbers} doubled: {doubledNumbers}");
// Output:
// The set (2, 3, 4) doubled: (4, 6, 8)
```

En règle générale, les champs d'un tuple sont nommés `Item1`, `Item2` et ainsi de suite. Vous pouvez cependant définir un tuple avec des composants nommés, comme dans l'exemple suivant.

C#

```
Func<(int n1, int n2, int n3), (int, int, int)> doubleThem = ns => (2 *
ns.n1, 2 * ns.n2, 2 * ns.n3);
var numbers = (2, 3, 4);
var doubledNumbers = doubleThem(numbers);
Console.WriteLine($"The set {numbers} doubled: {doubledNumbers}");
```

Pour plus d'informations sur les tuples C#, consultez les types de Tuple.

Lambdas avec les opérateurs de requête standard

LINQ to Objects, parmi d'autres implémentations, a un paramètre d'entrée dont le type fait partie de la famille Func<TResult> de délégués génériques. Ces délégués utilisent des paramètres de type pour définir le nombre et le type des paramètres d'entrée, ainsi que le type de retour du délégué. Les délégués `Func` sont utiles pour encapsuler des expressions définies par l'utilisateur qui sont appliquées à chaque élément dans un ensemble de données sources. Prenons par exemple le type délégué Func<T,TResult> :

C#

```
public delegate TResult Func<in T, out TResult>(T arg)
```

Ce délégué peut être instancié comme une instance `Func<int, bool>`, où `int` est un paramètre d'entrée et `bool` la valeur de retour. La valeur de retour est toujours spécifiée dans le dernier paramètre de type. Par exemple, `Func<int, string, bool>` définit un délégué avec deux paramètres d'entrée, `int` et `string`, et un type de retour `bool`. Le délégué `Func` suivant, lorsqu'il est appelé, retourne une valeur booléenne indiquant si le paramètre d'entrée est égal à cinq :

C#

```
Func<int, bool> equalsFive = x => x == 5;
bool result = equalsFive(4);
Console.WriteLine(result);   // False
```

Vous pouvez aussi fournir une expression lambda quand le type d'argument est Expression<TDelegate>, par exemple, dans les opérateurs de requête standard définis dans le type Queryable. Quand vous spécifiez un argument Expression<TDelegate>, l'expression lambda est compilée en arborescence de l'expression.

L'exemple suivant utilise l'opérateur de requête standard Count :

C#

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
int oddNumbers = numbers.Count(n => n % 2 == 1);
Console.WriteLine($"There are {oddNumbers} odd numbers in {string.Join(" ",
numbers)}");
```

Le compilateur peut déduire le type du paramètre d'entrée, ou vous pouvez également le spécifier explicitement. Cette expression lambda particulière compte les entiers (`n`) qui, lorsqu'ils sont divisés par deux, ont un reste de 1.

L'exemple suivant produit une séquence qui contient tous les éléments du tableau `numbers` précédant le 9, car c'est le premier nombre de la séquence qui ne remplit pas la condition :

C#

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var firstNumbersLessThanSix = numbers.TakeWhile(n => n < 6);
Console.WriteLine(string.Join(" ", firstNumbersLessThanSix));
// Output:
// 5 4 1 3
```

L'exemple suivant spécifie plusieurs paramètres d'entrée en les plaçant entre parenthèses. La méthode retourne tous les éléments du tableau `numbers` jusqu'à ce qu'il trouve un nombre dont la valeur est inférieure à sa position ordinale dans le tableau :

C#

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var firstSmallNumbers = numbers.TakeWhile((n, index) => n >= index);
Console.WriteLine(string.Join(" ", firstSmallNumbers));
// Output:
// 5 4
```

Vous n'utilisez pas d'expressions lambda directement dans les expressions de requête, mais vous pouvez les utiliser dans les appels de méthode dans les expressions de requête, comme l'illustre l'exemple suivant :

C#

```
var numberSets = new List<int[]>
{
    new[] { 1, 2, 3, 4, 5 },
    new[] { 0, 0, 0 },
    new[] { 9, 8 },
    new[] { 1, 0, 1, 0, 1, 0, 1, 0 }
};

var setsWithManyPositives =
    from numberSet in numberSets
    where numberSet.Count(n => n > 0) > 3
    select numberSet;

foreach (var numberSet in setsWithManyPositives)
{
    Console.WriteLine(string.Join(" ", numberSet));
}
// Output:
// 1 2 3 4 5
// 1 0 1 0 1 0 1 0
```

Inférence de type et expressions lambda

Il n'est généralement pas nécessaire, avec les expressions lambda, de spécifier un type pour les paramètres d'entrée, car le compilateur peut le déduire du corps de l'expression, des types de paramètres et d'autres facteurs, comme le décrit la spécification du langage C#. Pour la plupart des opérateurs de requête standard, la première entrée est le type des éléments dans la séquence source. Si vous interrogez un `IEnumerable<Customer>`, la variable d'entrée est déduite comme un objet `Customer`, ce qui signifie que vous avez accès à ses méthodes et propriétés :

C#

```
customers.Where(c => c.City == "London");
```

Voici les règles générales de l'inférence de type pour les expressions lambda :

- Le lambda doit contenir le même nombre de paramètres que le type délégué.
- Chaque paramètre d'entrée dans le lambda doit être implicitement convertible en son paramètre de délégué correspondant.
- La valeur de retour du lambda (le cas échéant) doit être implicitement convertible en type de retour du délégué.

Type naturel d'une expression lambda

Une expression lambda en soi n'a pas de type, car le système de type commun n'a pas de concept intrinsèque d'« expression lambda ». Toutefois, il est parfois pratique de parler de manière informelle du « type » d'une expression lambda. Ce « type » informel fait référence au type délégué ou au type Expression vers lequel l'expression lambda est convertie.

À compter de C# 10, une expression lambda peut avoir un *type*naturel. Au lieu de vous forcer à déclarer un type délégué, tel que `Func<...>` ou `Action<...>` pour une expression lambda, le compilateur peut déduire le type délégué de l'expression lambda. Observez par exemple la déclaration suivante :

```
C#

var parse = (string s) => int.Parse(s);
```

Le compilateur peut déduire que `parse` est un `Func<string, int>`. Le compilateur choisit un délégué `Func` ou `Action` disponible, s'il en existe un. Sinon, il synthétise un type délégué. Par exemple, le type délégué est synthétisé si l'expression lambda a des paramètres `ref`. Lorsqu'une expression lambda a un type naturel, elle peut être affectée à un type moins explicite, tel que System.Object ou System.Delegate :

```
C#

object parse = (string s) => int.Parse(s);   // Func<string, int>
Delegate parse = (string s) => int.Parse(s); // Func<string, int>
```

Les groupes de méthodes (autrement dit, les noms de méthodes sans listes de paramètres) avec exactement une surcharge ont un type naturel :

```
C#

var read = Console.Read; // Just one overload; Func<int> inferred
var write = Console.Write; // ERROR: Multiple overloads, can't choose
```

Si vous affectez une expression lambda à System.Linq.Expressions.LambdaExpression, ou System.Linq.Expressions.Expression, et que l'expression lambda a un type délégué naturel, l'expression a un type naturel de System.Linq.Expressions.Expression<TDelegate>, avec le type délégué naturel utilisé comme argument pour le paramètre de type :

```
```

```C#
LambdaExpression parseExpr = (string s) => int.Parse(s); //
Expression<Func<string, int>>
Expression parseExpr = (string s) => int.Parse(s);       //
Expression<Func<string, int>>
```

Toutes les expressions lambda n'ont pas de type naturel. Prenons la déclaration suivante :

```C#
var parse = s => int.Parse(s); // ERROR: Not enough type info in the lambda
```

Le compilateur ne peut pas déduire un type de paramètre pour `s`. Lorsque le compilateur ne peut pas déduire un type naturel, vous devez déclarer le type :

```C#
Func<string, int> parse = s => int.Parse(s);
```

Type de retour explicite

En règle générale, le type de retour d'une expression lambda est évident et déduit. Pour certaines expressions qui ne fonctionnent pas :

```C#
var choose = (bool b) => b ? 1 : "two"; // ERROR: Can't infer return type
```

À compter de C#10, vous pouvez spécifier le type de retour d'une expression lambda avant les paramètres d'entrée. Lorsque vous spécifiez un type de retour explicite, vous devez mettre entre parenthèses les paramètres d'entrée :

```C#
var choose = object (bool b) => b ? 1 : "two"; // Func<bool, object>
```

Attributs

À compter de C#10, vous pouvez ajouter des attributs à une expression lambda et à ses paramètres. L'exemple suivant montre comment ajouter des attributs à une expression lambda :

```
C#
```

```
Func<string?, int?> parse = [ProvidesNullCheck] (s) => (s is not null) ?
int.Parse(s) : null;
```

Vous pouvez également ajouter des attributs aux paramètres d'entrée ou à la valeur de retour, comme l'illustre l'exemple suivant :

```
C#
```

```
var concat = ([DisallowNull] string a, [DisallowNull] string b) => a + b;
var inc = [return: NotNullifNotNull(nameof(s))] (int? s) => s.HasValue ? s++
: null;
```

Comme le montrent les exemples précédents, vous devez mettre entre parenthèses les paramètres d'entrée lorsque vous ajoutez des attributs à une expression lambda ou à ses paramètres.

> **ⓘ Important**
>
> Les expressions lambda sont appelées par le biais du type de délégué sous-jacent. C'est différent des méthodes et des fonctions locales. La méthode `Invoke` du délégué ne vérifie pas les attributs sur l'expression lambda. Les attributs n'ont aucun effet lorsque l'expression lambda est appelée. Les attributs sur les expressions lambda sont utiles pour l'analyse du code et peuvent être découverts via la réflexion. L'une des conséquences de cette décision est que **System.Diagnostics.ConditionalAttribute** ne peut pas être appliqué à une expression lambda.

Capture des variables externes et de l'étendue variable dans les expressions lambda

Les expressions lambda peuvent faire référence à des *variables externes*. Ces *variables externes* sont des variables dans l'étendue de la méthode qui définit l'expression lambda, ou dans l'étendue du type qui contient l'expression lambda. Les variables capturées de cette manière sont stockées pour une utilisation dans l'expression lambda, même si les variables se trouvent en dehors de la portée et sont récupérées par le

garbage collector. Une variable externe doit être assignée de manière précise pour pouvoir être utilisée dans une expression lambda. L'exemple suivant illustre ces règles :

C#

```
public static class VariableScopeWithLambdas
{
    public class VariableCaptureGame
    {
        internal Action<int>? updateCapturedLocalVariable;
        internal Func<int, bool>? isEqualToCapturedLocalVariable;

        public void Run(int input)
        {
            int j = 0;

            updateCapturedLocalVariable = x =>
            {
                j = x;
                bool result = j > input;
                Console.WriteLine($"{j} is greater than {input}: {result}");
            };

            isEqualToCapturedLocalVariable = x => x == j;

            Console.WriteLine($"Local variable before lambda invocation:
{j}");
            updateCapturedLocalVariable(10);
            Console.WriteLine($"Local variable after lambda invocation:
{j}");
        }
    }

    public static void Main()
    {
        var game = new VariableCaptureGame();

        int gameInput = 5;
        game.Run(gameInput);

        int jTry = 10;
        bool result = game.isEqualToCapturedLocalVariable!(jTry);
        Console.WriteLine($"Captured local variable is equal to {jTry}:
{result}");

        int anotherJ = 3;
        game.updateCapturedLocalVariable!(anotherJ);

        bool equalToAnother = game.isEqualToCapturedLocalVariable(anotherJ);
        Console.WriteLine($"Another lambda observes a new value of captured
variable: {equalToAnother}");
    }
    // Output:
    // Local variable before lambda invocation: 0
```

```
    // 10 is greater than 5: True
    // Local variable after lambda invocation: 10
    // Captured local variable is equal to 10: True
    // 3 is greater than 5: False
    // Another lambda observes a new value of captured variable: True
}
```

Les règles suivantes s'appliquent à la portée des variables dans les expressions lambda :

- Une variable capturée ne sera pas collectée par le garbage collector tant que le délégué qui le référence devient éligible pour la garbage collection.
- Les variables introduites dans une expression lambda ne sont pas visibles dans la méthode englobante.
- Une expression lambda ne peut pas capturer directement un paramètre in, ref ou out à partir de la méthode englobante.
- Une instruction return dans une expression lambda ne provoque pas le retour de la méthode englobante.
- Une expression lambda ne peut pas contenir d'instruction goto, breakou continue si la cible de cette instruction jump se trouve en dehors du bloc d'expression lambda. Il est également incorrect d'avoir une instruction de saut en dehors du bloc de l'expression lambda si la cible se trouve à l'intérieur du bloc.

À compter de C# 9.0, vous pouvez appliquer le modificateur `static` à une expression lambda pour empêcher la capture involontaire de variables locales ou d'état d'instance par l'expression lambda :

C#

```
Func<double, double> square = static x => x * x;
```

Une expression lambda statique ne peut pas capturer les variables locales ou l'état d'instance à partir d'étendues englobantes, mais peut référencer des membres statiques et des définitions constantes.

spécification du langage C#

Pour plus d'informations, consultez la section Expressions de fonction anonyme de la spécification du langage C#.

Pour plus d'informations sur les fonctionnalités ajoutées dans C# 9.0 et versions ultérieures, consultez les notes de proposition de fonctionnalités suivantes :

- Paramètres d'abandon lambda (C# 9.0)

- Fonctions anonymes statiques (C# 9.0)
- Améliorations lambda (C# 10)

Voir aussi

- Utiliser une fonction locale au lieu d'une fonction lambda (règle de style IDE0039)
- Informations de référence sur C#
- Opérateurs et expressions C#
- LINQ (Language Integrated Query)
- Arborescences de l'expression
- Fonctions locales et expressions lambda
- Exemples de requêtes LINQ
- Exemple XQuery
- 101 exemples LINQ

Critères spéciaux : les expressions `is` et `switch` et les opérateurs `and`, `or` et `not` dans les modèles

Article • 07/04/2023

Vous utilisez l'expression is, l'instruction switch et l'expression switch pour faire correspondre une expression d'entrée à un nombre quelconque de caractéristiques. C# prend en charge plusieurs modèles, notamment la déclaration, le type, la constante, la relation, la propriété, la liste, var et discard. Les modèles peuvent être combinés à l'aide des mots clés logiques booléens `and`, `or` et `not`.

Les expressions et instructions C# suivantes prennent en charge les critères spéciaux :

- is, expression
- Instruction switch
- switch, expression

Dans ces constructions, vous pouvez faire correspondre une expression d'entrée à l'un des modèles suivants :

- Modèle de déclaration : pour vérifier le type d'exécution d'une expression et, si une correspondance réussit, affecter un résultat d'expression à une variable déclarée.
- Modèle de type : pour vérifier le type d'exécution d'une expression. Introduit dans C# 9.0.
- Modèle de constante : pour tester si un résultat d'expression est égal à une constante spécifiée.
- Modèles relationnels : pour comparer un résultat d'expression avec une constante spécifiée. Introduit dans C# 9.0.
- Modèles logiques : pour tester si une expression correspond à une combinaison logique de modèles. Introduit dans C# 9.0.
- Modèle de propriété : pour tester si les propriétés ou les champs d'une expression correspondent à des modèles imbriqués.
- Modèle positionnel : pour déconstruire un résultat d'expression et tester si les valeurs résultantes correspondent à des modèles imbriqués.
- Modèle var : pour faire correspondre n'importe quelle expression et affecter son résultat à une variable déclarée.
- Modèle Discard : pour faire correspondre n'importe quelle expression.
- Modèles de liste : pour tester si les éléments de séquence correspondent aux modèles imbriqués correspondants. Introduit dans C# 11.

Les modèles logiques, de propriété, positionnels et de liste sont des modèles *récursifs*. Autrement dit, ils peuvent contenir des *modèles imbriqués*.

Pour voir un exemple d'utilisation de ces modèles afin de créer un algorithme piloté par les données, consultez le Tutoriel : Utiliser des critères spéciaux pour générer des algorithmes pilotés par type et pilotés par des données.

Modèles de déclaration et de type

Utilisez des modèles de déclaration et de type pour vérifier si le type d'exécution d'une expression est compatible avec un type donné. Avec un modèle de déclaration, vous pouvez également déclarer une nouvelle variable locale. Lorsqu'un modèle de déclaration correspond à une expression, cette variable reçoit un résultat d'expression convertie, comme l'illustre l'exemple suivant :

C#

```
object greeting = "Hello, World!";
if (greeting is string message)
{
    Console.WriteLine(message.ToLower());  // output: hello, world!
}
```

Un *modèle de déclaration* avec le type `T` correspond à une expression lorsqu'un résultat d'expression n'est pas null et que l'une des conditions suivantes est vraie :

- Le type d'exécution d'un résultat d'expression est `T`.
- Le type d'exécution d'un résultat d'expression dérive du type `T`, implémente l'interface `T`, ou une autre conversion de référence implicite en `T` existe à partir de celui-ci. L'exemple suivant illustre deux cas où cette condition a la valeur true :

  C#

  ```
  var numbers = new int[] { 10, 20, 30 };
  Console.WriteLine(GetSourceLabel(numbers));  // output: 1

  var letters = new List<char> { 'a', 'b', 'c', 'd' };
  Console.WriteLine(GetSourceLabel(letters));  // output: 2

  static int GetSourceLabel<T>(IEnumerable<T> source) => source switch
  {
      Array array => 1,
      ICollection<T> collection => 2,
      _ => 3,
  };
  ```

Dans l'exemple précédent, au premier appel à la méthode `GetSourceLabel`, le premier modèle correspond à une valeur d'argument, car le type d'exécution `int[]` de l'argument dérive du type Array. Au deuxième appel à la méthode `GetSourceLabel`, le type d'exécution List<T> de l'argument ne dérive pas du type Array, mais implémente l'interface ICollection<T>.

- Le type d'exécution d'un résultat d'expression est un type de valeur pouvant accepter la valeur Null avec le type sous-jacent `T`.
- Il existe une conversion boxing ou unboxing en type `T` à partir du type d'exécution d'un résultat d'expression.

L'exemple suivant illustre les deux dernières conditions :

C#

```
int? xNullable = 7;
int y = 23;
object yBoxed = y;
if (xNullable is int a && yBoxed is int b)
{
    Console.WriteLine(a + b);  // output: 30
}
```

Si vous souhaitez vérifier uniquement le type d'une expression, vous pouvez utiliser un caractère `_` de type Discard à la place du nom d'une variable, comme l'illustre l'exemple suivant :

C#

```
public abstract class Vehicle {}
public class Car : Vehicle {}
public class Truck : Vehicle {}

public static class TollCalculator
{
    public static decimal CalculateToll(this Vehicle vehicle) => vehicle
switch
    {
        Car _ => 2.00m,
        Truck _ => 7.50m,
        null => throw new ArgumentNullException(nameof(vehicle)),
        _ => throw new ArgumentException("Unknown type of a vehicle",
nameof(vehicle)),
    };
}
```

À compter de C# 9.0, à cet effet, vous pouvez utiliser un *modèle de type*, comme l'illustre l'exemple suivant :

```
C#
```

```
public static decimal CalculateToll(this Vehicle vehicle) => vehicle switch
{
    Car => 2.00m,
    Truck => 7.50m,
    null => throw new ArgumentNullException(nameof(vehicle)),
    _ => throw new ArgumentException("Unknown type of a vehicle",
nameof(vehicle)),
};
```

À l'instar d'un modèle de déclaration, un modèle de type correspond à une expression lorsqu'un résultat d'expression est non-null et que son type d'exécution satisfait à l'une des conditions répertoriées ci-dessus.

Pour rechercher des valeurs non-null, vous pouvez utiliser un modèle de constante `null`nié, comme l'illustre l'exemple suivant :

```
C#
```

```
if (input is not null)
{
    // ...
}
```

Pour plus d'informations, consultez les sections Modèle de déclaration et Modèle de type des notes de proposition de fonctionnalités.

Modèle de constante

Vous utilisez un *modèle de constante* pour tester si un résultat d'expression est égal à une constante spécifiée, comme l'illustre l'exemple suivant :

```
C#
```

```
public static decimal GetGroupTicketPrice(int visitorCount) => visitorCount
switch
{
    1 => 12.0m,
    2 => 20.0m,
    3 => 27.0m,
    4 => 32.0m,
    0 => 0.0m,
    _ => throw new ArgumentException($"Not supported number of visitors:
```

```
{visitorCount}", nameof(visitorCount)),
};
```

Dans un modèle constant, vous pouvez utiliser n'importe quelle expression constante, par exemple :

- un littéral numérique entier ou à virgule flottante
- char
- littéral de chaîne.
- une valeur booléenne `true` ou `false`
- valeur enum
- le nom d'un champ const déclaré ou local
- `null`

L'expression doit être un type convertible en type constant, à une exception près : une expression dont le type est `Span<char>` ou `ReadOnlySpan<char>` peut être mise en correspondance avec des chaînes constantes en C# 11 et versions ultérieures.

Utilisez un modèle constant pour rechercher `null`, comme l'illustre l'exemple suivant :

C#

```
if (input is null)
{
    return;
}
```

Le compilateur garantit qu'aucun opérateur d'égalité `==` surchargé par l'utilisateur n'est appelé lorsque l'expression `x is null` est évaluée.

À compter de C# 9.0, vous pouvez utiliser un modèle de constante nié `null` pour rechercher des valeurs non-null, comme l'illustre l'exemple suivant :

C#

```
if (input is not null)
{
    // ...
}
```

Pour plus d'informations, consultez la section Modèle de constante de la note de proposition de fonctionnalités.

Modèles relationnels

À compter de C# 9.0, vous utilisez un *modèle relationnel* pour comparer un résultat d'expression à une constante, comme l'illustre l'exemple suivant :

C#

```
Console.WriteLine(Classify(13));  // output: Too high
Console.WriteLine(Classify(double.NaN));  // output: Unknown
Console.WriteLine(Classify(2.4));  // output: Acceptable

static string Classify(double measurement) => measurement switch
{
    < -4.0 => "Too low",
    > 10.0 => "Too high",
    double.NaN => "Unknown",
    _ => "Acceptable",
};
```

Dans un modèle relationnel, vous pouvez utiliser l'un des opérateurs relationnels , , `<`, `>`, `<=` ou `>=`. La partie droite d'un modèle relationnel doit être une expression constante. L'expression constante peut être d'un type entier, à virgule flottante, char ou énumération.

Pour vérifier si un résultat d'expression se trouve dans une certaine plage, faites-le correspondre à un modèle and conjonctif, comme l'illustre l'exemple suivant :

C#

```
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 3, 14)));  // output:
spring
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 7, 19)));  // output:
summer
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 2, 17)));  // output:
winter

static string GetCalendarSeason(DateTime date) => date.Month switch
{
    >= 3 and < 6 => "spring",
    >= 6 and < 9 => "summer",
    >= 9 and < 12 => "autumn",
    12 or (>= 1 and < 3) => "winter",
    _ => throw new ArgumentOutOfRangeException(nameof(date), $"Date with
unexpected month: {date.Month}."),
};
```

Si un résultat d'expression est `null` ou ne parvient pas à se convertir en type de constante par une conversion nullable ou unboxing, un modèle relationnel ne

correspond pas à une expression.

Pour plus d'informations, consultez la section Modèles relationnels de la note de proposition de fonctionnalités.

Modèles logiques

À compter de C# 9.0, vous utilisez les combinateurs de modèles `not`, `and` et `or` pour créer les *modèles logiques* suivants :

- Modèle de *négation* `not` qui correspond à une expression lorsque le modèle nié ne correspond pas à l'expression. L'exemple suivant montre comment annuler un modèle de constante `null` pour vérifier si une expression n'est pas null :

  C#

  ```
  if (input is not null)
  {
      // ...
  }
  ```

- Modèle *conjonctif* `and` qui correspond à une expression lorsque les deux modèles correspondent à l'expression. L'exemple suivant montre comment combiner des modèles relationnels pour vérifier si une valeur se trouve dans une certaine plage :

  C#

  ```
  Console.WriteLine(Classify(13));  // output: High
  Console.WriteLine(Classify(-100));  // output: Too low
  Console.WriteLine(Classify(5.7));  // output: Acceptable

  static string Classify(double measurement) => measurement switch
  {
      < -40.0 => "Too low",
      >= -40.0 and < 0 => "Low",
      >= 0 and < 10.0 => "Acceptable",
      >= 10.0 and < 20.0 => "High",
      >= 20.0 => "Too high",
      double.NaN => "Unknown",
  };
  ```

- Modèle *disjonctif* `or` qui correspond à une expression lorsque l'un ou l'autre modèle correspond à l'expression, comme l'illustre l'exemple suivant :

  C#

```
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 1, 19)));  //
output: winter
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 10, 9)));  //
output: autumn
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 5, 11)));  //
output: spring

static string GetCalendarSeason(DateTime date) => date.Month switch
{
    3 or 4 or 5 => "spring",
    6 or 7 or 8 => "summer",
    9 or 10 or 11 => "autumn",
    12 or 1 or 2 => "winter",
    _ => throw new ArgumentOutOfRangeException(nameof(date), $"Date
with unexpected month: {date.Month}."),
};
```

Comme l'illustre l'exemple précédent, vous pouvez utiliser à plusieurs reprises les combinateurs de modèles dans un modèle.

Priorité et ordre de vérification

La liste suivante présente les combinateurs de modèles par ordre de précédence, de la plus élevée à la plus basse :

- `not`
- `and`
- `or`

Pour spécifier explicitement la priorité, utilisez des parenthèses, comme l'illustre l'exemple suivant :

C#

```
static bool IsLetter(char c) => c is (>= 'a' and <= 'z') or (>= 'A' and <=
'Z');
```

> **Notes**
>
> L'ordre dans lequel les modèles sont vérifiés n'est pas défini. Au moment de l'exécution, les modèles imbriqués à droite des modèles `or` et `and` peuvent être vérifiés en premier.

Pour plus d'informations, consultez la section Combinateurs de modèles de la note de proposition de fonctionnalités.

Modèle de propriété

Vous utilisez un *modèle de propriété* pour faire correspondre les propriétés ou les champs d'une expression aux modèles imbriqués, comme l'illustre l'exemple suivant :

C#

```
static bool IsConferenceDay(DateTime date) => date is { Year: 2020, Month:
5, Day: 19 or 20 or 21 };
```

Un modèle de propriété correspond à une expression lorsqu'un résultat d'expression est non-null et chaque modèle imbriqué correspond à la propriété ou au champ correspondant du résultat de l'expression.

Vous pouvez également ajouter une vérification de type d'exécution et une déclaration de variable à un modèle de propriété, comme l'illustre l'exemple suivant :

C#

```
Console.WriteLine(TakeFive("Hello, world!"));  // output: Hello
Console.WriteLine(TakeFive("Hi!"));  // output: Hi!
Console.WriteLine(TakeFive(new[] { '1', '2', '3', '4', '5', '6', '7' }));
// output: 12345
Console.WriteLine(TakeFive(new[] { 'a', 'b', 'c' }));  // output: abc

static string TakeFive(object input) => input switch
{
    string { Length: >= 5 } s => s.Substring(0, 5),
    string s => s,

    ICollection<char> { Count: >= 5 } symbols => new
string(symbols.Take(5).ToArray()),
    ICollection<char> symbols => new string(symbols.ToArray()),

    null => throw new ArgumentNullException(nameof(input)),
    _ => throw new ArgumentException("Not supported input type."),
};
```

Un modèle de propriété est un modèle récursif. Autrement dit, vous pouvez utiliser n'importe quel modèle comme modèle imbriqué. Utilisez un modèle de propriété pour faire correspondre des parties de données à des modèles imbriqués, comme l'illustre l'exemple suivant :

C#

```
public record Point(int X, int Y);
public record Segment(Point Start, Point End);

static bool IsAnyEndOnXAxis(Segment segment) =>
    segment is { Start: { Y: 0 } } or { End: { Y: 0 } };
```

L'exemple précédent utilise deux fonctionnalités disponibles en C# 9.0 et versions ultérieures : le `or` combinateur de modèles et les types d'enregistrements.

À compter de C# 10, vous pouvez référencer des propriétés ou des champs imbriqués dans un modèle de propriété. Cette fonctionnalité est appelée *modèle de propriété étendu*. Par exemple, vous pouvez refactoriser la méthode de l'exemple précédent dans le code équivalent suivant :

C#

```
static bool IsAnyEndOnXAxis(Segment segment) =>
    segment is { Start.Y: 0 } or { End.Y: 0 };
```

Pour plus d'informations, consultez la section Modèle de propriété de la note de proposition de fonctionnalités et la note de proposition de fonctionnalités Modèles de propriétés étendus.

> **Conseil**
>
> Vous pouvez utiliser la règle de style **Simplifier le modèle de propriété (IDE0170)** pour améliorer la lisibilité du code en suggérant des emplacements pour utiliser des modèles de propriétés étendus.

Modèle positionnel

Vous utilisez un *modèle positionnel* pour déconstruire un résultat d'expression et faire correspondre les valeurs obtenues avec les modèles imbriqués correspondants, comme l'illustre l'exemple suivant :

C#

```
public readonly struct Point
{
    public int X { get; }
    public int Y { get; }
```

```
    public Point(int x, int y) => (X, Y) = (x, y);

    public void Deconstruct(out int x, out int y) => (x, y) = (X, Y);
}

static string Classify(Point point) => point switch
{
    (0, 0) => "Origin",
    (1, 0) => "positive X basis end",
    (0, 1) => "positive Y basis end",
    _ => "Just a point",
};
```

Dans l'exemple précédent, le type d'une expression contient la méthode Deconstruct, qui est utilisée pour déconstruire un résultat d'expression. Vous pouvez également faire correspondre des expressions de types de tuple avec des modèles positionnels. De cette façon, vous pouvez faire correspondre plusieurs entrées avec différents modèles, comme l'illustre l'exemple suivant :

C#

```
static decimal GetGroupTicketPriceDiscount(int groupSize, DateTime
visitDate)
    => (groupSize, visitDate.DayOfWeek) switch
    {
        (<= 0, _) => throw new ArgumentException("Group size must be
positive."),
        (_, DayOfWeek.Saturday or DayOfWeek.Sunday) => 0.0m,
        (>= 5 and < 10, DayOfWeek.Monday) => 20.0m,
        (>= 10, DayOfWeek.Monday) => 30.0m,
        (>= 5 and < 10, _) => 12.0m,
        (>= 10, _) => 15.0m,
        _ => 0.0m,
    };
```

L'exemple précédent utilise des modèles relationnels et logiques, qui sont disponibles dans C# 9.0 et versions ultérieures.

Vous pouvez utiliser les noms des éléments de tuple et les paramètres `Deconstruct` dans un modèle positionnel, comme l'illustre l'exemple suivant :

C#

```
var numbers = new List<int> { 1, 2, 3 };
if (SumAndCount(numbers) is (Sum: var sum, Count: > 0))
{
    Console.WriteLine($"Sum of [{string.Join(" ", numbers)}] is {sum}");  //
output: Sum of [1 2 3] is 6
```

```
}

static (double Sum, int Count) SumAndCount(IEnumerable<int> numbers)
{
    int sum = 0;
    int count = 0;
    foreach (int number in numbers)
    {
        sum += number;
        count++;
    }
    return (sum, count);
}
```

Vous pouvez également étendre un modèle positionnel de l'une des manières suivantes :

- Ajoutez une vérification de type d'exécution et une déclaration de variable, comme l'illustre l'exemple suivant :

  C#

  ```
  public record Point2D(int X, int Y);
  public record Point3D(int X, int Y, int Z);

  static string PrintIfAllCoordinatesArePositive(object point) => point
  switch
  {
      Point2D (> 0, > 0) p => p.ToString(),
      Point3D (> 0, > 0, > 0) p => p.ToString(),
      _ => string.Empty,
  };
  ```

  L'exemple précédent utilise des enregistrements positionnels qui fournissent implicitement la méthode `Deconstruct`.

- Utilisez un modèle de propriété dans un modèle positionnel, comme l'illustre l'exemple suivant :

  C#

  ```
  public record WeightedPoint(int X, int Y)
  {
      public double Weight { get; set; }
  }

  static bool IsInDomain(WeightedPoint point) => point is (>= 0, >= 0) {
  Weight: >= 0.0 };
  ```

- Combinez deux utilisations précédentes, comme l'illustre l'exemple suivant :

```
C#
```

```
if (input is WeightedPoint (> 0, > 0) { Weight: > 0.0 } p)
{
    // ..
}
```

Un modèle positionnel est un modèle récursif. Autrement dit, vous pouvez utiliser n'importe quel modèle comme modèle imbriqué.

Pour plus d'informations, consultez la section Modèle positionnel de la note de proposition de fonctionnalités.

`var` motif

Vous utilisez un *modèle* `var` pour faire correspondre n'importe quelle expression, y compris `null`, et affecter son résultat à une nouvelle variable locale, comme l'illustre l'exemple suivant :

```
C#
```

```
static bool IsAcceptable(int id, int absLimit) =>
    SimulateDataFetch(id) is var results
    && results.Min() >= -absLimit
    && results.Max() <= absLimit;

static int[] SimulateDataFetch(int id)
{
    var rand = new Random();
    return Enumerable
                .Range(start: 0, count: 5)
                .Select(s => rand.Next(minValue: -10, maxValue: 11))
                .ToArray();
}
```

Un modèle `var` est utile lorsque vous avez besoin qu'une variable temporaire dans une expression booléenne contienne le résultat des calculs intermédiaires. Vous pouvez également utiliser un modèle `var` lorsque vous devez effectuer davantage de vérifications dans les gardes de cas `when` d'une expression ou d'une instruction `switch`, comme l'illustre l'exemple suivant :

```
C#
```

```
public record Point(int X, int Y);

static Point Transform(Point point) => point switch
{
    var (x, y) when x < y => new Point(-x, y),
    var (x, y) when x > y => new Point(x, -y),
    var (x, y) => new Point(x, y),
};

static void TestTransform()
{
    Console.WriteLine(Transform(new Point(1, 2)));  // output: Point { X =
-1, Y = 2 }
    Console.WriteLine(Transform(new Point(5, 2)));  // output: Point { X =
5, Y = -2 }
}
```

Dans l'exemple précédent, le modèle `var (x, y)` équivaut à un modèle positionnel `(var x, var y)`.

Dans un modèle `var`, le type d'une variable déclarée est le type au moment de la compilation de l'expression qui est mis en correspondance avec le modèle.

Pour plus d'informations, consultez la section Modèle var de la note de proposition de fonctionnalités.

Ignorer le modèle

Vous utilisez un *modèle d'abandon* `_` pour faire correspondre n'importe quelle expression, y compris `null`, comme l'illustre l'exemple suivant :

C#

```
Console.WriteLine(GetDiscountInPercent(DayOfWeek.Friday));  // output: 5.0
Console.WriteLine(GetDiscountInPercent(null));  // output: 0.0
Console.WriteLine(GetDiscountInPercent((DayOfWeek)10));  // output: 0.0

static decimal GetDiscountInPercent(DayOfWeek? dayOfWeek) => dayOfWeek
switch
{
    DayOfWeek.Monday => 0.5m,
    DayOfWeek.Tuesday => 12.5m,
    DayOfWeek.Wednesday => 7.5m,
    DayOfWeek.Thursday => 12.5m,
    DayOfWeek.Friday => 5.0m,
    DayOfWeek.Saturday => 2.5m,
    DayOfWeek.Sunday => 2.0m,
```

```
    _ => 0.0m,
};
```

Dans l'exemple précédent, un modèle d'abandon est utilisé pour gérer `null` et toute valeur entière qui n'a pas le membre correspondant de l'énumération DayOfWeek. Cela garantit qu'une expression `switch` dans l'exemple gère toutes les valeurs d'entrée possibles. Si vous n'utilisez pas de modèle d'abandon dans une expression `switch` et qu'aucun des modèles de l'expression ne correspond à une entrée, le runtime lève une exception. Le compilateur génère un avertissement si une expression `switch` ne gère pas toutes les valeurs d'entrée possibles.

Un modèle d'abandon ne peut pas être un modèle dans une expression `is` ou une instruction `switch`. Dans ces cas, pour faire correspondre n'importe quelle expression, utilisez un modèle var avec un caractère d'abandon : `var _`.

Pour plus d'informations, consultez la section Modèle d'abandon de la note de proposition de fonctionnalités.

Modèle entre parenthèses

À compter de C# 9.0, vous pouvez placer des parenthèses autour de n'importe quel modèle. En règle générale, vous devez mettre en évidence ou modifier la priorité dans des modèles logiques, comme l'illustre l'exemple suivant :

C#

```
if (input is not (float or double))
{
    return;
}
```

Modèles de liste

À compter de C# 11, vous pouvez faire correspondre un tableau ou une liste à une *séquence* de modèles, comme l'illustre l'exemple suivant :

C#

```
int[] numbers = { 1, 2, 3 };

Console.WriteLine(numbers is [1, 2, 3]);  // True
Console.WriteLine(numbers is [1, 2, 4]);  // False
```

```
Console.WriteLine(numbers is [1, 2, 3, 4]);  // False
Console.WriteLine(numbers is [0 or 1, <= 2, >= 3]);  // True
```

Comme l'illustre l'exemple précédent, un modèle de liste est mis en correspondance lorsque chaque modèle imbriqué est mis en correspondance par l'élément correspondant d'une séquence d'entrée. Vous pouvez utiliser n'importe quel modèle dans un modèle de liste. Pour faire correspondre n'importe quel élément, utilisez le modèle d'abandon ou, si vous souhaitez également capturer l'élément, le modèle var, comme l'illustre l'exemple suivant :

C#

```
List<int> numbers = new() { 1, 2, 3 };

if (numbers is [var first, _, _])
{
    Console.WriteLine($"The first element of a three-item list is
{first}.");
}
// Output:
// The first element of a three-item list is 1.
```

Les exemples précédents correspondent à une séquence d'entrée entière par rapport à un modèle de liste. Pour faire correspondre des éléments uniquement au début ou/et à la fin d'une séquence d'entrée, utilisez le *modèle de section* `..`, comme l'illustre l'exemple suivant :

C#

```
Console.WriteLine(new[] { 1, 2, 3, 4, 5 } is [> 0, > 0, ..]);  // True
Console.WriteLine(new[] { 1, 1 } is [_, _, ..]);  // True
Console.WriteLine(new[] { 0, 1, 2, 3, 4 } is [> 0, > 0, ..]);  // False
Console.WriteLine(new[] { 1 } is [1, 2, ..]);  // False

Console.WriteLine(new[] { 1, 2, 3, 4 } is [.., > 0, > 0]);  // True
Console.WriteLine(new[] { 2, 4 } is [.., > 0, 2, 4]);  // False
Console.WriteLine(new[] { 2, 4 } is [.., 2, 4]);  // True

Console.WriteLine(new[] { 1, 2, 3, 4 } is [>= 0, .., 2 or 4]);  // True
Console.WriteLine(new[] { 1, 0, 0, 1 } is [1, 0, .., 0, 1]);  // True
Console.WriteLine(new[] { 1, 0, 1 } is [1, 0, .., 0, 1]);  // False
```

Un modèle de section correspond à zéro ou plusieurs éléments. Vous pouvez utiliser au plus un modèle de section dans un modèle de liste. Le modèle de section ne peut apparaître que dans un modèle de liste.

Vous pouvez également imbriquer un sous-modèle dans un modèle de section, comme l'illustre l'exemple suivant :

C#

```
void MatchMessage(string message)
{
    var result = message is ['a' or 'A', .. var s, 'a' or 'A']
        ? $"Message {message} matches; inner part is {s}."
        : $"Message {message} doesn't match.";
    Console.WriteLine(result);
}

MatchMessage("aBBA");  // output: Message aBBA matches; inner part is BB.
MatchMessage("apron");  // output: Message apron doesn't match.

void Validate(int[] numbers)
{
    var result = numbers is [< 0, .. { Length: 2 or 4 }, > 0] ? "valid" :
"not valid";
    Console.WriteLine(result);
}

Validate(new[] { -1, 0, 1 });  // output: not valid
Validate(new[] { -1, 0, 0, 1 });  // output: valid
```

Pour plus d'informations, consultez la note de proposition de fonctionnalités Modèles de liste.

spécification du langage C#

Pour plus d'informations, consultez la section Modèles et correspondances de modèles de la spécification du langage C#.

Pour plus d'informations sur les fonctionnalités ajoutées en C# 8 et ultérieur, consultez la note de propositions de fonctionnalités :

- C# 8 - Critères spéciaux récursifs
- C# 9 - Mises à jour des critères spéciaux
- C# 10 - Modèles de propriétés étendus
- C# 11 - Modèles de liste
- C# 11 - Correspondance de modèle Span<char> sur un littéral de chaîne

Voir aussi

- Informations de référence sur C#

- Opérateurs et expressions C#
- Vue d'ensemble des critères spéciaux
- Tutoriel : Utiliser les critères spéciaux pour générer des algorithmes pilotés par type et pilotés par les données

Opérateurs d'addition - `+` et `+=`

Article • 07/04/2023

Les opérateurs `+` et `+=` sont pris en charge par les types numériques intégraux et à virgule flottante intégrés, le type de chaîne et les types délégués .

Pour plus d'informations sur l'opérateur arithmétique `+`, consultez les sections Opérateurs plus et moins unaires et Opérateur d'addition + de l'article Opérateurs arithmétiques.

Concaténation de chaînes

Quand les deux opérandes ou l'un d'entre eux sont de type chaîne, l'opérateur `+` concatène les représentations sous forme de chaîne de ses opérandes (la représentation en chaîne de `null` est une chaîne vide) :

C#

```
Console.WriteLine("Forgot" + "white space");
Console.WriteLine("Probably the oldest constant: " + Math.PI);
Console.WriteLine(null + "Nothing to add.");
// Output:
// Forgotwhite space
// Probably the oldest constant: 3.14159265358979
// Nothing to add.
```

L'interpolation de chaîne fournit un moyen plus pratique de mettre en format les chaînes :

C#

```
Console.WriteLine($"Probably the oldest constant: {Math.PI:F2}");
// Output:
// Probably the oldest constant: 3.14
```

À compter de C# 10, vous pouvez utiliser l'interpolation de chaîne pour initialiser une chaîne constante lorsque toutes les expressions utilisées pour les espaces réservés sont également des chaînes constantes.

À compter de C# 11, l'opérateur `+` effectue la concaténation de chaînes pour les chaînes littérales UTF-8. Cet opérateur concatène deux objets `ReadOnlySpan<byte>`.

Combinaison de délégués

Pour les opérandes du même type délégué, l'opérateur `+` retourne une nouvelle instance de délégué qui, lorsqu'elle est appelée, appelle l'opérande de partie gauche, puis l'opérande de partie droite. Si un des opérandes est `null`, l'opérateur `+` retourne la valeur de l'autre opérande (qui peut également être `null`). L'exemple suivant montre comment les délégués peuvent être combinés avec l'opérateur `+` :

C#

```
Action a = () => Console.Write("a");
Action b = () => Console.Write("b");
Action ab = a + b;
ab();  // output: ab
```

Pour effectuer la suppression de délégué, utilisez l'opérateur -.

Pour plus d'informations sur les types de délégués, consultez Délégués.

Opérateur d'assignation d'addition +=

Expression utilisant l'opérateur `+=`, par exemple

C#

```
x += y
```

équivaut à :

C#

```
x = x + y
```

sauf que `x` n'est évalué qu'une seule fois.

L'exemple suivant illustre l'utilisation de l'opérateur `+=` :

C#

```
int i = 5;
i += 9;
Console.WriteLine(i);
// Output: 14
```

```
string story = "Start. ";
story += "End.";
Console.WriteLine(story);
// Output: Start. End.

Action printer = () => Console.Write("a");
printer();  // output: a

Console.WriteLine();
printer += () => Console.Write("b");
printer();  // output: ab
```

Vous utilisez également l'opérateur `+=` pour spécifier une méthode de gestionnaire d'événements lorsque vous vous abonnez à un événement. Pour plus d'informations, consultez Guide pratique pour s'abonner et annuler l'abonnement à des événements.

Capacité de surcharge de l'opérateur

Un type défini par l'utilisateur peut surcharger l'opérateur `+`. Quand un opérateur binaire `+` est surchargé, l'opérateur `+=` est aussi implicitement surchargé. Un type défini par l'utilisateur ne peut pas surcharger explicitement l'opérateur `+=`.

spécification du langage C#

Pour plus d'informations, consultez les sections Opérateur unaire plus et Opérateur d'addition de la spécification du langage C#.

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- Guide pratique pour concaténer plusieurs chaînes
- Événements
- Opérateurs arithmétiques
- - et -=, opérateurs

Opérateurs - et -= - soustraction (moins)

Article • 07/04/2023

Les opérateurs `-` et `-=` sont pris en charge par les types numériques intégraux et à virgule flottante intégrés et les types délégués.

Pour plus d'informations sur l'opérateur arithmétique `-`, consultez les sections Opérateurs plus et moins unaires et Opérateur de soustraction - de l'article Opérateurs arithmétiques.

Suppression de délégué

Pour les opérandes du même type délégué, l'opérateur `-` retourne une instance de délégué qui est calculée comme suit :

- Si les deux opérandes ont des valeurs non Null et que la liste d'appel de l'opérande de partie droite est une sous-liste contiguë correcte de la liste d'appel de l'opérande de partie droite, le résultat de l'opération est une nouvelle liste d'appel obtenue en supprimant les entrées de l'opérande de partie droite à partir de la liste d'appel de l'opérande de gauche. Si la liste des opérandes de partie droite correspond à plusieurs sous-listes contiguës dans la liste des opérandes de partie gauche, seule la sous-liste correspondante la plus à droite est supprimée. Si la suppression aboutit à une liste vide, le résultat est `null`.

  C#

  ```
  Action a = () => Console.Write("a");
  Action b = () => Console.Write("b");

  var abbaab = a + b + b + a + a + b;
  abbaab();  // output: abbaab
  Console.WriteLine();

  var ab = a + b;
  var abba = abbaab - ab;
  abba();  // output: abba
  Console.WriteLine();

  var nihil = abbaab - abbaab;
  Console.WriteLine(nihil is null);  // output: True
  ```

- Si la liste d'appel de l'opérande de partie droite n'est pas une sous-liste contiguë correcte de la liste d'appel de l'opérande de partie gauche, le résultat de l'opération est l'opérande de partie gauche. Par exemple, la suppression d'un délégué qui ne fait pas partie du délégué multicast ne fait rien et génère un délégué multicast inchangé.

  C#

  ```
  Action a = () => Console.Write("a");
  Action b = () => Console.Write("b");

  var abbaab = a + b + b + a + a + b;
  var aba = a + b + a;

  var first = abbaab - aba;
  first();  // output: abbaab
  Console.WriteLine();
  Console.WriteLine(object.ReferenceEquals(abbaab, first));  // output:
  True

  Action a2 = () => Console.Write("a");
  var changed = aba - a;
  changed();  // output: ab
  Console.WriteLine();
  var unchanged = aba - a2;
  unchanged();  // output: aba
  Console.WriteLine();
  Console.WriteLine(object.ReferenceEquals(aba, unchanged));  // output:
  True
  ```

  L'exemple précédent montre également que durant la suppression de délégué, les instances de délégués sont comparées. Par exemple, les délégués qui sont produits à partir de l'évaluation d'expressions lambda identiques ne sont pas égaux. Pour plus d'informations sur l'égalité des délégués, consultez la section Opérateurs d'égalité de délégués de la spécification du langage C#.

- Si l'opérande de partie gauche est `null`, le résultat de l'opération est `null`. Si l'opérande de partie droite est `null`, le résultat de l'opération est l'opérande de partie gauche.

  C#

  ```
  Action a = () => Console.Write("a");

  var nothing = null - a;
  Console.WriteLine(nothing is null);  // output: True

  var first = a - null;
  a();  // output: a
  ```

```
Console.WriteLine();
Console.WriteLine(object.ReferenceEquals(first, a));  // output: True
```

Pour combiner des délégués, utilisez l'opérateur +.

Pour plus d'informations sur les types de délégués, consultez Délégués.

Opérateur d'assignation de soustraction -=

Expression utilisant l'opérateur `-=`, par exemple

C#

```
x -= y
```

équivaut à :

C#

```
x = x - y
```

sauf que `x` n'est évalué qu'une seule fois.

L'exemple suivant illustre l'utilisation de l'opérateur `-=` :

C#

```
int i = 5;
i -= 9;
Console.WriteLine(i);
// Output: -4

Action a = () => Console.Write("a");
Action b = () => Console.Write("b");
var printer = a + b + a;
printer();  // output: aba

Console.WriteLine();
printer -= a;
printer();  // output: ab
```

Vous utilisez également l'opérateur `-=` pour spécifier une méthode de gestionnaire d'événements à supprimer quand vous vous désabonnez d'un événement. Pour plus d'informations, consultez Guide pratique pour s'abonner et annuler l'abonnement à des événements.

Capacité de surcharge de l'opérateur

Un type défini par l'utilisateur peut surcharger l'opérateur `-`. Quand un opérateur binaire `-` est surchargé, l'opérateur `-=` est aussi implicitement surchargé. Un type défini par l'utilisateur ne peut pas surcharger explicitement l'opérateur `-=`.

spécification du langage C#

Pour plus d'informations, consultez les sections Opérateur moins unaire et Opérateur de soustraction de la spécification du langage C#.

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- Événements
- Opérateurs arithmétiques
- + et +=, opérateurs

Opérateur ?: - L'opérateur conditionnel ternaire

Article • 26/07/2023

L'opérateur conditionnel `?:`, également appelé opérateur conditionnel ternaire, évalue une expression booléenne et renvoie le résultat de l'une des deux expressions, selon que l'expression booléenne est évaluée `true` ou `false`, comme l'illustre l'exemple suivant :

C#

```
string GetWeatherDisplay(double tempInCelsius) => tempInCelsius < 20.0 ?
"Cold." : "Perfect!";

Console.WriteLine(GetWeatherDisplay(15));  // output: Cold.
Console.WriteLine(GetWeatherDisplay(27));  // output: Perfect!
```

Comme le montre l'exemple précédent, la syntaxe de l'opérateur conditionnel est la suivante :

C#

```
condition ? consequent : alternative
```

L'expression `condition` doit donner `true` ou `false`. Si `condition` prend la valeur `true`, l'expression `consequent` est évaluée et son résultat devient le résultat de l'opération. Si `condition` prend la valeur `false`, l'expression `alternative` est évaluée et son résultat devient le résultat de l'opération. Soit `consequent`, soit `alternative` est évaluée.

À partir de C# 9.0, les expressions conditionnelles sont typées par la cible. Autrement dit, si le type cible d'une expression conditionnelle est connu, les types de `consequent` `alternative` doivent être implicitement convertibles en type cible, comme l'illustre l'exemple suivant :

C#

```
var rand = new Random();
var condition = rand.NextDouble() > 0.5;

int? x = condition ? 12 : null;
```

```
IEnumerable<int> xs = x is null ? new List<int>() { 0, 1 } : new int[] { 2,
3 };
```

Si le type cible d'une expression conditionnelle est inconnu (par exemple, lorsque vous utilisez le mot clé var) ou si le type de `consequent` ou `alternative` doit être identique, ou si une conversion implicite d'un type à l'autre est nécessaire :

C#

```
var rand = new Random();
var condition = rand.NextDouble() > 0.5;

var x = condition ? 12 : (int?)null;
```

L'opérateur conditionnel est associatif à droite ; autrement dit, une expression de la forme :

C#

```
a ? b : c ? d : e
```

est évaluée comme étant

C#

```
a ? b : (c ? d : e)
```

Conseil

Vous pouvez utiliser l'appareil mnémonique suivant pour vous souvenir du mode d'évaluation de l'opérateur conditionnel :

text

```
is this condition true ? yes : no
```

Expression ref conditionnelle

Une expression ref conditionnelle retourne de manière conditionnelle une référence de variable, comme le montre l'exemple suivant :

C#

```
var smallArray = new int[] { 1, 2, 3, 4, 5 };
var largeArray = new int[] { 10, 20, 30, 40, 50 };

int index = 7;
ref int refValue = ref ((index < 5) ? ref smallArray[index] : ref
largeArray[index - 5]);
refValue = 0;

index = 2;
((index < 5) ? ref smallArray[index] : ref largeArray[index - 5]) = 100;

Console.WriteLine(string.Join(" ", smallArray));
Console.WriteLine(string.Join(" ", largeArray));
// Output:
// 1 2 100 4 5
// 10 20 0 40 50
```

Vous pouvez ref affecter le résultat d'une expression ref conditionnelle, utilisez-le comme retour de référence ou passez-le en tant que paramètre de méthode ref, outou in. Vous pouvez également affecter au résultat d'une expression ref conditionnelle, comme le montre l'exemple précédent.

La syntaxe d'une expression ref conditionnelle est la suivante :

C#

```
condition ? ref consequent : ref alternative
```

Comme l'opérateur conditionnel, une expression ref n'évalue qu'une des deux expressions : `consequent` ou `alternative`.

Dans une expression ref conditionnelle, `consequent` et `alternative` doivent être du même type. Les expressions ref conditionnelles ne sont pas typées par la cible.

Opérateur conditionnel et instruction `if`

L'utilisation de l'opérateur conditionnel au lieu d'une instruction if permet de rester concis dans l'écriture du code lorsque l'objectif est de calculer une valeur sous condition. L'exemple suivant montre deux façons de classer un entier comme négatif ou non :

C#

```
int input = new Random().Next(-5, 5);
```

```
string classify;
if (input >= 0)
{
    classify = "nonnegative";
}
else
{
    classify = "negative";
}

classify = (input >= 0) ? "nonnegative" : "negative";
```

Capacité de surcharge de l'opérateur

Un type défini par l'utilisateur ne peut pas surcharger l'opérateur conditionnel.

spécification du langage C#

Pour plus d'informations, voir la section Opérateur conditionnel de la spécification du langage C#.

Les spécifications des nouvelles fonctionnalités sont les suivantes :

- Expression conditionnelle typée cible (C# 9.0)

Voir aussi

- Simplifier l'expression conditionnelle (règle de style IDE0075)
- Informations de référence sur C#
- Opérateurs et expressions C#
- if, instruction
- ?. Opérateurs and ?[]
- ?? et ??=, opérateurs
- ref, mot clé

! Opérateur ! (null-indulgent) (référence C#)

Article • 07/04/2023

L'opérateur unaire suffixé `!` est l'opérateur null-forgiving, ou null-suppression. Lorsque l'annotation pouvant accepter la valeur Null est activée, l'opérateur null-forgiving vous permet de supprimer tous les avertissements pouvant accepter la valeur Null pour l'expression précédente. L'opérateur unaire suffixé `!` est l'opérateur de négation logique. L'opérateur null-forgiving n'a aucun effet au moment de l'exécution. Il affecte uniquement l'analyse du flux statique du compilateur en modifiant l'état Null de l'expression. Au moment de l'exécution, l'expression `x!` correspond au résultat de l'expression `x` sous-jacente .

Pour plus d'informations sur la fonctionnalité des types référence pouvant accepter la valeur Null, consultez Types référence pouvant accepter la valeur Null.

Exemples

Le test de la logique de validation des arguments est l'un des cas d'usage de l'opérateur null-forgiving. Par exemple, considérons la classe suivante :

C#

```
#nullable enable
public class Person
{
    public Person(string name) => Name = name ?? throw new
ArgumentNullException(nameof(name));

    public string Name { get; }
}
```

Utilisez l'infrastructure de test MSTest afin de créer le test suivant pour la logique de validation dans le constructeur :

C#

```
[TestMethod, ExpectedException(typeof(ArgumentNullException))]
public void NullNameShouldThrowTest()
{
    var person = new Person(null!);
}
```

Sans l'opérateur null-forgiving, le compilateur génère l'avertissement suivant pour le code précédent : `Warning CS8625: Cannot convert null literal to non-nullable reference type`. Avec l'opérateur null-forgiving, vous informez le compilateur que le passage de `null` est attendu et ne doit pas faire l'objet d'un avertissement.

Vous pouvez également utiliser l'opérateur null-forgiving lorsque vous savez pertinemment qu'une expression ne peut pas être `null` mais que le compilateur ne parvient pas à le reconnaître. Dans l'exemple suivant, si la méthode `IsValid` renvoie `true`, son argument n'est pas `null` et vous pouvez le déréférencer en toute sécurité :

C#

```
public static void Main()
{
    Person? p = Find("John");
    if (IsValid(p))
    {
        Console.WriteLine($"Found {p!.Name}");
    }
}

public static bool IsValid(Person? person)
    => person is not null && person.Name is not null;
```

Sans l'opérateur null-forgiving, le compilateur génère l'avertissement suivant pour le code `p.Name` : `Warning CS8602: Dereference of a possibly null reference`.

Si vous pouvez modifier la méthode `IsValid`, utilisez l'attribut NotNullWhen pour informer le compilateur qu'un argument de la méthode `IsValid` ne peut pas être `null` lorsque la méthode renvoie `true` :

C#

```
public static void Main()
{
    Person? p = Find("John");
    if (IsValid(p))
    {
        Console.WriteLine($"Found {p.Name}");
    }
}

public static bool IsValid([NotNullWhen(true)] Person? person)
    => person is not null && person.Name is not null;
```

Dans l'exemple précédent, il n'est pas nécessaire d'utiliser l'opérateur null-forgiving, car le compilateur dispose d'informations suffisantes pour savoir que `p` ne peut pas être `null` dans l'instruction `if`. Pour plus d'informations sur les attributs qui vous permettent de fournir des informations supplémentaires sur l'état Null d'une variable, consultez Mettre à niveau les API avec des attributs pour définir les attentes en matière de valeurs Null.

spécification du langage C#

Pour plus d'informations, consultez la section L'opérateur null-forgiving du projet de spécification sur les types référence pouvant accepter la valeur Null.

Voir aussi

- Supprimer l'opérateur de suppression inutile (règle de style IDE0080)
- Informations de référence sur C#
- Opérateurs et expressions C#
- Tutoriel : Concevoir avec des types référence pouvant accepter la valeur Null

?? Opérateurs ??=, opérateurs de coalescence nulle

Article • 07/04/2023

L'opérateur de fusion null `??` retourne la valeur de l'opérande de gauche si elle n'est pas `null` ; sinon, il évalue l'opérande de droite et retourne son résultat. L'opérateur `??` n'évalue pas son opérande droit si l'opérande gauche a la valeur non Null. L'opérateur de coalescence nulle `??=` assigne la valeur de son opérande droit à son opérande gauche uniquement si l'opérande gauche a la valeur `null`. L'opérateur `??=` n'évalue pas son opérande droit si l'opérande gauche a la valeur non Null.

C#

```
List<int> numbers = null;
int? a = null;

Console.WriteLine((numbers is null)); // expected: true
// if numbers is null, initialize it. Then, add 5 to numbers
(numbers ??= new List<int>()).Add(5);
Console.WriteLine(string.Join(" ", numbers));  // output: 5
Console.WriteLine((numbers is null)); // expected: false


Console.WriteLine((a is null)); // expected: true
Console.WriteLine((a ?? 3)); // expected: 3 since a is still null
// if a is null then assign 0 to a and add a to the list
numbers.Add(a ??= 0);
Console.WriteLine((a is null)); // expected: false
Console.WriteLine(string.Join(" ", numbers));  // output: 5 0
Console.WriteLine(a);  // output: 0
```

L'opérande gauche de l'opérateur `??=` doit être une variable, une propriétéou un élément d'indexeur.

Le type de l'opérande gauche des opérateurs `??` et `??=` ne peut pas être un type valeur non-nullable. En particulier, vous pouvez utiliser les opérateurs de coalescence nulle avec des paramètres de type non contraints :

C#

```
private static void Display<T>(T a, T backup)
{
    Console.WriteLine(a ?? backup);
}
```

Les opérateurs de coalescence nulle sont associatifs de droite. Autrement dit, des expressions du formulaire

```
C#

a ?? b ?? c
d ??= e ??= f
```

sont évalués en tant que

```
C#

a ?? (b ?? c)
d ??= (e ??= f)
```

Exemples

Les opérateurs `??` et `??=` peuvent être utiles dans les scénarios suivants :

- Dans les expressions avec les opérateurs conditionnels null?. et ?[], vous pouvez utiliser l'opérateur `??` pour fournir une autre expression à évaluer si le résultat de l'expression avec des opérations conditionnelles null est `null` :

  ```
  C#

  double SumNumbers(List<double[]> setsOfNumbers, int indexOfSetToSum)
  {
      return setsOfNumbers?[indexOfSetToSum]?.Sum() ?? double.NaN;
  }

  var sum = SumNumbers(null, 0);
  Console.WriteLine(sum);  // output: NaN
  ```

- Lorsque vous utilisez des types valeur nullables et que vous devez fournir une valeur d'un type valeur sous-jacent, utilisez l'opérateur `??` pour spécifier la valeur à fournir au cas où une valeur de type nullable est `null` :

  ```
  C#

  int? a = null;
  int b = a ?? -1;
  Console.WriteLine(b);  // output: -1
  ```

Utilisez la méthode Nullable<T>.GetValueOrDefault() si la valeur à utiliser quand une valeur de type Nullable est `null` doit être la valeur par défaut du type valeur sous-jacent.

- Vous pouvez utiliser une expression throw comme opérande droit de l'opérateur `??` pour rendre le code de vérification des arguments plus concis :

```
C#

public string Name
{
    get => name;
    set => name = value ?? throw new
ArgumentNullException(nameof(value), "Name cannot be null");
}
```

L'exemple précédent montre également comment utiliser des membres expression-bodied pour définir une propriété.

- Vous pouvez utiliser l'opérateur `??=` pour remplacer le code du formulaire

```
C#

if (variable is null)
{
    variable = expression;
}
```

par le code suivant :

```
C#

variable ??= expression;
```

Capacité de surcharge de l'opérateur

Les opérateurs `??` et `??=` ne peuvent pas être surchargés.

spécification du langage C#

Pour plus d'informations sur l'opérateur `??`, consultez la section Opérateur de coalescence nulle de la spécification du langage C#.

Pour plus d'informations sur l'opérateur `??=`, consultez la note de proposition de fonctionnalité.

Voir aussi

- Utiliser l'expression coalesce (règles de style IDE0029 et IDE0030)
- Informations de référence sur C#
- Opérateurs et expressions C#
- ?. et opérateurs ?[]
- ?:, opérateur

L'opérateur d'expression lambda (`=>`) définit une expression lambda

Article • 07/04/2023

Le jeton `=>` est pris en charge sous deux formes : comme opérateur lambda, et comme séparateur d'un nom de membre et de l'implémentation de membre dans une définition de corps d'expression.

Opérateur lamdba

Dans expressions lambda, l'opérateur lambda `=>` sépare les paramètres d'entrée du côté gauche du corps lambda à droite.

L'exemple suivant utilise la fonctionnalité LINQ avec la syntaxe de méthode pour illustrer l'utilisation d'expressions lambda :

C#

```
string[] words = { "bot", "apple", "apricot" };
int minimalLength = words
  .Where(w => w.StartsWith("a"))
  .Min(w => w.Length);
Console.WriteLine(minimalLength);   // output: 5

int[] numbers = { 4, 7, 10 };
int product = numbers.Aggregate(1, (interim, next) => interim * next);
Console.WriteLine(product);   // output: 280
```

Les paramètres d'entrée d'une expression lambda sont fortement typés au moment de la compilation. Lorsque le compilateur peut déduire les types de paramètres d'entrée, comme dans l'exemple précédent, vous pouvez omettre les déclarations de type. Si vous devez spécifier le type de paramètres d'entrée, vous devez le faire pour chaque paramètre, comme l'illustre l'exemple suivant :

C#

```
int[] numbers = { 4, 7, 10 };
int product = numbers.Aggregate(1, (int interim, int next) => interim *
next);
Console.WriteLine(product);   // output: 280
```

L'exemple suivant montre comment définir une expression lambda sans paramètres d'entrée :

```
C#
```

```
Func<string> greet = () => "Hello, World!";
Console.WriteLine(greet());
```

Pour plus d'informations, consultez Expressions lambda.

Définition de corps d'expression

La syntaxe générale d'une définition de corps d'expression est la suivante :

```
C#
```

```
member => expression;
```

où `expression` est une expression valide. Le type de retour de `expression` doit être implicitement convertible en type de retour du membre. Si le membre :

- A un type de retour `void` ou
- Est un :
  - Constructeur
  - Finaliseur
  - Accesseur de propriété ou d'indexeur `set`

`expression` doit être une *expression d'instruction*. Étant donné que le résultat de l'expression est ignoré, le type de retour de cette expression peut être de n'importe quel type.

L'exemple suivant montre une définition de corps d'expression pour une méthode `Person.ToString` :

```
C#
```

```
public override string ToString() => $"{fname} {lname}".Trim();
```

Il s'agit d'une version raccourcie de la définition de méthode suivante :

```
C#
```

```
public override string ToString()
{
   return $"{fname} {lname}".Trim();
}
```

Vous pouvez créer des définitions de corps d'expression pour les méthodes, les opérateurs, les propriétés en lecture seule, les constructeurs, les finaliseurs et les accesseurs de propriété et d'indexeur. Pour plus d'informations, consultez Membres expression-bodied.

Capacité de surcharge de l'opérateur

L'opérateur `=>` ne peut pas être surchargé.

spécification du langage C#

Pour plus d'informations, consultez la section Expressions de fonction anonyme de la spécification du langage C#.

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#

:: opérateur - opérateur d'alias d'espace de noms

Article • 10/05/2023

Utilisez le qualificateur d'alias d'espace de noms `::` pour accéder au membre d'un espace de noms avec alias. Vous ne pouvez utiliser le qualificateur `::` qu'entre deux identificateurs. L'identificateur de gauche peut être un alias d'espace de noms, un alias externe ou l'alias `global`. Par exemple :

- Un alias d'espace de noms créé avec la directive using alias :

  C#

  ```
  using forwinforms = System.Drawing;
  using forwpf = System.Windows;

  public class Converters
  {
      public static forwpf::Point Convert(forwinforms::Point point) =>
  new forwpf::Point(point.X, point.Y);
  }
  ```

- Un alias externe.

- L'alias `global`, qui est l'alias d'espace de noms global. L'espace de noms global est l'espace de noms qui contient des espaces de noms et des types qui ne sont pas déclarés dans un espace de noms nommé. Quand il est utilisé avec le qualificateur `::`, l'alias `global` référence toujours l'espace de noms global, même si l'alias d'espace de noms `global` défini par l'utilisateur est présent.

  L'exemple suivant utilise l'alias `global` pour accéder à l'espace de noms .NET System, qui est un membre de l'espace de noms global. Sans l'alias `global`, l'accès se ferait à l'espace de noms `System` défini par l'utilisateur, qui est un membre de l'espace de noms `MyCompany.MyProduct` :

  C#

  ```
  namespace MyCompany.MyProduct.System
  {
      class Program
      {
          static void Main() => global::System.Console.WriteLine("Using
  global alias");
      }
  ```

```
    class Console
    {
        string Suggestion => "Consider renaming this class";
    }
}
```

> ⓘ **Notes**
>
> Le mot clé `global` est l'alias d'espace de noms global seulement quand il s'agit de l'identificateur de gauche du qualificateur `::`.

Vous pouvez également utiliser le jeton . pour accéder à un membre d'un espace de noms avec alias. Toutefois, le jeton `.` est également utilisé pour accéder à un membre de type. Le qualificateur `::` garantit que son identificateur de gauche référence toujours un alias d'espace de noms, même s'il existe un type ou un espace de noms portant le même nom.

spécification du langage C#

Pour plus d'informations, consultez la section Qualificateurs d'alias d'espace de noms de la spécification du langage C#.

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#

opérateur await : attend de façon asynchrone la fin d'une tâche

Article • 27/03/2023

L'opérateur `await` suspend l'évaluation de la méthode async englobante jusqu'à ce que l'opération asynchrone représentée par son opérande se termine. Une fois l'opération asynchrone terminée, l'opérateur `await` retourne le résultat de l'opération, le cas échéant. Quand l'opérateur `await` est appliqué à l'opérande qui représente une opération déjà terminée, il retourne immédiatement le résultat de l'opération sans suspendre la méthode englobante. L'opérateur `await` ne bloque pas le thread qui évalue la méthode async. Quand l'opérateur `await` suspend la méthode englobante, le contrôle retourne à l'appelant de la méthode.

Dans l'exemple suivant, la méthode HttpClient.GetByteArrayAsync retourne l'instance `Task<byte[]>`, qui représente une opération asynchrone qui produit un tableau d'octets au moment où elle se termine. Tant que l'opération n'est pas terminée, l'opérateur `await` suspend la méthode `DownloadDocsMainPageAsync`. Quand la méthode `DownloadDocsMainPageAsync` est suspendue, le contrôle est retourné à la méthode `Main`, qui est l'appelant de `DownloadDocsMainPageAsync`. La méthode `Main` s'exécute jusqu'à ce qu'elle ait besoin du résultat de l'opération asynchrone effectuée par la méthode `DownloadDocsMainPageAsync`. Une fois que tous les octets ont été obtenus par GetByteArrayAsync, le reste de la méthode `DownloadDocsMainPageAsync` est évalué. Après cela, le reste de la méthode `Main` est évalué.

C#

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

public class AwaitOperator
{
    public static async Task Main()
    {
        Task<int> downloading = DownloadDocsMainPageAsync();
        Console.WriteLine($"{nameof(Main)}: Launched downloading.");

        int bytesLoaded = await downloading;
        Console.WriteLine($"{nameof(Main)}: Downloaded {bytesLoaded}
bytes.");
    }

    private static async Task<int> DownloadDocsMainPageAsync()
```

```
    {
        Console.WriteLine($"{nameof(DownloadDocsMainPageAsync)}: About to
start downloading.");

        var client = new HttpClient();
        byte[] content = await
client.GetByteArrayAsync("https://learn.microsoft.com/en-us/");

        Console.WriteLine($"{nameof(DownloadDocsMainPageAsync)}: Finished
downloading.");
        return content.Length;
    }
}
// Output similar to:
// DownloadDocsMainPageAsync: About to start downloading.
// Main: Launched downloading.
// DownloadDocsMainPageAsync: Finished downloading.
// Main: Downloaded 27700 bytes.
```

L'exemple précédent utilise la méthode méthode Main asynchrone. Pour plus d'informations, consultez la section Opérateur await dans la méthode Main.

> ⓘ Notes
>
> Pour obtenir une introduction à la programmation asynchrone, consultez **Programmation asynchrone avec async et await**. La programmation asynchrone avec `async` et `await` suit le **modèle asynchrone basé sur les tâches**.

Vous pouvez utiliser l'opérateur `await` uniquement dans une méthode, une expression lambda ou une méthode anonyme modifiée par le mot clé async. Dans une méthode asynchrone, vous ne pouvez pas utiliser l'opérateur `await` dans le corps d'une fonction synchrone, à l'intérieur du bloc d'une instruction lock et dans un contexte unsafe.

L'opérande de l'opérateur `await` est généralement un des types .NET suivants : Task, Task<TResult>, ValueTask ou ValueTask<TResult>. Cependant, n'importe quelle expression awaitable peut être l'opérande de l'opérateur `await`. Pour plus d'informations, consultez la section Expressions awaitable de la spécification du langage C#.

Le type d'expression `await t` est `TResult` si le type d'expression `t` est Task<TResult> ou ValueTask<TResult>. Si le type de `t` est Task ou ValueTask, le type de `await t` est `void`. Dans les deux cas, si `t` lève une exception, `await t` lève à nouveau l'exception.

Flux asynchrones et éléments jetables

Vous utilisez l'instruction `await foreach` pour consommer un flux asynchrone de données. Pour plus d'informations, consultez la section foreach de l'article Instructions d'itération.

Vous utilisez l'instruction `await using` pour utiliser un objet jetable de façon asynchrone. Autrement dit un objet d'un type qui implémente une interface IAsyncDisposable. Pour plus d'informations, consultez la section Utilisation d'objet jetable asynchrone de l'article Implémenter une méthode DisposeAsync.

Opérateur await dans la méthode Main

La méthode Main, qui est le point d'entrée de l'application, peut retourner `Task` ou `Task<int>`, ce qui lui permet d'être asynchrone. Vous pouvez ainsi utiliser l'opérateur `await` dans son corps. Dans les versions antérieures de C#, pour être certain que la méthode `Main` attend la fin d'une opération asynchrone, vous pouvez récupérer la valeur de la propriété Task<TResult>.Result de l'instance Task<TResult> retournée par la méthode asyn correspondante. Pour les opérations asynchrones qui ne produisent pas de valeur, vous pouvez appeler la méthode Task.Wait. Pour savoir comment sélectionner la version du langage, voir Contrôle de version du langage C#.

spécification du langage C#

Pour plus d'informations, consultez la section Expressions await de la spécification du langage C#.

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- async
- Modèle de programmation asynchrone des tâches
- Programmation asynchrone
- Procédure pas à pas : accès au web avec async et await
- Tutoriel : Générer et consommer des flux asynchrones
- Blog .NET : Fonctionnement réel d'async/await en C#

expressions de valeur par défaut : produire la valeur par défaut

Article • 10/05/2023

Une expression de valeur par défaut produit la valeur par défaut d'un type . Il existe deux types d'expressions de valeur par défaut : l'appel de l'opérateur par défaut et un littéral par défaut.

Vous utilisez également le mot-clé `default` comme étiquette case par défaut dans une instruction switch.

Opérateur default

L'argument de l'opérateur `default` doit avoir le nom d'un type ou d'un paramètre de type, comme le montre l'exemple suivant :

C#

```
Console.WriteLine(default(int));  // output: 0
Console.WriteLine(default(object) is null);  // output: True

void DisplayDefaultOf<T>()
{
    var val = default(T);
    Console.WriteLine($"Default value of {typeof(T)} is {(val == null ?
"null" : val.ToString())}.");
}

DisplayDefaultOf<int?>();
DisplayDefaultOf<System.Numerics.Complex>();
DisplayDefaultOf<System.Collections.Generic.List<int>>();
// Output:
// Default value of System.Nullable`1[System.Int32] is null.
// Default value of System.Numerics.Complex is (0, 0).
// Default value of System.Collections.Generic.List`1[System.Int32] is null.
```

littéral par défaut

À partir de C# 7.1, vous pouvez utiliser le littéral `default` pour produire la valeur par défaut d'un type lorsque le compilateur peut déduire le type d'expression. L'expression littérale `default` génère la même valeur que l'expression `default(T)`, où `T` est le type déduit. Vous pouvez utiliser le littéral `default` dans les cas suivants :

- Dans l'affectation ou l'initialisation d'une variable.
- Dans la déclaration de la valeur par défaut d'un paramètre de méthode facultatif.
- Dans un appel de méthode pour fournir une valeur d'argument.
- Dans une instruction return ou en tant qu'expression d'un membre expression-bodied.

L'exemple suivant illustre l'utilisation du littéral `default` :

C#

```
T[] InitializeArray<T>(int length, T initialValue = default)
{
    if (length < 0)
    {
        throw new ArgumentOutOfRangeException(nameof(length), "Array length
must be nonnegative.");
    }

    var array = new T[length];
    for (var i = 0; i < length; i++)
    {
        array[i] = initialValue;
    }
    return array;
}

void Display<T>(T[] values) => Console.WriteLine($"[ {string.Join(", ",
values)} ]");

Display(InitializeArray<int>(3));  // output: [ 0, 0, 0 ]
Display(InitializeArray<bool>(4, default));  // output: [ False, False,
False, False ]

System.Numerics.Complex fillValue = default;
Display(InitializeArray(3, fillValue));  // output: [ (0, 0), (0, 0), (0, 0)
]
```

Conseil

Utilisez la règle de style .NET **IDE0034** pour spécifier une préférence sur l'utilisation du littéral `default` dans votre codebase.

spécification du langage C#

Pour plus d'informations, voir la section Expressions de valeur par défaut de la spécification du langage C#.

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- Valeurs par défaut des types C#
- Génériques en .NET

Opérateur delegate

Article • 10/05/2023

L'opérateur `delegate` crée une méthode anonyme qui peut être convertie en un type délégué. Une méthode anonyme peut être convertie en types (par exemple System.Action et System.Func<TResult>) et être utilisés comme arguments en plusieurs méthodes.

C#

```
Func<int, int, int> sum = delegate (int a, int b) { return a + b; };
Console.WriteLine(sum(3, 4));  // output: 7
```

Notes

Les expressions lambda fournissent un moyen plus concis et expressif de créer une fonction anonyme. Utilisez l'opérateur => pour construire une expression lambda :

C#

```
Func<int, int, int> sum = (a, b) => a + b;
Console.WriteLine(sum(3, 4));  // output: 7
```

Pour plus d'informations sur les fonctionnalités des expressions lambda, par exemple la capture de variables externes, consultez **Expressions lambda**.

Lorsque vous utilisez l'opérateur `delegate`, vous pouvez omettre la liste de paramètres. Dans ce cas, la méthode anonyme créée peut être convertie en un type délégué avec n'importe quelle liste de paramètres, comme le montre l'exemple suivant :

C#

```
Action greet = delegate { Console.WriteLine("Hello!"); };
greet();

Action<int, double> introduce = delegate { Console.WriteLine("This is
world!"); };
introduce(42, 2.7);

// Output:
// Hello!
// This is world!
```

C'est la seule fonctionnalité des méthodes anonymes qui n'est pas prise en charge par les expressions lambda. Dans tous les autres cas, une expression lambda est le moyen préféré pour écrire du code incorporé.

À compter de C# 9.0, vous pouvez utiliser des abandons pour spécifier deux paramètres d'entrée ou plus d'une méthode anonyme qui ne sont pas utilisés par la méthode :

```
C#

Func<int, int, int> constant = delegate (int _, int _) { return 42; };
Console.WriteLine(constant(3, 4));  // output: 42
```

Pour la compatibilité descendante, si un seul paramètre est nommé `_`, `_` est traité comme le nom de ce paramètre dans une méthode anonyme.

À compter de C# 9.0, vous pouvez également utiliser le modificateur `static` à la déclaration d'une méthode anonyme :

```
C#

Func<int, int, int> sum = static delegate (int a, int b) { return a + b; };
Console.WriteLine(sum(10, 4));  // output: 14
```

Une méthode anonyme statique ne peut pas capturer les variables locales ou l'état de l'instance à partir d'étendues englobantes.

Vous utilisez également le mot clé `delegate` pour déclarer un type délégué.

À compter de C# 11, le compilateur peut mettre en cache l'objet délégué créé à partir d'un groupe de méthodes. Tenez compte de la méthode suivante :

```
C#

static void StaticFunction() { }
```

Lorsque vous affectez le groupe de méthodes à un délégué, le compilateur met en cache le délégué :

```
C#

Action a = StaticFunction;
```

Avant C# 11, vous deviez utiliser une expression lambda pour réutiliser un objet délégué unique :

C#

```
Action a = () => StaticFunction();
```

spécification du langage C#

Pour plus d'informations, consultez la section Expressions de fonction anonyme de la spécification du langage C#.

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- => operateur

Opérateur « is » (référence C#)

Article • 07/04/2023

L'opérateur `is` vérifie si le résultat d'une expression est compatible avec un type donné. Pour plus d'informations sur l'opérateur de test de type `is`, consultez la section Opérateur « is » de l'article Opérateurs de test et de cast de type. Vous pouvez également utiliser l'opérateur `is` pour faire correspondre une expression à un modèle, comme l'illustre l'exemple suivant :

C#

```
static bool IsFirstFridayOfOctober(DateTime date) =>
    date is { Month: 10, Day: <=7, DayOfWeek: DayOfWeek.Friday };
```

Dans l'exemple précédent, l'opérateur `is` correspond à une expression par rapport à un modèle de propriété avec des modèles relationnels et constants imbriqués (disponibles dans C# 9.0 et versions ultérieures).

L'opérateur `is` peut être utile dans les scénarios suivants :

- Pour vérifier le type d'exécution d'une expression, comme l'illustre l'exemple suivant :

  C#

  ```
  int i = 34;
  object iBoxed = i;
  int? jNullable = 42;
  if (iBoxed is int a && jNullable is int b)
  {
      Console.WriteLine(a + b);  // output 76
  }
  ```

  L'exemple précédent montre l'utilisation d'un modèle de déclaration.

- Pour rechercher `null`, comme l'illustre l'exemple suivant :

  C#

  ```
  if (input is null)
  {
      return;
  }
  ```

Lorsque vous faites correspondre une expression par rapport à `null`, le compilateur garantit qu'aucun opérateur surchargé par l'utilisateur `==` ou `!=` n'est appelé.

- À compter de C# 9.0, vous pouvez utiliser un modèle de négation pour effectuer une vérification non null, comme l'illustre l'exemple suivant :

  C#

  ```
  if (result is not null)
  {
      Console.WriteLine(result.ToString());
  }
  ```

- À compter de C# 11, vous pouvez utiliser des modèles de liste pour faire correspondre des éléments d'une liste ou d'un tableau. Le code suivant vérifie les tableaux pour les valeurs entières dans les positions attendues :

  C#

  ```
  int[] empty = { };
  int[] one = { 1 };
  int[] odd = { 1, 3, 5 };
  int[] even = { 2, 4, 6 };
  int[] fib = { 1, 1, 2, 3, 5 };

  Console.WriteLine(odd is [1, _, 2, ..]);   // false
  Console.WriteLine(fib is [1, _, 2, ..]);   // true
  Console.WriteLine(fib is [_, 1, 2, 3, ..]);     // true
  Console.WriteLine(fib is [.., 1, 2, 3, _ ]);     // true
  Console.WriteLine(even is [2, _, 6]);     // true
  Console.WriteLine(even is [2, .., 6]);    // true
  Console.WriteLine(odd is [.., 3, 5]); // true
  Console.WriteLine(even is [.., 3, 5]); // false
  Console.WriteLine(fib is [.., 3, 5]); // true
  ```

> ⓘ **Notes**
>
> Pour obtenir la liste complète des modèles pris en charge par l'opérateur `is` , consultez **Modèles**.

spécification du langage C#

Pour plus d'informations, consultez la section L'opérateur is de la Specification du langage C# et les propositions de langage C# suivantes :

- Critères spéciaux
- Critères spéciaux avec génériques

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- Modèles
- Tutoriel : Utiliser la correspondance de modèle pour générer des algorithmes pilotés par type et pilotés par les données
- Opérateurs de conversion et de test de type

Expression nameof (référence C#)

Article • 16/03/2023

Une expression `nameof` produit le nom d'une variable, d'un type ou d'un membre en tant que constante de chaîne. Une expression `nameof` est évaluée au moment de la compilation et n'a aucun effet au moment de l'exécution. Lorsque l'opérande est un type ou un espace de noms, le nom produit n'est pas complet. L'exemple suivant illustre l'utilisation d'une expression `nameof` :

C#

```
Console.WriteLine(nameof(System.Collections.Generic));  // output: Generic
Console.WriteLine(nameof(List<int>));  // output: List
Console.WriteLine(nameof(List<int>.Count));  // output: Count
Console.WriteLine(nameof(List<int>.Add));  // output: Add

var numbers = new List<int> { 1, 2, 3 };
Console.WriteLine(nameof(numbers));  // output: numbers
Console.WriteLine(nameof(numbers.Count));  // output: Count
Console.WriteLine(nameof(numbers.Add));  // output: Add
```

Vous pouvez utiliser une expression `nameof` pour rendre le code de vérification des arguments plus gérable :

C#

```
public string Name
{
    get => name;
    set => name = value ?? throw new ArgumentNullException(nameof(value), $"
{nameof(Name)} cannot be null");
}
```

À compter de C# 11, vous pouvez utiliser une expression `nameof` avec un paramètre de méthode à l'intérieur d'un attribut sur une méthode ou son paramètre. Le code suivant montre comment procéder pour un attribut sur une méthode, une fonction locale et le paramètre d'une expression lambda :

C#

```
[ParameterString(nameof(msg))]
public static void Method(string msg)
{
    [ParameterString(nameof(T))]
    void LocalFunction<T>(T param) { }
```

```
    var lambdaExpression = ([ParameterString(nameof(aNumber))] int aNumber)
=> aNumber.ToString();
}
```

Une expression `nameof` avec un paramètre est utile lorsque vous utilisez les attributs d'analyse nullables ou l'attribut CallerArgumentExpression.

Lorsque l'opérande est un identificateur verbatim, le caractère `@` n'est pas la partie d'un nom, comme le montre l'exemple suivant :

C#

```
var @new = 5;
Console.WriteLine(nameof(@new));  // output: new
```

spécification du langage C#

Pour plus d'informations, consultez la section Expressions Nameof de la spécification du langage C# et la spécification de fonctionnalité C# 11 - Portéenameof étendue.

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- Convertir en typeof à nameof (règle de style IDE0082)

Nouvel opérateur : l'opérateur `new` crée une nouvelle instance d'un type

Article • 02/06/2023

L'opérateur `new` crée une nouvelle instance d'un type. Vous pouvez également utiliser le mot clé `new` comme modificateur de déclaration de membre ou contrainte de type générique.

Appel du constructeur

Pour créer une nouvelle instance d'un type, vous appelez généralement l'un des constructeurs de ce type, à l'aide de l'opérateur `new` :

C#

```
var dict = new Dictionary<string, int>();
dict["first"] = 10;
dict["second"] = 20;
dict["third"] = 30;

Console.WriteLine(string.Join("; ", dict.Select(entry => $"{entry.Key}:
{entry.Value}")));
// Output:
// first: 10; second: 20; third: 30
```

Vous pouvez utiliser un initialiseur d'objet ou de collection avec l'opérateur `new` pour instancier et initialiser un objet dans une instruction, comme dans l'exemple suivant :

C#

```
var dict = new Dictionary<string, int>
{
    ["first"] = 10,
    ["second"] = 20,
    ["third"] = 30
};

Console.WriteLine(string.Join("; ", dict.Select(entry => $"{entry.Key}:
{entry.Value}")));
// Output:
// first: 10; second: 20; third: 30
```

Type cible `new`

À compter de C# 9.0, les expressions d'appel de constructeur sont typées par cible. Autrement dit, si un type cible d'une expression est connu, vous pouvez omettre un nom de type, comme l'illustre l'exemple suivant :

C#

```
List<int> xs = new();
List<int> ys = new(capacity: 10_000);
List<int> zs = new() { Capacity = 20_000 };

Dictionary<int, List<int>> lookup = new()
{
    [1] = new() { 1, 2, 3 },
    [2] = new() { 5, 8, 3 },
    [5] = new() { 1, 0, 4 }
};
```

Comme l'illustre l'exemple précédent, vous utilisez toujours des parenthèses dans une expression `new` typée par cible.

Si le type cible d'une expression `new` est inconnu (par exemple, lorsque vous utilisez le mot clé var), vous devez spécifier un nom de type.

Création de tableau

Vous utilisez également l'opérateur `new` pour créer une instance de tableau, comme dans l'exemple suivant :

C#

```
var numbers = new int[3];
numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;

Console.WriteLine(string.Join(", ", numbers));
// Output:
// 10, 20, 30
```

Utilisez la syntaxe d'initialisation de tableau pour créer une instance de tableau et la remplir avec des éléments dans une instruction. L'exemple suivant montre différentes façons de procéder :

C#

```
var a = new int[3] { 10, 20, 30 };
var b = new int[] { 10, 20, 30 };
var c = new[] { 10, 20, 30 };
Console.WriteLine(c.GetType());  // output: System.Int32[]
```

Pour plus d'informations sur les tableaux, consultez Tableaux.

Instanciation des types anonymes

Pour créer une instance d'un type anonyme, utilisez l'opérateur `new` et la syntaxe d'initialiseur d'objet :

C#

```
var example = new { Greeting = "Hello", Name = "World" };
Console.WriteLine($"{example.Greeting}, {example.Name}!");
// Output:
// Hello, World!
```

Destruction des instances de type

Vous n'êtes pas obligé de détruire les instances de type précédemment créées. Les instances de types référence et valeur sont détruites automatiquement. Les instances de types valeur sont détruites dès que le contexte dans lequel elles se trouvent est détruit. Les instances de types référence sont détruites par le récupérateur de mémoire à un moment non spécifié, après la suppression de la dernière référence à celles-ci.

Pour les instances de type qui contiennent des ressources non managées, par exemple, un descripteur de fichier, il est recommandé d'utiliser le nettoyage déterministe pour s'assurer que les ressources qu'elles contiennent sont publiées dès que possible. Pour plus d'informations, consultez la référence d'API System.IDisposable et l'article using, instruction.

Capacité de surcharge de l'opérateur

Un type défini par l'utilisateur peut surcharger l'opérateur `new`.

spécification du langage C#

Pour plus d'informations, consultez la section Opérateur new de la spécification du langage C#.

Pour plus d'informations sur une expression `new` typée par cible, consultez la note de proposition de fonctionnalité.

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- Initialiseurs d'objets et de collections

L’opérateur sizeof détermine les besoins en mémoire d’un type donné

Article • 10/05/2023

L’opérateur `sizeof` retourne le nombre d’octets occupés par une variable d’un type donné. L’argument de l’opérateur `sizeof` doit être le nom d’un type non managé ou d’un paramètre de type qui est contraint comme type non managé.

L’opérateur `sizeof` nécessite un contexte unsafe. Toutefois, les expressions présentées dans le tableau suivant sont évaluées au moment de la compilation par rapport aux valeurs de constante correspondantes et ne nécessitent pas de contexte unsafe :

Expression	Valeur constante
`sizeof(sbyte)`	1
`sizeof(byte)`	1
`sizeof(short)`	2
`sizeof(ushort)`	2
`sizeof(int)`	4
`sizeof(uint)`	4
`sizeof(long)`	8
`sizeof(ulong)`	8
`sizeof(char)`	2
`sizeof(float)`	4
`sizeof(double)`	8
`sizeof(decimal)`	16
`sizeof(bool)`	1

Vous n’avez pas non plus besoin d’utiliser un contexte unsafe quand l’opérande de l’opérateur `sizeof` est le nom d’un type enum.

L’exemple suivant illustre l’utilisation de l’opérateur `sizeof` :

```
C#
```

```
public struct Point
{
    public Point(byte tag, double x, double y) => (Tag, X, Y) = (tag, x, y);

    public byte Tag { get; }
    public double X { get; }
    public double Y { get; }
}

public class SizeOfOperator
{
    public static void Main()
    {
        Console.WriteLine(sizeof(byte));  // output: 1
        Console.WriteLine(sizeof(double));  // output: 8

        DisplaySizeOf<Point>();  // output: Size of Point is 24
        DisplaySizeOf<decimal>();  // output: Size of System.Decimal is 16

        unsafe
        {
            Console.WriteLine(sizeof(Point*));  // output: 8
        }
    }

    static unsafe void DisplaySizeOf<T>() where T : unmanaged
    {
        Console.WriteLine($"Size of {typeof(T)} is {sizeof(T)}");
    }
}
```

L'opérateur `sizeof` retourne un nombre d'octets qui seraient alloués par la Common Language Runtime dans la mémoire managée. Pour les types struct, cette valeur comprend le remplissage, comme le montre l'exemple précédent. Le résultat de l'opérateur `sizeof` peut différer de celui de la méthode Marshal.SizeOf, qui retourne la taille d'un type dans la mémoire *non managée*.

spécification du langage C#

Pour plus d'informations, consultez la section Opérateur sizeof de la spécification du langage C#.

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- Opérateurs associés au pointeur

- Types de pointeur
- Types liés à la mémoire et à l'étendue
- Génériques en .NET

Expression stackalloc (référence C#)

Article • 07/04/2023

Une expression `stackalloc` alloue un bloc de mémoire dans la pile. Un bloc de mémoire alloué dans la pile pendant l'exécution de la méthode est automatiquement supprimé lorsque cette méthode retourne un résultat. Vous ne pouvez pas libérer explicitement la mémoire allouée avec `stackalloc`. Un bloc de mémoire alloué dans la pile n'est pas soumis au nettoyage de la mémoire et ne doit pas être épinglé avec une instruction fixed.

Vous pouvez attribuer le résultat d'une expression `stackalloc` à une variable d'un des types suivants :

- System.Span<T> ou System.ReadOnlySpan<T>, comme le montre l'exemple suivant :

  C#

  ```
  int length = 3;
  Span<int> numbers = stackalloc int[length];
  for (var i = 0; i < length; i++)
  {
      numbers[i] = i;
  }
  ```

  Vous n'avez pas l'obligation d'utiliser un contexte unsafe lorsque vous affectez un bloc de mémoire alloué dans la pile à une variable Span<T> ou ReadOnlySpan<T>.

  Lorsque vous travaillez avec ces types, vous pouvez utiliser une expression `stackalloc` les expressions conditionnelles ou d'affectation, comme le montre l'exemple suivant :

  C#

  ```
  int length = 1000;
  Span<byte> buffer = length <= 1024 ? stackalloc byte[length] : new
  byte[length];
  ```

  Vous pouvez utiliser une expression `stackalloc` dans d'autres expressions chaque fois qu'une variable Span<T> ou ReadOnlySpan<T> est autorisée, comme le montre l'exemple suivant :

C#

```
Span<int> numbers = stackalloc[] { 1, 2, 3, 4, 5, 6 };
var ind = numbers.IndexOfAny(stackalloc[] { 2, 4, 6, 8 });
Console.WriteLine(ind);  // output: 1
```

> ⓘ **Notes**
>
> Nous vous recommandons d'utiliser les types **Span<T>** ou **ReadOnlySpan<T>** pour travailler avec la mémoire allouée dans la pile quand cela est possible.

- Un type pointeur, comme illustré dans l'exemple suivant :

  C#

  ```
  unsafe
  {
      int length = 3;
      int* numbers = stackalloc int[length];
      for (var i = 0; i < length; i++)
      {
          numbers[i] = i;
      }
  }
  ```

  Comme le montre l'exemple précédent, vous devez utiliser un contexte `unsafe` lorsque vous travaillez avec des types pointeurs.

  Dans le cas des types pointeur, vous pouvez utiliser une expression `stackalloc` uniquement dans une déclaration de variable locale pour initialiser la variable.

La quantité de mémoire disponible dans la pile est limitée. Si vous allouez trop de mémoire dans la pile, une StackOverflowException est levée. Pour éviter cela, respectez les règles ci-dessous :

- Limitez la quantité de mémoire que vous allouez avec `stackalloc`. Par exemple, si la taille de mémoire tampon prévue est inférieure à une certaine limite, vous allouez la mémoire dans la pile ; sinon, utilisez un tableau de la longueur requise, comme le montre le code suivant :

  C#

  ```
  const int MaxStackLimit = 1024;
  Span<byte> buffer = inputLength <= MaxStackLimit ? stackalloc
  ```

```
byte[MaxStackLimit] : new byte[inputLength];
```

> ⓘ **Notes**
>
> Étant donné que la quantité de mémoire disponible dans la pile dépend de l'environnement dans lequel le code est exécuté, soyez prudent lorsque vous définissez la valeur limite réelle.

- Évitez d'utiliser des boucles internes `stackalloc`. Allouez le bloc de mémoire en dehors d'une boucle et réutilisez-le à l'intérieur de la boucle.

Le contenu de la mémoire nouvellement allouée n'est pas défini. Vous devez l'initialiser avant l'utilisation. Par exemple, vous pouvez utiliser la méthode Span<T>.Clear qui définit tous les éléments sur la valeur par défaut de type `T`.

Vous pouvez utiliser la syntaxe d'initialiseur de tableau pour définir le contenu de la mémoire nouvellement allouée. L'exemple suivant illustre différentes manières de le faire :

C#

```
Span<int> first = stackalloc int[3] { 1, 2, 3 };
Span<int> second = stackalloc int[] { 1, 2, 3 };
ReadOnlySpan<int> third = stackalloc[] { 1, 2, 3 };
```

Dans l'expression `stackalloc T[E]`, `T` doit être un type non managé et `E` doit être évalué à une valeur int non négative.

Sécurité

L'utilisation de `stackalloc` active automatiquement les fonctionnalités de détection des dépassements de mémoire tampon dans le Common Language Runtime (CLR). Si un dépassement de mémoire tampon est détecté, le processus est terminé aussi rapidement que possible pour réduire les risques d'exécution de code malveillant.

spécification du langage C#

Pour plus d'informations, consultez la section Allocation de piles de la spécification de langage C# et la note de proposition de fonctionnalité Autoriserstackalloc dans des contextes imbriqués.

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- Opérateurs associés au pointeur
- Types de pointeur
- Types liés à la mémoire et à l'étendue
- stackalloc : à faire et à ne pas faire

expression switch : expressions de correspondance de modèle à l'aide du mot clé `switch`

Article • 09/05/2023

Vous utilisez l'expression `switch` pour évaluer une seule expression à partir d'une liste d'expressions candidates basées sur une correspondance de modèle avec une expression d'entrée. Pour plus d'informations sur l'instruction `switch` qui prend en charge `switch`la sémantique -like dans un contexte d'instruction, consultez la section switchinstruction de l'article Instructions de sélection .

L'exemple suivant illustre une expression `switch`, qui convertit les valeurs d'un enum représentant des directions visuelles dans une carte en ligne dans les directions cardinales correspondantes :

C#

```
public static class SwitchExample
{
    public enum Direction
    {
        Up,
        Down,
        Right,
        Left
    }

    public enum Orientation
    {
        North,
        South,
        East,
        West
    }

    public static Orientation ToOrientation(Direction direction) =>
direction switch
    {
        Direction.Up    => Orientation.North,
        Direction.Right => Orientation.East,
        Direction.Down  => Orientation.South,
        Direction.Left  => Orientation.West,
        _ => throw new ArgumentOutOfRangeException(nameof(direction), $"Not
expected direction value: {direction}"),
    };
```

```
    public static void Main()
    {
        var direction = Direction.Right;
        Console.WriteLine($"Map view direction is {direction}");
        Console.WriteLine($"Cardinal orientation is
{ToOrientation(direction)}");
        // Output:
        // Map view direction is Right
        // Cardinal orientation is East
    }
}
```

L'exemple précédent montre les éléments de base d'une expression `switch` :

- Expression suivie du mot clé `switch`. Dans l'exemple précédent, il s'agit du paramètre de méthode `direction`.
- Les *`switch` bras d'expression*, séparés par des virgules. Chaque `switch` bras d'expression contient un *modèle*, une *protection de cas* facultative, le jeton `=>` et une *expression*.

Dans l'exemple précédent, une expression `switch` utilise les modèles suivants :

- Un modèle constant : pour gérer les valeurs définies de l'énumération `Direction`.
- Un modèle d'abandon : pour gérer toute valeur entière qui n'a pas le membre correspondant de l'énumération `Direction` (par exemple, `(Direction)10`). Cela rend l'expression `switch` exhaustive.

> **ⓘ Important**
>
> Pour plus d'informations sur les modèles pris en charge par l'expression `switch` et d'autres exemples, consultez **Modèles**.

Le résultat d'une expression `switch` est la valeur de l'expression du premier `switch` bras d'expression dont le modèle correspond à l'expression d'entrée et dont la protection de cas, le cas échéant, prend la valeur `true`. Les bras d'expression `switch` sont évalués dans l'ordre textuel.

Le compilateur génère une erreur lorsqu'un bras d'expression inférieur `switch` ne peut pas être choisi, car un bras d'expression supérieur `switch` correspond à toutes ses valeurs.

Protections de cas

Un modèle peut ne pas être suffisamment expressif pour spécifier la condition d'évaluation de l'expression d'un bras. Dans ce cas, vous pouvez utiliser une *protection de cas*. Une *protection contre la casse* est une autre condition qui doit être satisfaite avec un modèle correspondant. Une protection de cas doit être une expression booléenne. Vous spécifiez une protection de cas après le mot clé `when` qui suit un modèle, comme le montre l'exemple suivant :

C#

```
public readonly struct Point
{
    public Point(int x, int y) => (X, Y) = (x, y);

    public int X { get; }
    public int Y { get; }
}

static Point Transform(Point point) => point switch
{
    { X: 0, Y: 0 }                    => new Point(0, 0),
    { X: var x, Y: var y } when x < y => new Point(x + y, y),
    { X: var x, Y: var y } when x > y => new Point(x - y, y),
    { X: var x, Y: var y }            => new Point(2 * x, 2 * y),
};
```

L'exemple précédent utilise des modèles de propriété avec des modèles var imbriqués.

Expressions de commutateur non exhaustives

Si aucun des modèles d'une expression `switch` ne correspond à une valeur d'entrée, le runtime lève une exception. Dans .NET Core 3.0 et versions ultérieures, l'exception est un System.Runtime.CompilerServices.SwitchExpressionException. Dans .NET Framework, l'exception est un InvalidOperationException. Dans la plupart des cas, le compilateur génère un avertissement si une expression `switch` ne gère pas toutes les valeurs d'entrée possibles. Les modèles de liste ne génèrent pas d'avertissement lorsque toutes les entrées possibles ne sont pas gérées.

> **Conseil**
>
> Pour garantir qu'une `switch` expression gère toutes les valeurs d'entrée possibles, fournissez un bras d'expression `switch` avec un **modèle d'abandon**.

spécification du langage C#

Pour plus d'informations, consultez la switch section expression de la note de proposition de fonctionnalité.

Voir aussi

- Utiliser l'expression de commutateur (règle de style IDE0066)
- Ajouter des cas manquants à l'expression de commutateur (règle de style IDE0072)
- Informations de référence sur C#
- Opérateurs et expressions C#
- Modèles
- Didacticiel : Utiliser les critères spéciaux pour générer des algorithmes pilotés par type et pilotés par les données
- instruction switch

opérateurs true et false : traitez vos objets comme une valeur booléenne

Article • 07/04/2023

L'opérateur `true` retourne la valeur bool`true` pour indiquer que son opérande est vrai. L'opérateur `false` retourne la valeur `bool` `true` pour indiquer que son opérande est faux. Les opérateurs `true` et `false` ne sont pas forcément complémentaires. Autrement dit, les opérateurs `true` et `false` peuvent retourner la valeur `bool` `false` pour le même opérande. Si un type définit un des deux opérateurs, il doit aussi définir un autre opérateur.

> **Conseil**
>
> Utilisez le type `bool?` si vous voulez suivre une logique à trois valeurs, par exemple pour travailler avec des bases de données qui acceptent un type booléen à trois valeurs. C# fournit les opérateurs `&` et `|` qui prennent en charge la logique à trois valeurs avec les opérandes `bool?`. Pour plus d'informations, voir la section **Opérateurs logiques booléens Nullable** de l'article **Opérateurs logiques booléens**.

Expressions booléennes

Un type avec l'opérateur `true` défini peut être le type de résultat d'une expression conditionnelle de contrôle dans les instructions if, do, while et for et dans l'opérateur conditionnel ?:. Pour plus d'informations, voir la section Expressions booléennes de la spécification du langage C#.

Opérateurs logiques conditionnels définis par l'utilisateur

Si un type avec les opérateurs `true` et `false` définis surchargel'opérateur OR logique`|` ou l'opérateur AND logique`&` d'une façon, l'opérateur OR logique conditionnel`||` ou l'opérateur AND logique conditionnel`&&`, respectivement, peut être évalué pour les opérandes de ce type. Pour plus d'informations, consultez la section Opérateurs logiques conditionnels définis par l'utilisateur de la spécification du langage C#.

Exemple

L'exemple suivant présente le type qui définit les opérateurs `true` et `false`. Le type surcharge également l'opérateur AND logique `&` de sorte que l'opérateur `&&` peut aussi être évalué pour les opérandes de ce type.

C#

```
public struct LaunchStatus
{
    public static readonly LaunchStatus Green = new LaunchStatus(0);
    public static readonly LaunchStatus Yellow = new LaunchStatus(1);
    public static readonly LaunchStatus Red = new LaunchStatus(2);

    private int status;

    private LaunchStatus(int status)
    {
        this.status = status;
    }

    public static bool operator true(LaunchStatus x) => x == Green || x ==
Yellow;
    public static bool operator false(LaunchStatus x) => x == Red;

    public static LaunchStatus operator &(LaunchStatus x, LaunchStatus y)
    {
        if (x == Red || y == Red || (x == Yellow && y == Yellow))
        {
            return Red;
        }

        if (x == Yellow || y == Yellow)
        {
            return Yellow;
        }

        return Green;
    }

    public static bool operator ==(LaunchStatus x, LaunchStatus y) =>
x.status == y.status;
    public static bool operator !=(LaunchStatus x, LaunchStatus y) => !(x ==
y);

    public override bool Equals(object obj) => obj is LaunchStatus other &&
this == other;
    public override int GetHashCode() => status;
}

public class LaunchStatusTest
{
```

```
    public static void Main()
    {
        LaunchStatus okToLaunch = GetFuelLaunchStatus() &&
GetNavigationLaunchStatus();
        Console.WriteLine(okToLaunch ? "Ready to go!" : "Wait!");
    }

    static LaunchStatus GetFuelLaunchStatus()
    {
        Console.WriteLine("Getting fuel launch status...");
        return LaunchStatus.Red;
    }

    static LaunchStatus GetNavigationLaunchStatus()
    {
        Console.WriteLine("Getting navigation launch status...");
        return LaunchStatus.Yellow;
    }
}
```

Remarquez le comportement de court-circuit de l'opérateur `&&`. Lorsque la méthode `GetFuelLaunchStatus` retourne `LaunchStatus.Red`, l'opérande de la partie droite de l'opérateur `&&` n'est pas évalué. En effet, `LaunchStatus.Red` est false. Le résultat du AND logique ne dépend donc pas de la valeur de l'opérande de partie droite. Voici la sortie de l'exemple :

Console

```
Getting fuel launch status...
Wait!
```

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#

Expression « with » : une mutation non destructive qui crée un objet avec des propriétés modifiées

Article • 27/03/2023

Disponible en C# 9.0 et versions ultérieures, une expression `with` produit une copie de son opérande avec les propriétés et champs spécifiés modifiés. Vous utilisez la syntaxe de l'initialiseur d'objet pour spécifier les membres à modifier ainsi que leurs nouvelles valeurs :

C#

```
using System;

public class WithExpressionBasicExample
{
    public record NamedPoint(string Name, int X, int Y);

    public static void Main()
    {
        var p1 = new NamedPoint("A", 0, 0);
        Console.WriteLine($"{nameof(p1)}: {p1}");  // output: p1: NamedPoint
{ Name = A, X = 0, Y = 0 }

        var p2 = p1 with { Name = "B", X = 5 };
        Console.WriteLine($"{nameof(p2)}: {p2}");  // output: p2: NamedPoint
{ Name = B, X = 5, Y = 0 }

        var p3 = p1 with
            {
                Name = "C",
                Y = 4
            };
        Console.WriteLine($"{nameof(p3)}: {p3}");  // output: p3: NamedPoint
{ Name = C, X = 0, Y = 4 }

        Console.WriteLine($"{nameof(p1)}: {p1}");  // output: p1: NamedPoint
{ Name = A, X = 0, Y = 0 }

        var apples = new { Item = "Apples", Price = 1.19m };
        Console.WriteLine($"Original: {apples}");  // output: Original: {
Item = Apples, Price = 1.19 }
        var saleApples = apples with { Price = 0.79m };
        Console.WriteLine($"Sale: {saleApples}");  // output: Sale: { Item =
Apples, Price = 0.79 }
    }
}
```

Dans C# 9.0, l'opérande de partie gauche d'une expression `with` doit être d'un type d'enregistrement. À compter de C# 10, un opérande de partie gauche d'une expression `with` peut également être d'un type de structure ou d'un type anonyme.

Le résultat d'une expression `with` a le même type d'exécution que l'opérande de l'expression, comme l'illustre l'exemple suivant :

C#

```
using System;

public class InheritanceExample
{
    public record Point(int X, int Y);
    public record NamedPoint(string Name, int X, int Y) : Point(X, Y);

    public static void Main()
    {
        Point p1 = new NamedPoint("A", 0, 0);
        Point p2 = p1 with { X = 5, Y = 3 };
        Console.WriteLine(p2 is NamedPoint);  // output: True
        Console.WriteLine(p2);  // output: NamedPoint { X = 5, Y = 3, Name =
A }
    }
}
```

Dans le cas d'un membre de type référence, seule la référence à une instance de membre est copiée lorsqu'un opérande est copié. La copie et l'opérande d'origine ont accès à la même instance de type référence. L'exemple suivant illustre ce comportement :

C#

```
using System;
using System.Collections.Generic;

public class ExampleWithReferenceType
{
    public record TaggedNumber(int Number, List<string> Tags)
    {
        public string PrintTags() => string.Join(", ", Tags);
    }

    public static void Main()
    {
        var original = new TaggedNumber(1, new List<string> { "A", "B" });

        var copy = original with { Number = 2 };
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B
```

```
        original.Tags.Add("C");
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B, C
    }
}
```

Sémantique de copie personnalisée

Tout type de classe d'enregistrement a le *constructeur de copie*. Un *constructeur de copie* est un constructeur avec un seul paramètre du type d'enregistrement conteneur. Il copie l'état de son argument dans une nouvelle instance d'enregistrement. Lors de l'évaluation d'une expression `with`, le constructeur de copie est appelé pour instancier une nouvelle instance d'enregistrement basée sur un enregistrement d'origine. Après cela, la nouvelle instance est mise à jour en fonction des modifications spécifiées. Par défaut, le constructeur de copie est implicite, c'est-à-dire généré par le compilateur. Si vous voulez personnaliser la sémantique de copie d'enregistrement, déclarez explicitement un constructeur de copie avec le comportement souhaité. L'exemple suivant met à jour l'exemple précédent avec un constructeur de copie explicite. Le nouveau comportement de copie consiste à copier des éléments de liste au lieu d'une référence de liste lorsqu'un enregistrement est copié :

C#

```
using System;
using System.Collections.Generic;

public class UserDefinedCopyConstructorExample
{
    public record TaggedNumber(int Number, List<string> Tags)
    {
        protected TaggedNumber(TaggedNumber original)
        {
            Number = original.Number;
            Tags = new List<string>(original.Tags);
        }

        public string PrintTags() => string.Join(", ", Tags);
    }

    public static void Main()
    {
        var original = new TaggedNumber(1, new List<string> { "A", "B" });

        var copy = original with { Number = 2 };
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B
```

```
        original.Tags.Add("C");
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B
    }
}
```

Vous ne pouvez pas personnaliser la sémantique de copie pour les types de structure.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la note de proposition de fonctionnalité d'enregistrements :

- with, expression
- Copier et cloner des membres

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- Enregistrements
- Types de structure

Surcharge d'opérateur : opérateurs unaires, arithmétiques, d'égalité et de comparaison prédéfinis

Article • 07/04/2023

Un type défini par l'utilisateur peut surcharger un opérateur C# prédéfini. Autrement dit, un type peut fournir l'implémentation personnalisée d'une opération si l'un des opérandes (voire les deux) est de ce type. La section Opérateurs surchargeables indique les opérateurs C# qui peuvent être surchargés.

Utilisez le mot clé `operator` pour déclarer un opérateur. Une déclaration d'opérateur doit respecter les règles suivantes :

- Elle contient un modificateur `public` et un modificateur `static`.
- Un opérateur unaire possède un paramètre d'entrée. Un opérateur binaire possède deux paramètres d'entrée. Dans chaque cas, au moins un paramètre doit avoir le type `T` ou `T?`, où `T` est le type qui contient la déclaration d'opérateur.

L'exemple suivant définit une structure simplifiée pour représenter un nombre rationnel. La structure surcharge certains des opérateurs arithmétiques :

C#

```
public readonly struct Fraction
{
    private readonly int num;
    private readonly int den;

    public Fraction(int numerator, int denominator)
    {
        if (denominator == 0)
        {
            throw new ArgumentException("Denominator cannot be zero.",
nameof(denominator));
        }
        num = numerator;
        den = denominator;
    }

    public static Fraction operator +(Fraction a) => a;
    public static Fraction operator -(Fraction a) => new Fraction(-a.num,
a.den);

    public static Fraction operator +(Fraction a, Fraction b)
        => new Fraction(a.num * b.den + b.num * a.den, a.den * b.den);
```

```
    public static Fraction operator -(Fraction a, Fraction b)
        => a + (-b);

    public static Fraction operator *(Fraction a, Fraction b)
        => new Fraction(a.num * b.num, a.den * b.den);

    public static Fraction operator /(Fraction a, Fraction b)
    {
        if (b.num == 0)
        {
            throw new DivideByZeroException();
        }
        return new Fraction(a.num * b.den, a.den * b.num);
    }

    public override string ToString() => $"{num} / {den}";
}

public static class OperatorOverloading
{
    public static void Main()
    {
        var a = new Fraction(5, 4);
        var b = new Fraction(1, 2);
        Console.WriteLine(-a);   // output: -5 / 4
        Console.WriteLine(a + b);  // output: 14 / 8
        Console.WriteLine(a - b);  // output: 6 / 8
        Console.WriteLine(a * b);  // output: 5 / 8
        Console.WriteLine(a / b);  // output: 10 / 4
    }
}
```

Vous pouvez étendre l'exemple précédent en définissant une conversion implicite de `int` à `Fraction`. Les opérateurs surchargés devraient alors prendre en charge les arguments de ces deux types. Autrement dit, il deviendrait possible d'ajouter un entier à une fraction et d'obtenir une fraction en conséquence.

Vous utilisez également le mot clé `operator` pour définir une conversion de type personnalisée. Pour plus d'informations, consultez Opérateurs de conversion définie par l'utilisateur.

Opérateurs surchargeables

Le tableau suivant répertorie les opérateurs qui peuvent être surchargés :

Opérateurs	Notes

Opérateurs	Notes
+x, -x, !x, ~x, ++, --, true, false	Les opérateurs `true` et `false` doivent être surchargés ensemble.
x + y, x - y, x * y, x / y, x % y, x & y, x | y, x ^ y, x << y, x >> y, x >>> y	
x == y, x != y, x < y, x > y, x <= y, x >= y	Doivent être surchargés par paires comme suit : `==` et `!=`, `<` et `>`, `<=` et `>=`.

Opérateurs non surchargeables

Le tableau suivant répertorie les opérateurs qui ne peuvent pas être surchargés :

Opérateurs	Autres solutions
x && y, x || y	Surchargez à la fois les opérateurs true et false, et les opérateurs & ou |. Pour plus d'informations, consultez Opérateurs logiques conditionnels définis par l'utilisateur.
a[i], a?[i]	Définissez un indexeur.
(T)x	Définissez des conversions de type personnalisées qui peuvent être exécutées par une expression cast. Pour plus d'informations, consultez Opérateurs de conversion définie par l'utilisateur.
+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=	Surchargez l'opérateur binaire correspondant. Par exemple, lorsque vous surchargez l'opérateur binaire `+`, `+=` est implicitement surchargé.
^x, x = y, x.y, x?.y, c ? t : f, x ?? y, ??= y, x..y, x->y, =>, f(x), as, await, checked, unchecked, default, delegate, is, nameof, new, sizeof, stackalloc, switch, typeof, with	Aucun.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Surcharge d'opérateur
- Opérateurs

Voir aussi

- Informations de référence sur C#
- Opérateurs et expressions C#
- Opérateurs de conversion définie par l'utilisateur
- Instructions de conception - Surcharges d'opérateur
- Instructions de conception - Opérateurs d'égalité
- Why are overloaded operators always static in C#?

Instructions d'itération - `for`, `foreach`, `do` et `while`

Article • 07/04/2023

Les instructions d'itération exécutent à plusieurs reprises une instruction ou un bloc d'instructions. L'instruction for exécute son corps tandis qu'une expression booléenne spécifiée donne la valeur `true`. L'instruction foreach énumère les éléments d'une collection et exécute son corps pour chaque élément de la collection. L'instruction do exécute de manière conditionnelle son corps une ou plusieurs fois. L'instruction while exécute de manière conditionnelle son corps zéro ou plusieurs fois.

À tout moment dans le corps d'une instruction d'itération, vous pouvez sortir de la boucle à l'aide de l'instruction break. Vous pouvez passer à l'itération suivante dans la boucle à l'aide de l'instruction continue.

Instruction `for`

L'instruction `for` exécute une instruction ou un bloc d'instructions tant qu'une expression booléenne donne la valeur `true`. L'exemple suivant montre l'instruction `for` qui exécute son corps alors qu'un compteur entier est inférieur à trois :

```
C#
```

```
for (int i = 0; i < 3; i++)
{
    Console.Write(i);
}
// Output:
// 012
```

L'exemple précédent montre les éléments de l'instruction `for` :

- La section *initialiseur* exécutée une seule fois, avant d'entrer dans la boucle. En règle générale, vous déclarez et initialisez une variable de boucle locale dans cette section. La variable déclarée n'est pas accessible à partir de l'extérieur de l'instruction `for`.

  La section *initialiseur* de l'exemple précédent déclare et initialise une variable de compteur entier :

  ```
  C#
  ```

```
int i = 0
```

- La section *condition* qui détermine si l'itération suivante de la boucle doit être exécutée. Si elle est évaluée à `true` ou si elle n'est pas présente, l'itération suivante est exécutée ; sinon, la boucle est supprimée. La section *condition* doit être une expression booléenne.

  La section *condition* de l'exemple précédent vérifie si une valeur de compteur est inférieure à trois :

  C#

  ```
  i < 3
  ```

- La section *itérateur* définit ce qui se produit après chaque exécution du corps de la boucle.

  La section *itérateur* de l'exemple précédent incrémente le compteur :

  C#

  ```
  i++
  ```

- Le corps de la boucle, qui doit être une instruction ou un bloc d'instructions.

La section itérateur peut contenir zéro, une ou plusieurs des expressions d'instruction suivantes, séparées par des virgules :

- Expression d'incrémentation préfixée ou suffixée, telle que `++i` ou `i++`
- Expression de décrémentation préfixée ou suffixée, telle que `--i` ou `i--`
- assignation
- Appel d'une méthode
- await, expression
- création d'un objet à l'aide de l'opérateur new

Si vous ne déclarez pas de variable de boucle dans la section initialiseur, vous pouvez également utiliser zéro ou plusieurs expressions de la liste précédente dans la section initialiseur. L'exemple suivant illustre plusieurs utilisations moins courantes des sections initialiseur et itérateur : l'attribution d'une valeur à une variable externe dans la section initialiseur, l'appel d'une méthode dans les sections initialiseur et itérateur, et la modification des valeurs de deux variables dans la section itérateur :

C#

```
int i;
int j = 3;
for (i = 0, Console.WriteLine($"Start: i={i}, j={j}"); i < j; i++, j--,
Console.WriteLine($"Step: i={i}, j={j}"))
{
    //...
}
// Output:
// Start: i=0, j=3
// Step: i=1, j=2
// Step: i=2, j=1
```

Toutes les sections de l'instruction `for` sont facultatives. Par exemple, le code suivant définit la boucle `for` infinie :

C#

```
for ( ; ; )
{
    //...
}
```

Instruction `foreach`

L'instruction `foreach` exécute une instruction ou un bloc d'instructions pour chaque élément d'une instance du type qui implémente l'interface System.Collections.IEnumerable ou System.Collections.Generic.IEnumerable<T>, comme l'indique l'exemple suivant :

C#

```
var fibNumbers = new List<int> { 0, 1, 1, 2, 3, 5, 8, 13 };
foreach (int element in fibNumbers)
{
    Console.Write($"{element} ");
}
// Output:
// 0 1 1 2 3 5 8 13
```

L'instruction `foreach` n'est pas limitée à ces types. Vous pouvez l'utiliser avec une instance de n'importe quel type qui répond aux conditions suivantes :

- Un type a la méthode `GetEnumerator` sans paramètre publique. À compter de C# 9.0, la méthode `GetEnumerator` peut être la méthode d'extension d'un type.

- Le type de retour de la méthode `GetEnumerator` contient la propriété publique `Current` et la méthode sans paramètre publique `MoveNext`, dont le type de retour est `bool`.

L'exemple suivant utilise l'instruction `foreach` avec une instance de type System.Span<T> qui n'implémente aucune interface :

C#

```
Span<int> numbers = new int[] { 3, 14, 15, 92, 6 };
foreach (int number in numbers)
{
    Console.Write($"{number} ");
}
// Output:
// 3 14 15 92 6
```

Si la propriété `Current` de l'énumérateur retourne une valeur de retour de référence (`ref T` où `T` est le type de l'élément de collection), vous pouvez déclarer une variable d'itération avec le modificateur `ref` ou `ref readonly`, comme l'indique l'exemple suivant :

C#

```
Span<int> storage = stackalloc int[10];
int num = 0;
foreach (ref int item in storage)
{
    item = num++;
}
foreach (ref readonly var item in storage)
{
    Console.Write($"{item} ");
}
// Output:
// 0 1 2 3 4 5 6 7 8 9
```

Si la collection source de l'instruction `foreach` est vide, le corps de l'instruction `foreach` n'est pas exécuté et est ignoré. Si l'instruction `foreach` est appliquée à `null`, une NullReferenceException est levée.

await foreach

Vous pouvez utiliser l'instruction `await foreach` pour consommer un flux asynchrone de données, autrement dit, le type de collection qui implémente l'interface

IAsyncEnumerable<T>. Chaque itération de la boucle peut être suspendue pendant que l'élément suivant est récupéré de façon asynchrone. L'exemple suivant montre comment utiliser l'instruction `await foreach` :

```
C#
```

```
await foreach (var item in GenerateSequenceAsync())
{
    Console.WriteLine(item);
}
```

Vous pouvez également utiliser l'instruction `await foreach` avec une instance de n'importe quel type qui répond aux conditions suivantes :

- Un type a la méthode `GetAsyncEnumerator` sans paramètre public. Cette méthode peut être la méthode d'extension d'un type.
- Le type de retour de la méthode `GetAsyncEnumerator` a la propriété `Current` publique et la méthode `MoveNextAsync` publique sans paramètre dont le type de retour est Task<bool>, ValueTask<bool> ou tout autre type awaitable dont la méthode du `GetResult` de l'awaiter retourne une valeur `bool`.

Par défaut, les éléments de flux sont traités dans le contexte capturé. Si vous souhaitez désactiver la capture du contexte, utilisez la méthode d'extension TaskAsyncEnumerableExtensions.ConfigureAwait. Pour plus d'informations sur les contextes de synchronisation et la capture du contexte actuel, consultez Utilisation du modèle asynchrone basé sur les tâches. Pour plus d'informations sur les flux asynchrones, consultez le didacticiel sur les flux asynchrones.

Type d'une variable d'itération

Vous pouvez utiliser le mot clé `var` pour laisser le compilateur déduire le type d'une variable d'itération dans l'instruction `foreach`, comme le montre le code suivant :

```
C#
```

```
foreach (var item in collection) { }
```

Vous pouvez également spécifier explicitement le type d'une variable d'itération, comme le montre le code suivant :

```
C#
```

```
IEnumerable<T> collection = new T[5];
foreach (V item in collection) { }
```

Dans le formulaire précédent, le type `T` d'un élément de collection doit être implicitement ou explicitement convertible en type `V` d'une variable d'itération. Si une conversion explicite à partir de `T` vers `V` échoue à son temps d'exécution, l'instruction `foreach` lève une InvalidCastException. Par exemple, si `T` est un type de classe non scellé, `V` peut être n'importe quel type d'interface, même celui que `T` n'implémente pas. Au moment de l'exécution, le type d'un élément de collection peut être celui qui dérive de `T` et implémente réellement `V`. Si ce n'est pas le cas, une InvalidCastException est levée.

Instruction `do`

L'instruction `do` exécute une instruction ou un bloc d'instructions tant qu'une expression booléenne donne la valeur `true`. Dans la mesure où cette expression est évaluée après chaque exécution de la boucle, une boucle `do` s'exécute une ou plusieurs fois. La boucle `do` diffère de la boucle while, qui s'exécute zéro ou plusieurs fois.

L'exemple suivant illustre l'utilisation de l'instruction `do` :

C#

```
int n = 0;
do
{
    Console.Write(n);
    n++;
} while (n < 5);
// Output:
// 01234
```

Instruction `while`

L'instruction `while` exécute une instruction ou un bloc d'instructions tant qu'une expression booléenne donne la valeur `true`. Dans la mesure où cette expression est évaluée avant chaque exécution de la boucle, une boucle `while` s'exécute plusieurs fois ou pas du tout. La boucle `while` diffère de la boucle do, qui s'exécute une ou plusieurs fois.

L'exemple suivant illustre l'utilisation de l'instruction `while` :

C#

```
int n = 0;
while (n < 5)
{
    Console.Write(n);
    n++;
}
// Output:
// 01234
```

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Instruction for
- Instruction foreach
- Instruction do
- L'instruction while

Pour plus d'informations sur les fonctionnalités ajoutées dans C# 8.0 et versions ultérieures, consultez les notes de proposition des fonctionnalités suivantes :

- Flux asynchrones (C# 8.0)
- Prise en charge GetEnumerator de l'extension pour les boucles foreach (C# 9.0)

Voir aussi

- Informations de référence sur C#
- Utiliser foreach avec des tableaux
- Itérateurs

Instructions de sélection : `if` `if-else` et `switch`

Article • 05/06/2023

Les instructions `if`, `if-else` et `switch` sélectionnent les instructions à exécuter parmi différents chemins possibles en fonction de la valeur d'une expression. L'ifinstruction exécute une instruction uniquement si une expression booléenne fournie prend la valeur `true`. L'if-else instruction vous permet de choisir les deux chemins de code à suivre en fonction d'une expression booléenne. L'switchinstruction sélectionne une liste d'instructions à exécuter sur la base de critères spéciaux avec une expression.

Instruction `if`

Une instruction `if` peut se présenter sous l'une des deux formes suivantes :

- Une instruction `if` comportant une partie `else` sélectionne l'une des deux instructions à exécuter en fonction de la valeur d'une expression booléenne, comme l'illustre l'exemple suivant :

  C#

  ```
  DisplayWeatherReport(15.0);  // Output: Cold.
  DisplayWeatherReport(24.0);  // Output: Perfect!

  void DisplayWeatherReport(double tempInCelsius)
  {
      if (tempInCelsius < 20.0)
      {
          Console.WriteLine("Cold.");
      }
      else
      {
          Console.WriteLine("Perfect!");
      }
  }
  ```

- Une instruction `if` sans partie `else` n'exécute son corps que si une expression booléenne prend la valeur `true`, comme l'illustre l'exemple suivant :

  C#

  ```
  DisplayMeasurement(45);  // Output: The measurement value is 45
  DisplayMeasurement(-3);  // Output: Warning: not acceptable value! The
  ```

```
measurement value is -3

void DisplayMeasurement(double value)
{
    if (value < 0 || value > 100)
    {
        Console.Write("Warning: not acceptable value! ");
    }

    Console.WriteLine($"The measurement value is {value}");
}
```

Vous pouvez imbriquer des instructions `if` pour vérifier différentes conditions, comme l'illustre l'exemple suivant :

C#

```
DisplayCharacter('f');  // Output: A lowercase letter: f
DisplayCharacter('R');  // Output: An uppercase letter: R
DisplayCharacter('8');  // Output: A digit: 8
DisplayCharacter(',');  // Output: Not alphanumeric character: ,

void DisplayCharacter(char ch)
{
    if (char.IsUpper(ch))
    {
        Console.WriteLine($"An uppercase letter: {ch}");
    }
    else if (char.IsLower(ch))
    {
        Console.WriteLine($"A lowercase letter: {ch}");
    }
    else if (char.IsDigit(ch))
    {
        Console.WriteLine($"A digit: {ch}");
    }
    else
    {
        Console.WriteLine($"Not alphanumeric character: {ch}");
    }
}
```

Dans le contexte d'une expression, vous pouvez utiliser l'opérateur conditionnel ?: pour évaluer l'une des deux expressions en fonction de la valeur d'une expression booléenne.

Instruction `switch`

L'instruction `switch` sélectionne une liste d'instructions à exécuter en fonction de critères spéciaux avec une expression de correspondance, comme l'illustre l'exemple

suivant :

C#

```
DisplayMeasurement(-4);  // Output: Measured value is -4; too low.
DisplayMeasurement(5);  // Output: Measured value is 5.
DisplayMeasurement(30);  // Output: Measured value is 30; too high.
DisplayMeasurement(double.NaN);  // Output: Failed measurement.

void DisplayMeasurement(double measurement)
{
    switch (measurement)
    {
        case < 0.0:
            Console.WriteLine($"Measured value is {measurement}; too low.");
            break;

        case > 15.0:
            Console.WriteLine($"Measured value is {measurement}; too
high.");
            break;

        case double.NaN:
            Console.WriteLine("Failed measurement.");
            break;

        default:
            Console.WriteLine($"Measured value is {measurement}.");
            break;
    }
}
```

Dans l'exemple précédent, l'instruction `switch` utilise les modèles suivants :

- Modèle relationnel (disponible en C# 9.0 et versions ultérieures) : permet de comparer le résultat d'une expression avec une constante.
- Modèle constant : permet de tester si le résultat d'une expression est égal à une constante.

> **ⓘ Important**
>
> Pour plus d'informations sur les modèles pris en charge par l'instruction `switch`, consultez **Modèles**.

L'exemple précédent illustre également la casse `default`. La casse `default` spécifie les instructions à exécuter lorsqu'une expression de correspondance ne correspond à aucun autre modèle de casse. Si une expression de correspondance ne correspond à aucun

modèle de casse et qu'il n'y a pas de casse `default`, le contrôle passe à une instruction `switch`.

Une instruction `switch` exécute la *liste d'instructions* de la première *section switch* dont le *modèle de casse* correspond à une expression de correspondance et dont la protection contre la casse, le cas échéant, prend la valeur `true`. Une instruction `switch` évalue les modèles de casse dans l'ordre du texte, de haut en bas. Le compilateur génère une erreur lorsqu'une instruction `switch` contient une casse inaccessible. Il s'agit d'une casse déjà traitée par une majuscule ou dont le modèle est impossible à mettre en correspondance.

> **Notes**
>
> La casse `default` peut apparaître à n'importe quel emplacement d'une instruction `switch`. Quelle que soit sa position, le cas `default` est évalué uniquement si tous les autres modèles de cas ne sont pas mis en correspondance ou si l'instruction `goto default;` est exécutée dans l'une des sections switch.

Vous pouvez spécifier plusieurs modèles de casse pour une section d'une instruction `switch`, comme l'illustre l'exemple suivant :

C#

```
DisplayMeasurement(-4);  // Output: Measured value is -4; out of an
acceptable range.
DisplayMeasurement(50);  // Output: Measured value is 50.
DisplayMeasurement(132);  // Output: Measured value is 132; out of an
acceptable range.

void DisplayMeasurement(int measurement)
{
    switch (measurement)
    {
        case < 0:
        case > 100:
            Console.WriteLine($"Measured value is {measurement}; out of an
acceptable range.");
            break;

        default:
            Console.WriteLine($"Measured value is {measurement}.");
            break;
    }
}
```

Dans une instruction `switch`, le contrôle ne peut pas passer d'une section switch à une autre. Comme le montrent les exemples de cette section, l'instruction `break` est généralement utilisée à la fin de chaque section switch pour transmettre le contrôle à partir d'une instruction `switch`. Vous pouvez également utiliser les instructions return et throw pour transmettre le contrôle à partir d'une instruction `switch`. Pour imiter le comportement de passage et transmettre le contrôle à d'autres sections switch, vous pouvez utiliser l'instruction goto.

Dans le contexte d'une expression, vous pouvez utiliser l'expression switch pour évaluer une seule expression à partir d'une liste d'expressions candidates basées sur des critères spéciaux avec une expression.

Protections contre la casse

Un modèle de casse peut ne pas être suffisamment expressif pour spécifier la condition d'exécution de la section switch. Dans ce cas, vous pouvez utiliser une *protection contre la casse*. Il s'agit d'une condition supplémentaire qui doit être satisfaite avec un modèle correspondant. Une protection contre la casse doit être une expression booléenne. Vous spécifiez une protection contre la casse après le mot clé `when` qui suit un modèle, comme l'illustre l'exemple suivant :

C#

```
DisplayMeasurements(3, 4);  // Output: First measurement is 3, second
measurement is 4.
DisplayMeasurements(5, 5);  // Output: Both measurements are valid and equal
to 5.

void DisplayMeasurements(int a, int b)
{
    switch ((a, b))
    {
        case (> 0, > 0) when a == b:
            Console.WriteLine($"Both measurements are valid and equal to
{a}.");
            break;

        case (> 0, > 0):
            Console.WriteLine($"First measurement is {a}, second measurement
is {b}.");
            break;

        default:
            Console.WriteLine("One or both measurements are not valid.");
            break;
    }
}
```

L'exemple précédent utilise des modèles positionnels avec des modèles relationnels imbriqués.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Instruction if
- Instruction switch

Pour plus d'informations sur les modèles, consultez la section Modèles et correspondances de modèles de la spécification du langage C#.

Voir aussi

- Informations de référence sur C#
- Opérateur conditionnel ?:
- Opérateurs logiques
- Modèles
- switch, expression
- Ajouter des casses manquantes à l'instruction switch (règle de style IDE0010)

Instructions de saut : `break`, `continue`, `return` et `goto`

Article • 05/06/2023

Les instruction de saut transfèrent le contrôle de manière inconditionnelle. L'instruction break met un terme à l'instruction d'itération englobante la plus proche ou à l'instruction switch. L'instruction continue lance une nouvelle itération de l'instruction d'itération englobante la plus proche. L'instruction return met un terme à l'exécution de la fonction dans laquelle elle apparaît et renvoie le contrôle à l'appelant. L'instructiongoto transfère le contrôle à une instruction marquée par une étiquette.

Pour plus d'informations sur l'instruction `throw` qui lève une exception et transfère aussi de manière inconditionnelle le contrôle, consultez la section Instruction throw de l'article Instructions de gestion des exceptions.

Instruction `break`

L'instruction `break` met un terme à l'instruction d'itération englobante la plus proche (boucle `for`, `foreach`, `while` ou `do`) ou à l'instruction switch. L'instruction `break` transfère le contrôle à l'instruction qui suit l'instruction terminée, le cas échéant.

C#

```
int[] numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
foreach (int number in numbers)
{
    if (number == 3)
    {
        break;
    }

    Console.Write($"{number} ");
}
Console.WriteLine();
Console.WriteLine("End of the example.");
// Output:
// 0 1 2
// End of the example.
```

Dans les boucles imbriquées, l'instruction `break` ne met un terme qu'à la boucle la plus interne qui la contient, comme l'illustre l'exemple suivant :

C#

```
for (int outer = 0; outer < 5; outer++)
{
    for (int inner = 0; inner < 5; inner++)
    {
        if (inner > outer)
        {
            break;
        }

        Console.Write($"{inner} ");
    }
    Console.WriteLine();
}
// Output:
// 0
// 0 1
// 0 1 2
// 0 1 2 3
// 0 1 2 3 4
```

Lorsque vous utilisez l'instruction `switch` au sein d'une boucle, une instruction `break` à la fin d'une section switch transfère uniquement le contrôle en dehors de l'instruction `switch`. La boucle qui contient l'instruction `switch` n'est pas affectée, comme l'illustre l'exemple suivant :

C#

```
double[] measurements = { -4, 5, 30, double.NaN };
foreach (double measurement in measurements)
{
    switch (measurement)
    {
        case < 0.0:
            Console.WriteLine($"Measured value is {measurement}; too low.");
            break;

        case > 15.0:
            Console.WriteLine($"Measured value is {measurement}; too
high.");
            break;

        case double.NaN:
            Console.WriteLine("Failed measurement.");
            break;

        default:
            Console.WriteLine($"Measured value is {measurement}.");
            break;
    }
}
// Output:
```

```
// Measured value is -4; too low.
// Measured value is 5.
// Measured value is 30; too high.
// Failed measurement.
```

Instruction `continue`

L'instruction `continue` lance une nouvelle itération de l'instruction d'itération englobante la plus proche (boucle `for`, `foreach` `while` ou `do`), comme l'illustre l'exemple suivant :

C#

```
for (int i = 0; i < 5; i++)
{
    Console.Write($"Iteration {i}: ");

    if (i < 3)
    {
        Console.WriteLine("skip");
        continue;
    }

    Console.WriteLine("done");
}
// Output:
// Iteration 0: skip
// Iteration 1: skip
// Iteration 2: skip
// Iteration 3: done
// Iteration 4: done
```

Instruction `return`

L'instruction `return` met un terme à l'exécution de la fonction dans laquelle elle apparaît, et renvoie le contrôle et le résultat de la fonction, le cas échéant, à l'appelant.

Si une fonction membre ne calcule pas de valeur, vous devez utiliser l'instruction `return` sans expression, comme l'illustre l'exemple suivant :

C#

```
Console.WriteLine("First call:");
DisplayIfNecessary(6);

Console.WriteLine("Second call:");
```

```
DisplayIfNecessary(5);

void DisplayIfNecessary(int number)
{
    if (number % 2 == 0)
    {
        return;
    }

    Console.WriteLine(number);
}
// Output:
// First call:
// Second call:
// 5
```

Comme le montre l'exemple précédent, l'instruction `return` sans expression est généralement utilisée pour mettre prématurément un terme à une fonction membre. Si une fonction membre ne contient pas l'instruction `return`, elle se termine après l'exécution de sa dernière instruction.

Si une fonction membre calcule une valeur, vous devez utiliser l'instruction `return` avec une expression, comme l'illustre l'exemple suivant :

C#

```
double surfaceArea = CalculateCylinderSurfaceArea(1, 1);
Console.WriteLine($"{surfaceArea:F2}"); // output: 12.57

double CalculateCylinderSurfaceArea(double baseRadius, double height)
{
    double baseArea = Math.PI * baseRadius * baseRadius;
    double sideArea = 2 * Math.PI * baseRadius * height;
    return 2 * baseArea + sideArea;
}
```

Lorsque l'instruction `return` comporte une expression, celle-ci doit être implicitement convertible en type de retour d'une fonction membre, sauf si elle est asynchrone. L'expression renvoyée à partir d'une fonction `async` doit être implicitement convertible en argument de type de Task<TResult> ou ValueTask<TResult>, selon le type de retour de la fonction. Si le type de retour d'une fonction `async` est Task ou ValueTask, vous devez utiliser l'instruction `return` sans expression.

Retours par référence

Par défaut, l'instruction `return` renvoie la valeur d'une expression. Vous pouvez renvoyer une référence à une variable. Pour ce faire, utilisez l'instruction `return` avec le mot clé ref, comme l'illustre l'exemple suivant :

C#

```
var xs = new int[] { 10, 20, 30, 40 };
ref int found = ref FindFirst(xs, s => s == 30);
found = 0;
Console.WriteLine(string.Join(" ", xs));  // output: 10 20 0 40

ref int FindFirst(int[] numbers, Func<int, bool> predicate)
{
    for (int i = 0; i < numbers.Length; i++)
    {
        if (predicate(numbers[i]))
        {
            return ref numbers[i];
        }
    }
    throw new InvalidOperationException("No element satisfies the given
condition.");
}
```

Les valeurs renvoyées peuvent l'être par référence (retours `ref`). Une valeur de retour de référence permet à une méthode de retourner à un appelant une référence à une variable, plutôt qu'à une valeur. L'appelant peut alors choisir de traiter la variable retournée comme si elle était retournée par valeur ou par référence. L'appelant peut créer une nouvelle variable qui est elle-même une référence à la valeur renvoyée, appelée variable ref local. Une *valeur de retour de référence* signifie qu'une méthode retourne une *référence* (ou un alias) à une variable. L'étendue de cette variable doit inclure la méthode. La durée de vie de cette variable doit s'étendre au-delà du retour de la méthode. Des modifications apportées par l'appelant à la valeur de retour de la méthode portent sur la variable qui est retournée par la méthode.

La déclaration qu'une méthode retourne une *valeur de retour de référence* indique que la méthode retourne un alias vers une variable. L'intention de conception est souvent que le code appelant accède à cette variable par le biais de l'alias, y compris à des fins de modification. Le type de retour `void` ne peut pas être utilisé pour les méthodes de retour par référence.

La valeur renvoyée `ref` est l'alias d'une autre variable dans l'étendue de la méthode appelée. Toute utilisation d'une valeur de retour de référence revient à utiliser la variable dont elle est l'alias :

- Lorsque vous lui attribuez une valeur, vous attribuez une valeur à la variable dont elle est l'alias.
- Lorsque vous lisez sa valeur, vous lisez la valeur de la variable dont elle est l'alias.
- Si vous la renvoyez *par référence*, vous renvoyez un alias de cette même variable.
- Si vous la transmettez à une autre méthode *par référence*, vous transmettez une référence à la variable dont elle est l'alias.
- Quand vous créez un alias de variable locale ref, vous créez un nouvel alias vers la même variable.

Un retour par référence doit être *ref_safe_to_escape* pour la méthode d'appel. Cela signifie que :

- La valeur de retour doit avoir une durée de vie qui s'étend au-delà de l'exécution de la méthode. En d'autres termes, il ne peut pas s'agir d'une variable locale dans la méthode de retour. Il peut s'agir d'un champ d'instance ou statique d'une classe, ou bien un argument passé à la méthode. Toute tentative de retour d'une variable locale génère l'erreur de compilation CS8168, « Impossible de renvoyer la variable locale 'obj' par référence, car il ne s'agit pas d'une variable 'ref local' ».
- La valeur renvoyée ne peut pas être le littéral `null`. Une méthode avec retour par référence peut renvoyer l'alias d'une variable dont la valeur est actuellement la valeur (non instanciée) `null` ou un type valeur pouvant accepter la valeur Null.
- La valeur renvoyée ne peut être ni une constante, ni le membre d'une énumération, ni la valeur renvoyée « par valeur » d'une propriété, ni la méthode d'une `class` ou d'un `struct`.

En outre, les valeurs renvoyées par référence ne sont pas autorisées sur les méthodes asynchrones. Une méthode asynchrone peut être retournée avant la fin de son exécution, même si sa valeur de retour est encore inconnue.

Une méthode qui renvoie une *valeur renvoyée par référence* doit :

- Comprendre le mot clé ref devant le type de retour.
- Pour chaque instruction return dans le corps de la méthode, le mot clé ref précède le nom de l'instance retournée.

L'exemple suivant montre une méthode qui remplit ces conditions et retourne une référence à un objet `Person` nommé `p` :

C#

```
public ref Person GetContactInformation(string fname, string lname)
{
    // ...method implementation...
```

```
    return ref p;
}
```

Instruction `goto`

L'instruction `goto` transfère le contrôle à une instruction marquée par une étiquette, comme l'illustre l'exemple suivant :

C#

```
var matrices = new Dictionary<string, int[][]>
{
    ["A"] = new[]
    {
        new[] { 1, 2, 3, 4 },
        new[] { 4, 3, 2, 1 }
    },
    ["B"] = new[]
    {
        new[] { 5, 6, 7, 8 },
        new[] { 8, 7, 6, 5 }
    },
};

CheckMatrices(matrices, 4);

void CheckMatrices(Dictionary<string, int[][]> matrixLookup, int target)
{
    foreach (var (key, matrix) in matrixLookup)
    {
        for (int row = 0; row < matrix.Length; row++)
        {
            for (int col = 0; col < matrix[row].Length; col++)
            {
                if (matrix[row][col] == target)
                {
                    goto Found;
                }
            }
        }
        Console.WriteLine($"Not found {target} in matrix {key}.");
        continue;

    Found:
        Console.WriteLine($"Found {target} in matrix {key}.");
    }
}
// Output:
// Found 4 in matrix A.
// Not found 4 in matrix B.
```

Comme le montre l'exemple précédent, vous pouvez utiliser l'instruction `goto` pour sortir d'une boucle imbriquée.

> **Conseil**
>
> Lorsque vous utilisez des boucles imbriquées, pensez à refactoriser les différentes boucles en méthodes distinctes. Sans l'instruction `goto`, le code sera probablement plus simple et plus lisible.

Vous pouvez également utiliser l'instruction `goto` dans l'instruction switch pour transférer le contrôle vers une section switch avec une étiquette de casse constante, comme l'illustre l'exemple suivant :

C#

```
using System;

public enum CoffeeChoice
{
    Plain,
    WithMilk,
    WithIceCream,
}

public class GotoInSwitchExample
{
    public static void Main()
    {
        Console.WriteLine(CalculatePrice(CoffeeChoice.Plain));  // output:
10.0
        Console.WriteLine(CalculatePrice(CoffeeChoice.WithMilk));  //
output: 15.0
        Console.WriteLine(CalculatePrice(CoffeeChoice.WithIceCream));  //
output: 17.0
    }

    private static decimal CalculatePrice(CoffeeChoice choice)
    {
        decimal price = 0;
        switch (choice)
        {
            case CoffeeChoice.Plain:
                price += 10.0m;
                break;

            case CoffeeChoice.WithMilk:
                price += 5.0m;
                goto case CoffeeChoice.Plain;

            case CoffeeChoice.WithIceCream:
```

```
                price += 7.0m;
                goto case CoffeeChoice.Plain;
            }
            return price;
        }
    }
```

Dans l'instruction `switch`, vous pouvez également utiliser l'instruction `goto default;` pour transférer le contrôle vers la section switch dotée de l'étiquette `default`.

S'il n'existe aucune étiquette portant le nom donné dans la fonction membre actuelle, ou si l'instruction `goto` ne se trouve pas dans l'étendue de l'étiquette, une erreur se produit au moment de la compilation. Autrement dit, vous ne pouvez pas utiliser l'instruction `goto` pour transférer le contrôle en dehors de la fonction membre actuelle ou vers une étendue imbriquée.

spécification du langage C#

Pour plus d'informations, consultez les sections suivantes de la spécification du langage C# :

- Instruction break
- Instruction continue
- Instruction return
- Instruction goto

Voir aussi

- Informations de référence sur C#
- instruction yield

Instruction lock : garantissez un accès exclusif à une ressource partagée

Article • 02/06/2023

L'instruction `lock` obtient le verrou d'exclusion mutuelle d'un objet donné, exécute un bloc d'instructions, puis libère le verrou. Tant qu'un verrou est maintenu, le thread qui contient le verrou peut à nouveau obtenir et libérer le verrou. Tout autre thread se voit bloquer l'obtention du verrou et attend que ce dernier soit libéré. L'instruction `lock` garantit qu'au maximum un seul thread exécute son corps à tout moment.

L'instruction `lock` se présente sous la forme

C#

```
lock (x)
{
    // Your code...
}
```

où `x` est une expression de type référence. Elle équivaut précisément à

C#

```
object __lockObj = x;
bool __lockWasTaken = false;
try
{
    System.Threading.Monitor.Enter(__lockObj, ref __lockWasTaken);
    // Your code...
}
finally
{
    if (__lockWasTaken) System.Threading.Monitor.Exit(__lockObj);
}
```

Dans la mesure où le code utilise une instruction try-finally, le verrou est libéré même si une exception est levée dans le corps d'une instruction `lock`.

Vous ne pouvez pas utiliser l'expression await dans le corps d'une instruction `lock`.

Consignes

Quand vous synchronisez l'accès des threads à une ressource partagée, verrouillez une instance d'objet dédiée (par exemple `private readonly object balanceLock = new object();`) ou toute autre instance peu susceptible d'être utilisée comme objet de verrouillage par des parties du code non associées. Évitez d'utiliser la même instance d'objet de verrouillage pour différentes ressources partagées, car cela peut entraîner une contention d'interblocage ou de verrouillage. En particulier, évitez d'utiliser les instances suivantes en tant qu'objets de verrouillage :

- `this`, qui peut être utilisé en tant que verrou par les appelants.
- les instances Type, car elles peuvent être obtenues par l'opérateur typeof ou par réflexion.
- les instances de chaîne, notamment les littéraux de chaîne, qui peuvent être internés.

Maintenez le verrouillage aussi peu de temps que possible pour réduire les conflits d'accès.

Exemple

L'exemple suivant définit une classe `Account`, qui synchronise l'accès à son champ `balance` privé en verrouillant une instance `balanceLock` dédiée. L'utilisation de la même instance pour le verrouillage permet de garantir que le champ `balance` ne peut pas être mis à jour simultanément par deux threads qui tentent d'appeler les méthodes `Debit` ou `Credit` en même temps.

C#

```
using System;
using System.Threading.Tasks;

public class Account
{
    private readonly object balanceLock = new object();
    private decimal balance;

    public Account(decimal initialBalance) => balance = initialBalance;

    public decimal Debit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "The debit
amount cannot be negative.");
        }

        decimal appliedAmount = 0;
```

```
        lock (balanceLock)
        {
            if (balance >= amount)
            {
                balance -= amount;
                appliedAmount = amount;
            }
        }
        return appliedAmount;
    }

    public void Credit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "The
credit amount cannot be negative.");
        }

        lock (balanceLock)
        {
            balance += amount;
        }
    }

    public decimal GetBalance()
    {
        lock (balanceLock)
        {
            return balance;
        }
    }
}

class AccountTest
{
    static async Task Main()
    {
        var account = new Account(1000);
        var tasks = new Task[100];
        for (int i = 0; i < tasks.Length; i++)
        {
            tasks[i] = Task.Run(() => Update(account));
        }
        await Task.WhenAll(tasks);
        Console.WriteLine($"Account's balance is {account.GetBalance()}");
        // Output:
        // Account's balance is 2000
    }

    static void Update(Account account)
    {
        decimal[] amounts = { 0, 2, -3, 6, -2, -1, 8, -5, 11, -6 };
        foreach (var amount in amounts)
        {
```

```
            if (amount >= 0)
            {
                account.Credit(amount);
            }
            else
            {
                account.Debit(Math.Abs(amount));
            }
        }
    }
}
```

spécification du langage C#

Pour plus d'informations, voir la section Instruction lock de la spécification du langage C#.

Voir aussi

- Informations de référence sur C#
- System.Threading.Monitor
- System.Threading.SpinLock
- System.Threading.Interlocked
- Vue d'ensemble des primitives de synchronisation
- Présentation de System.Threading.Channels

Caractères spéciaux C#

Article • 07/04/2023

Les caractères spéciaux sont des caractères contextuels prédéfinis qui modifient l'élément de programme (chaîne littérale, identificateur ou nom d'attribut) auquel ils sont ajoutés. C# prend en charge les caractères spéciaux suivants :

- @, le caractère identificateur textuel.
- $, le caractère de chaîne interpolé.

Cette section inclut uniquement les jetons qui ne sont pas des opérateurs. Consultez la section Opérateurs pour tous les opérateurs.

Voir aussi

- Référence C#
- Guide de programmation C#

Interpolation de chaîne à l'aide de `$`

Article • 03/09/2023

Le caractère `$` identifie un littéral de chaîne comme une *chaîne interpolée*. Une chaîne interpolée est un littéral de chaîne qui peut contenir des *expressions d'interpolation*. Quand une chaîne interpolée est résolue en une chaîne de résultat, les éléments avec des expressions d'interpolation sont remplacés par les représentations sous forme de chaîne des résultats des expressions.

L'interpolation de chaîne fournit une syntaxe plus lisible et plus pratique pour la mise en forme des chaînes. Cette syntaxe est plus facile à lire que la mise en forme composite de chaîne. L'exemple suivant utilise les deux fonctionnalités pour produire la même sortie :

```
C#

var name = "Mark";
var date = DateTime.Now;

// Composite formatting:
Console.WriteLine("Hello, {0}! Today is {1}, it's {2:HH:mm} now.", name,
date.DayOfWeek, date);
// String interpolation:
Console.WriteLine($"Hello, {name}! Today is {date.DayOfWeek}, it's
{date:HH:mm} now.");
// Both calls produce the same output that is similar to:
// Hello, Mark! Today is Wednesday, it's 19:40 now.
```

À partir de C# 10, vous pouvez utiliser une chaîne interpolée pour initialiser une chaîne constante. Vous pouvez le faire uniquement si toutes les expressions d'interpolation au sein de la chaîne interpolée sont également des chaînes constantes.

Structure d'une chaîne interpolée

Pour identifier un littéral de chaîne comme chaîne interpolée, préfixez-la du symbole `$`. N'ajoutez pas d'espace blanc entre les signes `$` et `"` au début d'un littéral de chaîne.

La structure d'un élément avec une expression d'interpolation se présente comme suit :

```
C#

{<interpolationExpression>[,<alignment>][:<formatString>]}
```

Les éléments entre crochets sont facultatifs. Le tableau suivant décrit chaque élément :

Élément	Description
`interpolationExpression`	Expression qui produit un résultat à mettre en forme. String.Empty constitue une représentation sous forme de chaîne de `null`.
`alignment`	Expression constante dont la valeur définit le nombre minimal de caractères dans la représentation sous forme de chaîne du résultat de l'expression. Si le nombre est positif, la représentation sous forme de chaîne est alignée à droite ; s'il est négatif, elle est alignée à gauche. Pour plus d'informations, consultez la section Composant d'alignement de l'article Mise en forme composite.
`formatString`	Chaîne de format qui est prise en charge par le type de résultat de l'expression. Pour plus d'informations, consultez la section Composant de chaîne de mise en forme de l'article Mise en forme composite.

L'exemple suivant utilise les composants de mise en forme facultatifs décrits ci-dessus :

C#

```
Console.WriteLine($"|{"Left",-7}|{"Right",7}|");

const int FieldWidthRightAligned = 20;
Console.WriteLine($"{Math.PI,FieldWidthRightAligned} - default formatting of
the pi number");
Console.WriteLine($"{Math.PI,FieldWidthRightAligned:F3} - display only three
decimal digits of the pi number");
// Output is:
// |Left    |  Right|
//     3.14159265358979 - default formatting of the pi number
//                3.142 - display only three decimal digits of the pi number
```

À compter de C# 11, vous pouvez utiliser de nouvelles lignes dans une expression d'interpolation pour rendre le code de l'expression plus lisible. L'exemple suivant montre comment les nouvelles lignes peuvent améliorer la lisibilité d'une expression impliquant des critères spéciaux :

C#

```
string message = $"The usage policy for {safetyScore} is {
    safetyScore switch
    {
        > 90 => "Unlimited usage",
        > 80 => "General usage, with daily safety check",
        > 70 => "Issues must be addressed within 1 week",
        > 50 => "Issues must be addressed within 1 day",
        _ => "Issues must be addressed before continued use",
    }
    }";
```

Littéraux de chaîne brute interpolés

À compter de C# 11, vous pouvez utiliser un littéral de chaîne brute interpolé, comme l'illustre l'exemple suivant :

C#

```
int X = 2;
int Y = 3;

var pointMessage = $"""The point "{X}, {Y}" is {Math.Sqrt(X * X + Y * Y):F3}
from the origin""";

Console.WriteLine(pointMessage);
// Output is:
// The point "2, 3" is 3.606 from the origin
```

Pour incorporer les caractères `{` et `}` dans la chaîne de résultat, démarrez un littéral de chaîne brut interpolé avec plusieurs caractères `$`. Lorsque vous faites cela, toute séquence de caractères `{` ou `}` plus courte que le nombre de caractères `$` est incorporée dans la chaîne de résultat. Pour placer une expression d'interpolation dans cette chaîne, vous devez utiliser le même nombre d'accolades que le nombre de caractères `$`, comme illustre l'exemple suivant :

C#

```
int X = 2;
int Y = 3;

var pointMessage = $$"""{The point {{{X}}, {{Y}}} is {{Math.Sqrt(X * X + Y *
Y):F3}} from the origin}""";
Console.WriteLine(pointMessage);
// Output is:
// {The point {2, 3} is 3.606 from the origin}
```

Dans l'exemple précédent, un littéral de chaîne brute interpolé commence par deux caractères `$`. C'est pourquoi vous devez placer chaque expression d'interpolation entre accolades doubles `{{` et `}}`. Une seule accolade est incorporée dans une chaîne de résultat. Si vous devez incorporer des caractères répétés `{` ou `}` dans une chaîne de résultat, utilisez un nombre `$` de caractères suffisamment élevé pour désigner un littéral de chaîne brute interpolé.

Caractères spéciaux

Pour ajouter une accolade, « { » ou « } », dans le texte produit par une chaîne interpolée, entrez deux accolades, « {{ » ou « }} ». Pour plus d'informations, consultez la section Accolades d'échappement de l'article Mise en forme composite.

Comme les deux-points (« : ») ont une signification particulière dans un élément d'expression d'interpolation, pour utiliser un opérateur conditionnel dans une expression d'interpolation, mettez cette expression entre parenthèses.

L'exemple suivant montre comment inclure une accolade dans une chaîne de résultats. Il montre également comment utiliser un opérateur conditionnel :

C#

```
string name = "Horace";
int age = 34;
Console.WriteLine($"He asked, \"Is your name {name}?\", but didn't wait for
a reply :-{{");
Console.WriteLine($"{name} is {age} year{(age == 1 ? "" : "s")} old.");
// Output is:
// He asked, "Is your name Horace?", but didn't wait for a reply :-{
// Horace is 34 years old.
```

Une chaîne verbatim interpolée commence par les caractères `$` et `@`. Vous pouvez utiliser `$` et `@` dans n'importe quel ordre : `$@"..."` et `@$"..."` sont toutes deux des chaînes verbatim interpolées valides. Pour plus d'informations sur les chaînes verbatim, consultez les articles Chaîne et Identificateur verbatim.

Mise en forme spécifique à la culture

Par défaut, une chaîne interpolée utilise la culture active définie par la propriété CultureInfo.CurrentCulture pour toutes les opérations de mise en forme.

Pour résoudre une chaîne interpolée en chaîne de résultat spécifique à la culture, utilisez la méthode String.Create(IFormatProvider, DefaultInterpolatedStringHandler), qui est disponible à partir de .NET 6. L'exemple suivant montre comment effectuer cette opération :

C#

```
double speedOfLight = 299792.458;

System.Globalization.CultureInfo.CurrentCulture =
System.Globalization.CultureInfo.GetCultureInfo("nl-NL");
string messageInCurrentCulture = $"The speed of light is {speedOfLight:N3}
km/s.";
```

```
var specificCulture = System.Globalization.CultureInfo.GetCultureInfo("en-
IN");
string messageInSpecificCulture = string.Create(
    specificCulture, $"The speed of light is {speedOfLight:N3} km/s.");

string messageInInvariantCulture = string.Create(
    System.Globalization.CultureInfo.InvariantCulture, $"The speed of light
is {speedOfLight:N3} km/s.");

Console.WriteLine($"{System.Globalization.CultureInfo.CurrentCulture,-10}
{messageInCurrentCulture}");
Console.WriteLine($"{specificCulture,-10} {messageInSpecificCulture}");
Console.WriteLine($"{"Invariant",-10} {messageInInvariantCulture}");
// Output is:
// nl-NL      The speed of light is 299.792,458 km/s.
// en-IN      The speed of light is 2,99,792.458 km/s.
// Invariant  The speed of light is 299,792.458 km/s.
```

Dans .NET 5 et versions antérieures de .NET, utilisez la conversion implicite d'une chaîne interpolée en une instance FormattableString . Ensuite, vous pouvez utiliser une méthode d'instance FormattableString.ToString(IFormatProvider) ou une méthode statique FormattableString.Invariant pour produire une chaîne de résultat spécifique à la culture. L'exemple suivant montre comment effectuer cette opération :

C#

```
double speedOfLight = 299792.458;
FormattableString message = $"The speed of light is {speedOfLight:N3}
km/s.";

var specificCulture = System.Globalization.CultureInfo.GetCultureInfo("en-
IN");
string messageInSpecificCulture = message.ToString(specificCulture);
Console.WriteLine(messageInSpecificCulture);
// Output:
// The speed of light is 2,99,792.458 km/s.

string messageInInvariantCulture = FormattableString.Invariant(message);
Console.WriteLine(messageInInvariantCulture);
// Output is:
// The speed of light is 299,792.458 km/s.
```

Pour plus d'informations sur la lise en forma personnalisée, consultez la section Mise en forme personnalisée avec ICustomFormatter de l'article Types de mise en forme dans .NET.

Autres ressources

Si vous débutez en matière d'interpolation de chaîne, consultez le tutoriel interactif Interpolation de chaîne en C#. Vous pouvez également consulter un autre tutoriel Interpolation de chaîne en C#. Ce tutoriel montre comment utiliser des chaînes interpolées pour produire des chaînes mises en forme.

Compilation de chaînes interpolées

À compter de C# 10 et .NET 6, le compilateur vérifie si une chaîne interpolée est affectée à un type qui satisfait au *modèle de gestionnaire de chaîne interpolé*. Un *gestionnaire de chaîne interpolée* est un type qui convertit la chaîne interpolée en chaîne de résultat. Lorsqu'une chaîne interpolée a le type `string`, elle est traitée par System.Runtime.CompilerServices.DefaultInterpolatedStringHandler. Pour obtenir l'exemple d'un gestionnaire de chaînes interpolé personnalisé, consultez le didacticiel Écrire un gestionnaire d'interpolation de chaîne personnalisé. L'utilisation d'un gestionnaire de chaîne interpolée est un scénario avancé, généralement nécessaire pour des raisons de performances.

> ⓘ **Notes**
>
> L'un des effets secondaires des gestionnaires de chaînes interpolées est que, un gestionnaire personnalisé, y compris **System.Runtime.CompilerServices.DefaultInterpolatedStringHandler**, peut ne pas évaluer toutes les expressions d'interpolations dans la chaîne interpolée dans toutes les conditions. Cela peut se traduire par l'absence d'effets secondaires dans ces expressions.

Dans C# 9 et précédents, si une chaîne interpolée est de type `string`, elle est généralement transformée en un appel de méthode String.Format. Le compilateur peut remplacer String.Format par String.Concat si le comportement analysé équivaut à une concaténation.

Si une chaîne interpolée est de type IFormattable ou FormattableString, le compilateur génère un appel à la méthode FormattableStringFactory.Create.

spécification du langage C#

Pour plus d'informations, consultez la section Expressions des chaînes interpolées de la Spécification du langage C# et les nouvelles spécifications de fonctionnalités suivantes :

- C# 10 : chaînes interpolées améliorées

- C# 11 - Les littéraux de chaîne bruts
- C# 11 : nouvelles lignes dans les interpolations de chaînes

Voir aussi

- Informations de référence sur C#
- Caractères spéciaux C#
- Chaînes
- Chaînes de format numériques standard
- Mise en forme composite
- String.Format
- Simplifier l'interpolation (règle de style IDE0071)
- Interpolation de chaîne dans C# 10 et .NET 6 (blog .NET)

Texte détaillé : `@` dans les variables, les attributs et les littéraux de chaîne

Article • 27/03/2023

Le caractère spécial `@` sert d'identificateur de chaîne textuelle. Vous l'utilisez des façons suivantes :

1. Pour indiquer qu'un littéral de chaîne doit être interprété textuellement. Le caractère `@` dans cette instance définit un *littéral de chaîne textuelle*. Les séquences d'échappement simples (comme `"\\"` pour une barre oblique inverse), hexadécimales (comme `"\x0041"` pour un A majuscule) et Unicode (comme `"\u0041"` pour un A majuscule), sont interprétées littéralement. Seule une séquence d'échappement de guillemet (`""`) n'est pas interprétée littéralement ; elle génère un guillemet double. En outre, dans le cas d'une chaîne interpolée textuelle, les séquences d'échappement des accolades (`{{` et `}}`) ne sont pas interprétées littéralement ; elles produisent une seule accolade. L'exemple suivant définit deux chemins de fichier identiques, un à l'aide d'un littéral de chaîne standard et l'autre à l'aide d'un littéral de chaîne textuelle. Il s'agit là de l'une des utilisations les plus courantes de littéraux de chaîne textuelle.

   C#

   ```
   string filename1 = @"c:\documents\files\u0066.txt";
   string filename2 = "c:\\documents\\files\\u0066.txt";

   Console.WriteLine(filename1);
   Console.WriteLine(filename2);
   // The example displays the following output:
   //     c:\documents\files\u0066.txt
   //     c:\documents\files\u0066.txt
   ```

   L'exemple suivant illustre l'effet de la définition d'un littéral de chaîne standard et d'un littéral de chaîne textuelle qui contiennent des séquences de caractères identiques.

   C#

   ```
   string s1 = "He said, \"This is the last \u0063hance\x0021\"";
   string s2 = @"He said, ""This is the last \u0063hance\x0021""";

   Console.WriteLine(s1);
   Console.WriteLine(s2);
   // The example displays the following output:
   ```

```
//     He said, "This is the last chance!"
//     He said, "This is the last \u0063hance\x0021"
```

2. Pour utiliser des mots clés C# en tant qu'identificateurs. Le caractère `@` est le préfixe d'un élément de code que le compilateur doit interpréter comme un identificateur plutôt qu'un mot clé C#. L'exemple suivant utilise le caractère `@` pour définir un identificateur nommé `for` qu'il utilise dans une boucle `for`.

```
C#

string[] @for = { "John", "James", "Joan", "Jamie" };
for (int ctr = 0; ctr < @for.Length; ctr++)
{
   Console.WriteLine($"Here is your gift, {@for[ctr]}!");
}
// The example displays the following output:
//     Here is your gift, John!
//     Here is your gift, James!
//     Here is your gift, Joan!
//     Here is your gift, Jamie!
```

3. Pour permettre au compilateur de faire la distinction entre les attributs en cas de conflit de noms. Un attribut est une classe qui dérive de Attribute. Son nom de type comprend généralement le suffixe **Attribute**, bien que le compilateur n'applique pas cette convention. L'attribut peut ensuite être référencé dans le code par son nom de type complet (par exemple, `[InfoAttribute]`) ou son nom abrégé (par exemple, `[Info]`). Toutefois, un conflit survient si deux noms de type d'attribut abrégés sont identiques et que l'un d'eux inclut le suffixe **Attribute**, mais pas l'autre. Par exemple, la compilation du code suivant échoue, car le compilateur ne peut pas déterminer si l'attribut `Info` ou `InfoAttribute` est appliqué à la classe `Example`. Pour plus d'informations, consultez CS1614.

```
C#

using System;

[AttributeUsage(AttributeTargets.Class)]
public class Info : Attribute
{
   private string information;

   public Info(string info)
   {
      information = info;
   }
}
```

```
[AttributeUsage(AttributeTargets.Method)]
public class InfoAttribute : Attribute
{
   private string information;

   public InfoAttribute(string info)
   {
      information = info;
   }
}

[Info("A simple executable.")] // Generates compiler error CS1614.
Ambiguous Info and InfoAttribute.
// Prepend '@' to select 'Info' ([@Info("A simple executable.")]).
Specify the full name 'InfoAttribute' to select it.
public class Example
{
   [InfoAttribute("The entry point.")]
   public static void Main()
   {
   }
}
```

Voir aussi

- Référence C#
- Guide de programmation C#
- Caractères spéciaux C#

Attributs au niveau de l'assembly interprétés par le compilateur C#

Article • 10/05/2023

La plupart des attributs sont appliqués à des éléments de langage spécifiques, tels que les classes ou les méthodes. Toutefois, certains attributs sont globaux : ils s'appliquent à la totalité d'un assembly ou d'un module. Par exemple, l'attribut AssemblyVersionAttribute peut être utilisé pour incorporer des informations de version dans un assembly, de la manière suivante :

C#

```
[assembly: AssemblyVersion("1.0.0.0")]
```

Les attributs globaux apparaissent dans le code source après toute directive `using` de niveau supérieur et avant toute déclaration de type, de module ou d'espace de noms. Les attributs globaux peuvent apparaître dans plusieurs fichiers sources, mais les fichiers doivent être compilés en un seul passage. Visual Studio ajoute des attributs globaux au fichier AssemblyInfo.cs dans les projets .NET Framework. Ces attributs ne sont pas ajoutés aux projets .NET Core.

Les attributs d'assembly sont des valeurs qui fournissent des informations sur un assembly. Ils sont répartis dans les catégories suivantes :

- Attributs d'identité de l'assembly
- Attributs d'informations
- Attributs de manifeste de l'assembly

Attributs d'identité de l'assembly

Trois attributs (avec un nom fort, le cas échéant), déterminent l'identité d'un assembly : nom, version et culture. Ces attributs constituent le nom complet de l'assembly et sont obligatoires quand vous le référencez le code. Vous pouvez définir la version et la culture d'un assembly en utilisant des attributs. Toutefois, la valeur de nom est définie par le compilateur, l'IDE de Visual Studio dans la boîte de dialogue Informations sur l'assembly ou l'utilitaire Assembly Linker (Al.exe) lors de la création de l'assembly. Le nom de l'assembly est basé sur le manifeste de l'assembly. L'attribut AssemblyFlagsAttribute spécifie si plusieurs copies de l'assembly peuvent coexister.

Le tableau suivant présente les attributs d'identité.

Attribut	Objectif
AssemblyVersionAttribute	Spécifie la version d'un assembly.
AssemblyCultureAttribute	Spécifie la culture prise en charge par l'assembly.
AssemblyFlagsAttribute	Spécifie si un assembly prend en charge l'exécution côte à côte sur le même ordinateur, dans le même processus ou dans le même domaine d'application.

Attributs d'informations

Utilisez les attributs d'informations pour fournir des informations supplémentaires sur le produit ou la société de l'assembly. Le tableau suivant présente les attributs d'informations définis dans l'espace de noms System.Reflection.

Attribut	Objectif
AssemblyProductAttribute	Spécifie un nom de produit pour un manifeste d'assembly.
AssemblyTrademarkAttribute	Spécifie une marque de commerce pour un manifeste d'assembly.
AssemblyInformationalVersionAttribute	Spécifie une version d'informations pour un manifeste d'assembly.
AssemblyCompanyAttribute	Spécifie un nom de société pour un manifeste d'assembly.
AssemblyCopyrightAttribute	Définit un attribut personnalisé qui spécifie un copyright pour un manifeste d'assembly.
AssemblyFileVersionAttribute	Définit un numéro de version spécifique pour la ressource de la version du fichier Win32.
CLSCompliantAttribute	Indique si l'assembly est conforme à la spécification CLS (Common Language Specification).

Attributs de manifeste de l'assembly

Vous pouvez utiliser les attributs de manifeste de l'assembly pour fournir des informations dans le manifeste de l'assembly. Les attributs incluent le titre, la description, l'alias par défaut et la configuration. Le tableau suivant présente les attributs de manifeste de l'assembly définis dans l'espace de noms System.Reflection.

Attribut	Objectif
AssemblyTitleAttribute	Spécifie un titre d'assembly pour un manifeste d'assembly.
AssemblyDescriptionAttribute	Spécifie une description d'assembly pour un manifeste d'assembly.
AssemblyConfigurationAttribute	Spécifie une configuration d'assembly (telles que retail ou debug) pour un manifeste d'assembly.
AssemblyDefaultAliasAttribute	Définit un alias par défaut convivial pour un manifeste d'assembly.

Déterminer les informations de l'appelant à l'aide d'attributs interprétés par le compilateur C#

Article • 07/04/2023

À l'aide des attributs d'informations, vous pouvez obtenir des informations sur l'appelant d'une méthode. Vous pouvez obtenir le chemin d'accès du fichier de code source, le numéro de ligne dans le code source, et le nom du membre de l'appelant. Pour obtenir des informations de membre de l'appelant, vous utilisez les attributs qui sont appliqués aux paramètres facultatifs. Chaque paramètre facultatif spécifie une valeur par défaut. Le tableau suivant répertorie les attributs d'informations de l'appelant définis dans l'espace de noms System.Runtime.CompilerServices :

Attribut	Description	Type
CallerFilePathAttribute	Chemin d'accès complet du fichier source qui contient l'appelant. Le chemin d'accès complet est le chemin d'accès au moment de la compilation.	`String`
CallerLineNumberAttribute	Numéro de ligne dans le fichier source à partir duquel la méthode est appelée.	`Integer`
CallerMemberNameAttribute	Nom de la méthode ou nom de la propriété de l'appelant.	`String`
CallerArgumentExpressionAttribute	Représentation sous forme de chaîne de l'expression d'argument.	`String`

Ces informations vous aident à effectuer un traçage et à déboguer. Elles vous aident également à créer des outils de diagnostic. L'exemple suivant indique comment utiliser des attributs d'informations de l'appelant. À chaque appel à la méthode `TraceMessage`, les informations de l'appelant sont ajoutées aux arguments des paramètres optionnels.

C#

```
public void DoProcessing()
{
    TraceMessage("Something happened.");
}

public void TraceMessage(string message,
        [CallerMemberName] string memberName = "",
        [CallerFilePath] string sourceFilePath = "",
```

```
        [CallerLineNumber] int sourceLineNumber = 0)
{
    Trace.WriteLine("message: " + message);
    Trace.WriteLine("member name: " + memberName);
    Trace.WriteLine("source file path: " + sourceFilePath);
    Trace.WriteLine("source line number: " + sourceLineNumber);
}

// Sample Output:
//  message: Something happened.
//  member name: DoProcessing
//  source file path: c:\Visual Studio
Projects\CallerInfoCS\CallerInfoCS\Form1.cs
//  source line number: 31
```

Vous devez spécifier une valeur par défaut explicite pour chaque paramètre optionnel. Vous ne pouvez pas appliquer des attributs d'informations de l'appelant aux paramètres qui ne sont pas spécifiés comme facultatifs. Les attributs d'informations de l'appelant ne rendent pas un paramètre facultatif. À la place, ils affectent la valeur par défaut qui est passée si l'argument est oublié. Les valeurs d'informations de l'appelant sont émises en tant que littéraux en langage intermédiaire (IL) au moment de la compilation. Contrairement aux résultats de la propriété StackTrace pour les exceptions, les résultats ne sont pas affectés par l'obfuscation (protection de code). Vous pouvez fournir explicitement les arguments facultatifs pour contrôler ou masquer des informations de l'appelant.

Noms de membres

Vous pouvez utiliser l'attribut `CallerMemberName` pour éviter de spécifier le nom du membre comme argument de `String` à la méthode appelée. Vous évitez ainsi le problème que la **refactorisation de changement de nom** ne modifie pas les valeurs `String`. Cet avantage est particulièrement utile pour les tâches suivantes :

- Utilisation du traçage et des programmes de diagnostic.
- Implémentation de l'interface INotifyPropertyChanged lors de la liaison de données. Cette interface permet à la propriété d'un objet de signaler à un contrôle dépendant que la propriété a été modifiée. Le contrôle peut afficher les informations mises à jour. Sans attribut `CallerMemberName`, vous devez spécifier le nom de la propriété comme littéral.

Le graphique suivant affiche les noms des membres qui sont retournés lorsque vous utilisez l'attribut `CallerMemberName`.

Les appels se produisent à l'intérieur de/d'	Résultat de nom de membre
Méthode, propriété ou événement	Le nom de la méthode, la propriété ou l'événement dont l'appel est originaire.
Constructeur	La chaîne « .ctor »
Constructeur statique	La chaîne « .cctor »
Finaliseur	La chaîne « finalize »
Opérateurs définis par l'utilisateur ou conversions	Le nom généré pour le membre, par exemple, « op_Addition ».
Constructeur d'attribut	Le nom de la méthode ou de la propriété à laquelle s'applique l'attribut. Si l'attribut est un élément dans un membre (tel qu'un paramètre, une valeur de retour, ou un paramètre de type générique), le résultat est le nom du membre qui est associé à cet élément.
Aucun membre contenant (par exemple, niveau assembly ou attributs qui sont appliqués aux types)	Valeur par défaut du paramètre optionnel.

Expressions d'argument

Vous utilisez le System.Runtime.CompilerServices.CallerArgumentExpressionAttribute quand vous souhaitez que l'expression passe au statut d'argument. Les bibliothèques de diagnostic peuvent vouloir fournir plus de détails sur les *expressions* passées au statut d'arguments. En fournissant l'expression qui a déclenché le diagnostic, en plus du nom du paramètre, les développeurs ont plus de détails sur la condition qui a déclenché le diagnostic. Ces informations supplémentaires facilitent la correction.

L'exemple suivant montre comment fournir des informations détaillées sur l'argument lorsqu'il n'est pas valide :

C#

```
public static void ValidateArgument(string parameterName, bool condition,
[CallerArgumentExpression("condition")] string? message=null)
{
    if (!condition)
    {
        throw new ArgumentException($"Argument failed validation:
```

```
<{message}>", parameterName);
    }
}
```

Vous pouvez l'appeler comme indiqué dans l'exemple suivant :

C#

```
public void Operation(Action func)
{
    Utilities.ValidateArgument(nameof(func), func is not null);
    func();
}
```

L'expression utilisée pour la `condition` est injectée par le compilateur dans l'argument `message`. Lorsqu'un développeur appelle une `Operation` avec un argument `null`, le message suivant est stocké dans `ArgumentException` :

CLI .NET

```
Argument failed validation: <func is not null>
```

Cet attribut vous permet de créer des utilitaires de diagnostic qui fournissent plus de détails. Les développeurs peuvent comprendre plus rapidement les modifications nécessaires. Vous pouvez également utiliser le CallerArgumentExpressionAttribute pour déterminer quelle expression a été utilisée comme récepteur pour les méthodes d'extension. La méthode suivante échantillonne une séquence à intervalles réguliers. Si la séquence a moins d'éléments que la fréquence, elle signale une erreur :

C#

```
public static IEnumerable<T> Sample<T>(this IEnumerable<T> sequence, int
frequency,
    [CallerArgumentExpression(nameof(sequence))] string? message = null)
{
    if (sequence.Count() < frequency)
        throw new ArgumentException($"Expression doesn't have enough
elements: {message}", nameof(sequence));
    int i = 0;
    foreach (T item in sequence)
    {
        if (i++ % frequency == 0)
            yield return item;
    }
}
```

L'exemple précédent utilise l'opérateur nameof pour le paramètre `sequence`. Cette fonctionnalité est disponible dans C# 11. Avant C# 11, vous devez taper le nom du paramètre sous forme de chaîne. Vous pouvez appeler cette méthode comme suit :

C#

```
sample = Enumerable.Range(0, 10).Sample(100);
```

L'exemple précédent lève une classe d'ArgumentException dont le message correspond au texte suivant :

CLI .NET

```
Expression doesn't have enough elements: Enumerable.Range(0, 10) (Parameter
'sequence')
```

Voir aussi

- Arguments nommés et facultatifs
- System.Reflection
- Attribute
- Attributs

Attributs pour l'analyse statique d'état null interprétés par le compilateur C#

Article • 01/06/2023

Dans un contexte prenant en compte les types nullables, le compilateur effectue une analyse statique du code pour déterminer l'*état null* de toutes les variables de type référence :

- *not-null* : l'analyse statique détermine qu'une variable a une valeur non-null.
- *maybe-null* : l'analyse statique ne peut pas déterminer qu'une variable a une valeur non-null.

Ces états permettent au compilateur de fournir des avertissements lorsque vous pouvez déréférencer une valeur null, en levant une exception System.NullReferenceException. Ces attributs fournissent au compilateur des informations sémantiques sur l'*état null* des arguments, des valeurs de retour et des membres d'objet en fonction de l'état des arguments et des valeurs de retour. Le compilateur fournit des avertissements plus précis quand vos API ont été correctement annotées avec ces informations sémantiques.

Cet article fournit une brève description de chaque attribut de type référence nullable et de la manière de l'utiliser.

Commençons avec un exemple. Imaginez que votre bibliothèque dispose de l'API suivante pour récupérer une chaîne de ressource. Cette méthode a été compilée à l'origine dans un contexte *inconscient de nullable* :

C#

```
bool TryGetMessage(string key, out string message)
{
    if (_messageMap.ContainsKey(key))
        message = _messageMap[key];
    else
        message = null;
    return message != null;
}
```

L'exemple précédent suit le modèle `Try*` familier dans .NET. Il existe deux paramètres de référence pour cette API : `key` et `message`. Cette API a les règles suivantes relatives à *l'état null* de ces paramètres :

- Les appelants ne doivent pas transmettre `null` comme argument pour `key`.

- Les appelants peuvent transmettre une variable dont la valeur est `null` comme argument pour `message`.
- Si la méthode `TryGetMessage` retourne `true`, la valeur de `message` n'est pas null. Si la valeur de retour est `false`, la valeur de `message` est null.

La règle pour `key` peut être exprimée succinctement : `key` doit être un type référence non-nullable. Le paramètre `message` est plus complexe. Il autorise une variable qui est `null` comme argument, mais garantit, en cas de réussite, que l'argument `out` n'est pas `null`. Pour ces scénarios, vous avez besoin d'un vocabulaire plus riche pour décrire les attentes. L'attribut `NotNullWhen`, décrit ci-dessous, décrit l'*état null* de l'argument utilisé pour le paramètre `message`.

> **Notes**
>
> L'ajout de ces attributs donne au compilateur plus d'informations sur les règles pour votre API. Quand le code d'appel est compilé dans un contexte prenant en compte les types nullables, le compilateur avertit les appelants quand ils enfreignent ces règles. Ces attributs n'activent pas davantage de vérifications sur votre implémentation.

Attribut	Category	Signification
AllowNull	Precondition	Un paramètre, un champ ou une propriété non-nullable peut être null.
DisallowNull	Precondition	Un paramètre, un champ ou une propriété nullable ne doit jamais être null.
MaybeNull	Postcondition	Un paramètre, un champ, une propriété ou une valeur de retour non-nullable peut être null.
NotNull	Postcondition	Un paramètre, un champ, une propriété ou une valeur de retour nullable ne sera jamais null.
MaybeNullWhen	Post-condition conditionnelle	Un argument non-nullable peut être null quand la méthode retourne la valeur `bool` spécifiée.
NotNullWhen	Post-condition conditionnelle	Un argument nullable n'est pas null quand la méthode retourne la valeur `bool` spécifiée.
NotNullIfNotNull	Post-condition conditionnelle	Une valeur de retour, une propriété ou un argument n'est pas null si l'argument du paramètre spécifié n'est pas null.

Attribut	Category	Signification
MemberNotNull	Méthodes d'assistance de méthode et de propriété	Le membre listé n'est pas null quand la méthode est retournée.
MemberNotNullWhen	Méthodes d'assistance de méthode et de propriété	Le membre listé n'est pas null quand la méthode retourne la valeur `bool` spécifiée.
DoesNotReturn	Code inaccessible	Une méthode ou une propriété ne retourne jamais. En d'autres termes, elle lève toujours une exception.
DoesNotReturnIf	Code inaccessible	Cette méthode ou propriété ne retourne jamais si le paramètre `bool` associé a la valeur spécifiée.

Les descriptions précédentes sont une référence rapide à ce que fait chaque attribut. Les sections suivantes décrivent plus en détail le comportement et la signification de ces attributs.

Conditions préalables : `AllowNull` et `DisallowNull`

Prenons l'exemple d'une propriété en lecture/écriture qui ne retourne jamais `null`, car elle a une valeur par défaut raisonnable. Les appelants transmettent `null` à l'accesseur set lorsqu'ils le définissent sur cette valeur par défaut. Par exemple, prenons l'exemple d'un système de messagerie qui demande un nom d'écran dans une salle de conversation. Si aucun n'est fourni, le système génère un nom aléatoire :

C#

```
public string ScreenName
{
    get => _screenName;
    set => _screenName = value ?? GenerateRandomScreenName();
}
private string _screenName;
```

Quand vous compilez le code précédent dans un contexte inconscient de nullable, tout se passe correctement. Une fois que vous activez les types référence nullables, la propriété `ScreenName` devient une référence non-nullable. Ceci est correct pour

l'accesseur `get` : il ne retourne jamais `null`. Les appelants n'ont pas besoin de vérifier la propriété retournée pour `null`. Mais maintenant, la définition de la propriété sur `null` génère un avertissement. Pour prendre en charge ce type de code, vous ajoutez l'attribut System.Diagnostics.CodeAnalysis.AllowNullAttribute à la propriété, comme indiqué dans le code suivant :

C#

```
[AllowNull]
public string ScreenName
{
    get => _screenName;
    set => _screenName = value ?? GenerateRandomScreenName();
}
private string _screenName = GenerateRandomScreenName();
```

Vous devrez peut-être ajouter une directive `using` pour System.Diagnostics.CodeAnalysis afin d'utiliser les attribut décrits dans cet article. L'attribut est appliqué à la propriété, et non pas à l'accesseur `set`. L'attribut `AllowNull` spécifie des *conditions préalables* et s'applique uniquement aux arguments. L'accesseur `get` a une valeur de retour, mais aucun paramètre. Par conséquent, l'attribut `AllowNull` s'applique uniquement à l'accesseur `set`.

L'exemple précédent montre ce qu'il faut rechercher lors de l'ajout de l'attribut `AllowNull` sur un argument :

1. Le contrat général pour cette variable est qu'elle ne doit pas être `null`, vous souhaitez donc un type référence non-nullable.
2. Il existe des scénarios pour qu'un appelant transmette `null` comme argument, bien qu'il ne s'agisse pas de l'utilisation la plus courante.

Le plus souvent, vous aurez besoin de cet attribut pour les propriétés, ou les arguments `in`, `out` et `ref`. L'attribut `AllowNull` est le meilleur choix quand une variable n'est généralement pas null, mais vous devez autoriser `null` comme condition préalable.

Comparez cela avec les scénarios d'utilisation de `DisallowNull` : vous utilisez cet attribut pour spécifier qu'un argument d'un type référence nullable ne doit pas être `null`. Prenons l'exemple d'une propriété où `null` est la valeur par défaut, mais les clients peuvent uniquement la définir sur une valeur non-null. Prenez le code suivant :

C#

```
public string ReviewComment
{
```

```
    get => _comment;
    set => _comment = value ?? throw new
ArgumentNullException(nameof(value), "Cannot set to null");
}
string _comment;
```

Le code précédent est la meilleure façon d'exprimer votre conception que `ReviewComment` peut être `null`, mais ne peut pas être défini sur `null`. Une fois que ce code est conscient de nullable, vous pouvez exprimer ce concept plus clairement aux appelants à l'aide de System.Diagnostics.CodeAnalysis.DisallowNullAttribute :

C#

```
[DisallowNull]
public string? ReviewComment
{
    get => _comment;
    set => _comment = value ?? throw new
ArgumentNullException(nameof(value), "Cannot set to null");
}
string? _comment;
```

Dans un contexte nullable, l'accesseur `ReviewComment` `get` peut retourner la valeur par défaut de `null`. Le compilateur avertit qu'il doit être vérifié avant l'accès. En outre, il avertit les appelants que, même s'il peut s'agir de `null`, les appelants ne doivent pas le définir explicitement sur `null`. L'attribut `DisallowNull` spécifie également une *condition préalable*, elle n'affecte pas l'accesseur `get`. Vous utilisez l'attribut `DisallowNull` quand vous observez ces caractéristiques sur :

1. La variable peut être `null` dans des scénarios principaux, souvent lors de sa première instanciation.
2. La variable ne doit pas être explicitement définie sur `null`.

Ces situations sont courantes dans le code qui était à l'origine *inconscient de null*. Il se peut que les propriétés d'objet soient définies dans deux opérations d'initialisation distinctes. Il se peut que certaines propriétés soient définies seulement après la fin d'un travail asynchrone.

Les attributs `AllowNull` et `DisallowNull` vous permettent de spécifier que les conditions préalables sur les variables peuvent ne pas correspondre aux annotations nullables sur ces variables. Celles-ci fournissent plus de détails sur les caractéristiques de votre API. Ces informations supplémentaires aident les appelants à utiliser votre API correctement. N'oubliez pas que vous spécifiez des conditions préalables à l'aide des attributs suivants :

- AllowNull : un argument non-nullable peut être null.
- DisallowNull : un argument nullable ne doit jamais être null.

Post-conditions : `MaybeNull` et `NotNull`

Supposons que vous ayez une méthode avec la signature suivante :

```
C#

public Customer FindCustomer(string lastName, string firstName)
```

Vous avez probablement écrit une méthode comme celle-ci pour retourner `null` quand le nom recherché n'a pas été trouvé. `null` indique clairement que l'enregistrement n'a pas été trouvé. Dans cet exemple, vous changeriez probablement le type de retour en remplaçant `Customer` par `Customer?`. La déclaration de la valeur de retour en tant que type référence nullable spécifie clairement l'intention de cette API :

```
C#

public Customer? FindCustomer(string lastName, string firstName)
```

Pour les raisons décrites sous Nullabilité des génériques, cette technique peut ne pas produire l'analyse statique qui correspond à votre API. Vous pouvez avoir une méthode générique qui suit un modèle similaire :

```
C#

public T Find<T>(IEnumerable<T> sequence, Func<T, bool> predicate)
```

La méthode retourne `null` quand l'élément recherché est introuvable. Vous pouvez préciser que la méthode retourne `null` quand un élément est introuvable en ajoutant l'annotation `MaybeNull` au retour de la méthode :

```
C#

[return: MaybeNull]
public T Find<T>(IEnumerable<T> sequence, Func<T, bool> predicate)
```

Le code précédent informe les appelants que la valeur de retour *peut* en fait être null. Il informe également le compilateur que la méthode peut retourner une expression `null` même si le type est non-nullable. Lorsque vous avez une méthode générique qui

retourne une instance de son paramètre de type, `T`, vous pouvez exprimer qu'elle ne retourne jamais `null` en utilisant l'attribut `NotNull`.

Vous pouvez également spécifier qu'une valeur de retour ou un argument n'est pas null, même si le type est un type référence nullable. La méthode suivante est une méthode d'aide qui lance la procédure si son premier argument est `null` :

C#

```
public static void ThrowWhenNull(object value, string valueExpression = "")
{
    if (value is null) throw new ArgumentNullException(nameof(value),
valueExpression);
}
```

Vous pouvez appeler cette routine comme suit :

C#

```
public static void LogMessage(string? message)
{
    ThrowWhenNull(message, $"{nameof(message)} must not be null");

    Console.WriteLine(message.Length);
}
```

Après avoir activé les types référence null, vous souhaitez vous assurer que le code précédent se compilera sans avertissements. Lorsque la méthode retourne son résultat, il est garanti que le paramètre `value` n'est pas null. Toutefois, il est acceptable d'appeler `ThrowWhenNull` avec une référence null. Vous pouvez faire de `value` un type référence nullable et ajouter la post-condition `NotNull` à la déclaration de paramètre :

C#

```
public static void ThrowWhenNull([NotNull] object? value, string
valueExpression = "")
{
    _ = value ?? throw new ArgumentNullException(nameof(value),
valueExpression);
    // other logic elided
```

Le code précédent exprime clairement le contrat existant : les appelants peuvent transmettre une variable avec la valeur `null`, mais il est garanti que l'argument n'est jamais null si la méthode retourne son résultat sans lever d'exception.

Vous spécifiez des post-conditions inconditionnelles à l'aide des attributs suivants :

- MaybeNull : une valeur de retour non-nullable peut être null.
- NotNull : une valeur de retour nullable ne sera jamais null.

Post-conditions conditionnelles : `NotNullWhen`, `MaybeNullWhen` et `NotNullIfNotNull`

Vous connaissez probablement la méthode `string`String.IsNullOrEmpty(String). Cette méthode retourne `true` quand l'argument est null ou une chaîne vide. Ceci est une forme de contrôle de valeur null : les appelants n'ont pas besoin d'effectuer un contrôle de valeur null de l'argument si la méthode retourne `false`. Pour faire qu'une méthode comme celle-ci prenne en compte la valeur nullable, vous devez définir l'argument sur un type référence nullable et ajouter l'attribut `NotNullWhen` :

C#

```
bool IsNullOrEmpty([NotNullWhen(false)] string? value)
```

Cela informe le compilateur que tout code où la valeur de retour est `false` n'a pas besoin de contrôles de valeur null. L'ajout de l'attribut informe l'analyse statique du compilateur que `IsNullOrEmpty` effectue le contrôle de valeur null nécessaire : quand il retourne `false`, l'argument n'est pas `null`.

C#

```
string? userInput = GetUserInput();
if (!string.IsNullOrEmpty(userInput))
{
    int messageLength = userInput.Length; // no null check needed.
}
// null check needed on userInput here.
```

> ⓘ **Notes**
>
> L'exemple précédent est valide uniquement pour C# 11 et versions ultérieures. À partir de C# 11, l'expression `[nameof]`(../operators/nameof.md) peut référencer des noms de paramètre et de type lorsqu'ils sont utilisés dans un attribut appliqué à une méthode. Dans C# 10 et versions antérieures, vous devez utiliser un littéral de chaîne au lieu de l'expression `nameof`.

La méthode String.IsNullOrEmpty(String) sera annotée comme indiqué ci-dessus pour .NET Core 3.0. Vous pouvez avoir des méthodes similaires dans votre codebase qui vérifient l'état des objets pour les valeurs null. Le compilateur ne reconnaît pas les méthodes de contrôle de valeur null personnalisées et vous devrez ajouter vous-même les annotations. Lorsque vous ajoutez l'attribut, l'analyse statique du compilateur sait quand la variable testée a fait l'objet d'un contrôle de valeur null.

Le modèle `Try*` est une autre utilisation pour ces attributs. Les post-conditions pour les arguments `ref` et `out` sont communiquées par le biais de la valeur de retour. Considérez cette méthode présentée précédemment (dans un contexte ne prenant pas en compte les types nullables) :

C#

```
bool TryGetMessage(string key, out string message)
{
    if (_messageMap.ContainsKey(key))
        message = _messageMap[key];
    else
        message = null;
    return message != null;
}
```

La méthode précédente suit un idiome .NET standard : la valeur de retour indique si `message` a été défini sur la valeur trouvée ou, si aucun message n'est trouvé, sur la valeur par défaut. Si la méthode retourne `true`, la valeur de `message` n'est pas null ; sinon, la méthode définit `message` sur null.

Dans un contexte prenant en compte les types nullables, vous pouvez communiquer cet idiome à l'aide de l'attribut `NotNullWhen`. Lorsque vous annotez des paramètres pour les types référence nullables, faites de `message` un `string?` et ajoutez un attribut :

C#

```
bool TryGetMessage(string key, [NotNullWhen(true)] out string? message)
{
    if (_messageMap.ContainsKey(key))
        message = _messageMap[key];
    else
        message = null;
    return message is not null;
}
```

Dans l'exemple précédent, la valeur de `message` est connue pour n'être pas null quand `TryGetMessage` retourne `true`. Vous devez annoter des méthodes similaires dans votre

codebase de la même façon : les arguments peuvent être égaux à `null`, et sont connus pour ne pas être null quand la méthode retourne `true`.

Vous pouvez également avoir besoin d'un attribut final. Parfois, l'état null d'une valeur de retour dépend de l'état null d'un ou de plusieurs arguments. Ces méthodes retournent une valeur non-null chaque fois que certains arguments ne sont pas `null`. Pour annoter correctement ces méthodes, vous utilisez l'attribut `NotNullIfNotNull`. Considérez la méthode suivante :

```
C#

string GetTopLevelDomainFromFullUrl(string url)
```

Si l'argument `url` n'est pas null, la sortie n'est pas `null`. Une fois les références nullables activées, vous devez ajouter d'autres annotations si votre API peut accepter un argument null. Vous pouvez annoter le type de retour comme indiqué dans le code suivant :

```
C#

string? GetTopLevelDomainFromFullUrl(string? url)
```

Cela fonctionne également, mais force souvent les appelants à implémenter des contrôles de valeur `null` supplémentaires. Le contrat stipule que la valeur de retour est `null` uniquement quand l'argument `url` est `null`. Pour exprimer ce contrat, vous devez annoter cette méthode comme indiqué dans le code suivant :

```
C#

[return: NotNullIfNotNull(nameof(url))]
string? GetTopLevelDomainFromFullUrl(string? url)
```

L'exemple précédent utilise l'opérateur nameof pour le paramètre `url`. Cette fonctionnalité est disponible dans C# 11. Avant C# 11, vous devez taper le nom du paramètre sous forme de chaîne. La valeur de retour et l'argument ont tous les deux été annotés avec le `?` indiquant que l'un ou l'autre peut être `null`. L'attribut clarifie également que la valeur de retour ne sera pas null quand l'argument `url` ne sera pas `null`.

Vous spécifiez des post-conditions conditionnelles à l'aide de ces attributs :

- MaybeNullWhen : un argument non-nullable peut être null quand la méthode retourne la valeur `bool` spécifiée.
- NotNullWhen : un argument nullable ne sera pas null quand la méthode retournera la valeur `bool` spécifiée.
- NotNullIfNotNull : une valeur de retour n'est pas null si l'argument du paramètre spécifié n'est pas null.

Méthodes d'assistance : `MemberNotNull` et `MemberNotNullWhen`

Ces attributs spécifient votre intention lorsque vous avez refactorisé du code commun à partir de constructeurs en méthodes d'assistance. Le compilateur C# analyse les constructeurs et les initialiseurs de champs pour s'assurer que tous les champs de référence non-nullables ont été initialisés avant le retour de chaque constructeur. Toutefois, le compilateur C# ne suit pas les affectations de champs via toutes les méthodes d'assistance. Le compilateur émet un avertissement `CS8618` quand des champs ne sont pas initialisés directement dans le constructeur, mais plutôt dans une méthode d'assistance. Vous ajoutez l'attribut MemberNotNullAttribute à une déclaration de méthode et spécifiez les champs qui sont initialisés sur une valeur non-null dans la méthode. Considérez l'exemple suivant :

C#

```
public class Container
{
    private string _uniqueIdentifier; // must be initialized.
    private string? _optionalMessage;

    public Container()
    {
        Helper();
    }

    public Container(string message)
    {
        Helper();
        _optionalMessage = message;
    }

    [MemberNotNull(nameof(_uniqueIdentifier))]
    private void Helper()
    {
        _uniqueIdentifier = DateTime.Now.Ticks.ToString();
    }
}
```

Vous pouvez spécifier plusieurs noms de champs comme arguments pour le constructeur d'attribut `MemberNotNull`.

L'attribut MemberNotNullWhenAttribute a un argument `bool`. Vous utilisez `MemberNotNullWhen` dans les situations où votre méthode d'assistance retourne un `bool` indiquant si votre méthode d'assistance a initialisé des champs.

Arrêter l'analyse nullable quand la méthode appelée lève une exception

Certaines méthodes, généralement des assistants d'exception ou d'autres méthodes utilitaires, s'arrêtent toujours en levant une exception. Ou bien, une méthode d'assistance peut lever une exception basée sur la valeur d'un argument booléen.

Dans le premier cas, vous pouvez ajouter l'attribut DoesNotReturnAttribute à la déclaration de la méthode. L'analyse d'*état null* du compilateur ne vérifie aucun code dans une méthode qui suit un appel à une méthode annotée avec `DoesNotReturn`. Prenons la méthode suivante :

C#

```
[DoesNotReturn]
private void FailFast()
{
    throw new InvalidOperationException();
}

public void SetState(object containedField)
{
    if (containedField is null)
    {
        FailFast();
    }

    // containedField can't be null:
    _field = containedField;
}
```

Le compilateur n'émet aucun avertissement après l'appel à `FailFast`.

Dans le deuxième cas, vous ajoutez l'attribut System.Diagnostics.CodeAnalysis.DoesNotReturnIfAttribute à un paramètre booléen de la méthode. Vous pouvez modifier l'exemple précédent comme suit :

C#

```
private void FailFastIf([DoesNotReturnIf(true)] bool isNull)
{
    if (isNull)
    {
        throw new InvalidOperationException();
    }
}

public void SetFieldState(object? containedField)
{
    FailFastIf(containedField == null);
    // No warning: containedField can't be null here:
    _field = containedField;
}
```

Lorsque la valeur de l'argument correspond à la valeur du constructeur `DoesNotReturnIf`, le compilateur n'effectue aucune analyse d'*état null* après cette méthode.

Résumé

L'ajout de types référence nullables fournit un vocabulaire initial pour décrire vos attentes d'API pour les variables qui peuvent être `null`. Les attributs fournissent un vocabulaire plus riche pour décrire l'état null des variables en tant que conditions préalables et post-conditions. Ces attributs décrivent plus clairement vos attentes et fournissent une meilleure expérience aux développeurs qui utilisent vos API.

Lorsque vous mettez à jour les bibliothèques pour un contexte nullable, ajoutez ces attributs pour guider les utilisateurs de vos API vers l'utilisation correcte. Ces attributs vous aident à décrire entièrement l'état null des arguments et des valeurs de retour.

- AllowNull : un champ, un paramètre ou une propriété non-nullable peut être null.
- DisallowNull : un champ, un paramètre ou une propriété nullable ne doit jamais être null.
- MaybeNull : un champ, un paramètre, une propriété ou une valeur de retour non-nullable peut être null.
- NotNull : un champ, un paramètre, une propriété ou une valeur de retour nullable ne sera jamais null.
- MaybeNullWhen : un argument non-nullable peut être null quand la méthode retourne la valeur `bool` spécifiée.
- NotNullWhen : un argument nullable ne sera pas null quand la méthode retournera la valeur `bool` spécifiée.
- NotNullIfNotNull : un paramètre, une propriété ou une valeur de retour n'est pas null si l'argument du paramètre spécifié n'est pas null.

- DoesNotReturn : une méthode ou une propriété ne retourne jamais son résultat. En d'autres termes, elle lève toujours une exception.
- DoesNotReturnIf : cette méthode ou propriété ne retourne jamais son résultat si le paramètre `bool` associé a la valeur spécifiée.

Attributs divers interprétés par le compilateur C#

Article • 16/03/2023

Les attributs `Conditional`, `Obsolete`, `AttributeUsage`, `AsyncMethodBuilder`, `InterpolatedStringHandler` et `ModuleInitializer` peuvent être appliqués aux éléments de votre code. Ils ajoutent une signification sémantique à ces éléments. Le compilateur utilise ces significations sémantiques pour modifier sa sortie et signaler les erreurs possibles par les développeurs qui utilisent votre code.

Attribut `Conditional`

Avec l'attribut `Conditional`, l'exécution d'une méthode dépend d'un identificateur de prétraitement. L'attribut `Conditional` est un alias pour ConditionalAttribute, et peut être appliqué à une méthode ou une classe d'attributs.

Dans l'exemple suivant, `Conditional` est appliqué à une méthode pour activer ou désactiver l'affichage d'informations de diagnostic spécifiques au programme :

C#

```
#define TRACE_ON
using System.Diagnostics;

namespace AttributeExamples;

public class Trace
{
    [Conditional("TRACE_ON")]
    public static void Msg(string msg)
    {
        Console.WriteLine(msg);
    }
}

public class TraceExample
{
    public static void Main()
    {
        Trace.Msg("Now in Main...");
        Console.WriteLine("Done.");
    }
}
```

Si l'identificateur `TRACE_ON` n'est pas défini, la sortie de trace n'est pas affichée. Explorez par vous-même dans la fenêtre interactive.

L'attribut `Conditional` est souvent utilisé avec l'identificateur `DEBUG` pour activer des fonctionnalités de traçage et de journalisation pour les builds de débogage, mais pas pour les versions Release, comme le montre l'exemple suivant :

C#

```
[Conditional("DEBUG")]
static void DebugMethod()
{
}
```

Quand une méthode marquée comme conditionnelle est appelée, la présence ou l'absence du symbole de prétraitement spécifié détermine si le compilateur inclut ou omet l'appel à la méthode. Si le symbole est défini, l'appel est inclus ; sinon, il est omis. Une méthode conditionnelle doit figurer dans une déclaration de classe ou de struct et doit avoir une type de retour `void`. `Conditional` offre une solution à la fois plus efficace, plus élégante et moins sujette aux erreurs que les méthodes englobantes contenues dans les blocs `#if…#endif`.

Si une méthode a plusieurs attributs `Conditional`, le compilateur inclut des appels à la méthode si un ou plusieurs symboles conditionnels sont définis (les symboles sont liés logiquement à l'aide de l'opérateur OR). Dans l'exemple suivant, la présence de résultats `A` ou `B` entraîne un appel de méthode :

C#

```
[Conditional("A"), Conditional("B")]
static void DoIfAorB()
{
    // ...
}
```

Utilisation de `Conditional` avec des classes d'attributs

L'attribut `Conditional` peut également être appliqué à une définition de classe d'attributs. Dans l'exemple suivant, l'attribut personnalisé `Documentation` ajoute des informations uniquement aux métadonnées si `DEBUG` est défini.

C#

```
[Conditional("DEBUG")]
public class DocumentationAttribute : System.Attribute
{
    string text;

    public DocumentationAttribute(string text)
    {
        this.text = text;
    }
}

class SampleClass
{
    // This attribute will only be included if DEBUG is defined.
    [Documentation("This method displays an integer.")]
    static void DoWork(int i)
    {
        System.Console.WriteLine(i.ToString());
    }
}
```

Attribut `Obsolete`

L'attribut `Obsolete` marque un élément de code comme n'est plus recommandé pour une utilisation. L'utilisation d'une entité marquée comme obsolète génère un avertissement ou une erreur. `Obsolete` est un attribut à usage unique et peut être appliqué à toute entité qui autorise des attributs. `Obsolete` est un alias pour ObsoleteAttribute.

Dans l'exemple suivant, l'attribut `Obsolete` est appliqué à la classe `A` et à la méthode `B.OldMethod`. Comme le deuxième argument du constructeur d'attribut appliqué à `B.OldMethod` a la valeur `true`, l'utilisation de cette méthode entraîne une erreur du compilateur, alors que l'utilisation de la classe `A` n'entraîne qu'un avertissement. Toutefois, l'appel de `B.NewMethod` ne produit aucun avertissement ni aucune erreur. Par exemple, utilisé avec les définitions précédentes, le code suivant génère deux avertissements et une erreur :

C#

```
namespace AttributeExamples
{
    [Obsolete("use class B")]
    public class A
    {
        public void Method() { }
```

```
    }

    public class B
    {
        [Obsolete("use NewMethod", true)]
        public void OldMethod() { }

        public void NewMethod() { }
    }

    public static class ObsoleteProgram
    {
        public static void Main()
        {
            // Generates 2 warnings:
            A a = new A();

            // Generate no errors or warnings:
            B b = new B();
            b.NewMethod();

            // Generates an error, compilation fails.
            // b.OldMethod();
        }
    }
}
```

La chaîne fournie comme premier argument au constructeur d'attribut est affichée dans l'avertissement ou dans l'erreur. Deux avertissements pour la classe `A` sont générés : un pour la déclaration de la référence de classe et un pour le constructeur de classe. L'attribut `Obsolete` peut être utilisé sans arguments, mais il est recommandé d'inclure une explication indiquant quoi utiliser en remplacement.

En C# 10, vous pouvez utiliser l'interpolation de chaîne constante et l'opérateur `nameof` pour vous assurer que les noms correspondent :

C#

```
public class B
{
    [Obsolete($"use {nameof(NewMethod)} instead", true)]
    public void OldMethod() { }

    public void NewMethod() { }
}
```

Attribut `SetsRequiredMembers`

L'attribut `SetsRequiredMembers` informe le compilateur qu'un constructeur définit tous les membres `required` de cette classe ou de ce struct. Le compilateur suppose que tout constructeur avec l'attribut System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute initialise tous les membres `required`. Tout code qui appelle un tel constructeur n'a pas besoin d'initialiseurs d'objet pour définir les membres requis. Cela est principalement utile pour les enregistrements positionnels et les constructeurs principaux.

Attribut `AttributeUsage`

L'attribut `AttributeUsage` détermine de quelle manière une classe d'attributs personnalisés peut être utilisée. AttributeUsageAttribute est un attribut que vous appliquez à des définitions d'attribut personnalisé. L'attribut `AttributeUsage` vous permet de contrôler :

- À quels éléments de programme les attributs peuvent être appliqués. Sauf si vous en limitez l'utilisation, un attribut peut être appliqué aux éléments de programme suivants :
  - Assembly
  - Module
  - Champ
  - Événement
  - Méthode
  - Paramètre
  - Propriété
  - Renvoie
  - Type
- Si un attribut peut être appliqué plusieurs fois à un même élément de programme.
- Si les attributs sont hérités dans les classes dérivées.

L'exemple suivant montre des paramètres par défaut quand ils sont appliqués explicitement :

C#

```
[AttributeUsage(AttributeTargets.All,
                  AllowMultiple = false,
                  Inherited = true)]
class NewAttribute : Attribute { }
```

Dans cet exemple, la classe `NewAttribute` peut être appliquée à n'importe quel élément de programme pris en charge. Elle ne peut cependant être appliquée qu'une seule fois à

chaque entité. L'attribut est hérité par les classes dérivées quand il est appliqué à une classe de base.

Les arguments AllowMultiple et Inherited étant facultatifs, le code suivant a le même effet :

C#

```
[AttributeUsage(AttributeTargets.All)]
class NewAttribute : Attribute { }
```

Le premier argument AttributeUsageAttribute doit correspondre à un ou plusieurs éléments de l'énumération AttributeTargets. Vous pouvez lier ensemble plusieurs types de cibles avec l'opérateur OR, comme dans l'exemple suivant :

C#

```
[AttributeUsage(AttributeTargets.Property | AttributeTargets.Field)]
class NewPropertyOrFieldAttribute : Attribute { }
```

Les attributs peuvent être appliqués à la propriété ou au champ de stockage pour une propriété implémentée automatiquement. L'attribut s'applique à la propriété, sauf si vous spécifiez le spécificateur `field` sur l'attribut. Les deux cas sont illustrés dans l'exemple suivant :

C#

```
class MyClass
{
    // Attribute attached to property:
    [NewPropertyOrField]
    public string Name { get; set; } = string.Empty;

    // Attribute attached to backing field:
    [field: NewPropertyOrField]
    public string Description { get; set; } = string.Empty;
}
```

Si l'argument AllowMultiple est défini sur `true`, l'attribut résultant peut être appliqué plusieurs fois à une même entité, comme illustré dans l'exemple suivant :

C#

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
class MultiUse : Attribute { }
```

```
[MultiUse]
[MultiUse]
class Class1 { }

[MultiUse, MultiUse]
class Class2 { }
```

Dans ce cas, `MultiUseAttribute` peut être appliqué à plusieurs reprises, car `AllowMultiple` est défini sur `true`. Les deux formats indiqués pour appliquer plusieurs attributs sont valides.

Si Inherited est `false`, l'attribut n'est pas hérité par les classes dérivées d'une classe avec attributs. Par exemple :

C#

```
[AttributeUsage(AttributeTargets.Class, Inherited = false)]
class NonInheritedAttribute : Attribute { }

[NonInherited]
class BClass { }

class DClass : BClass { }
```

Dans ce cas, `NonInheritedAttribute` n'est pas appliqué à `DClass` via l'héritage.

Vous pouvez également utiliser ces mots clés pour spécifier où un attribut doit être appliqué. Par exemple, vous pouvez utiliser le spécificateur `field:` pour ajouter un attribut au champ de stockage d'une propriété implémentée automatiquement. Vous pouvez également utiliser le spécificateur `field:`, `property:` ou `param:` pour appliquer un attribut à l'un des éléments générés à partir d'un enregistrement positionnel. Pour obtenir un exemple, consultez Syntaxe positionnelle pour la définition de propriété.

Attribut `AsyncMethodBuilder`

Vous ajoutez l'attribut System.Runtime.CompilerServices.AsyncMethodBuilderAttribute à un type qui peut être un type de retour asynchrone. L'attribut spécifie le type qui génère l'implémentation de la méthode asynchrone lorsque le type spécifié est retourné à partir d'une méthode asynchrone. L'attribut `AsyncMethodBuilder` peut être appliqué à un type qui :

- A une méthode accessible `GetAwaiter`.
- L'objet retourné par la méthode `GetAwaiter` implémente l'interface System.Runtime.CompilerServices.ICriticalNotifyCompletion.

Le constructeur de l'attribut `AsyncMethodBuilder` spécifie le type du générateur associé. Le générateur doit implémenter les membres accessibles suivants :

- Méthode statique `Create()` qui retourne le type du générateur.
- Propriété lisible `Task` qui renvoie le type de retour asynchrone.
- Méthode `void SetException(Exception)` qui définit l'exception en cas d'erreur d'une tâche.
- Méthode `void SetResult()` ou `void SetResult(T result)` qui marque la tâche comme terminée et définit éventuellement le résultat de la tâche
- Méthode `Start` avec la signature d'API suivante :

```
C#

void Start<TStateMachine>(ref TStateMachine stateMachine)
          where TStateMachine :
System.Runtime.CompilerServices.IAsyncStateMachine
```

- Méthode `AwaitOnCompleted` avec la signature suivante :

```
C#

public void AwaitOnCompleted<TAwaiter, TStateMachine>(ref TAwaiter
awaiter, ref TStateMachine stateMachine)
    where TAwaiter : System.Runtime.CompilerServices.INotifyCompletion
    where TStateMachine :
System.Runtime.CompilerServices.IAsyncStateMachine
```

- Méthode `AwaitUnsafeOnCompleted` avec la signature suivante :

```
C#

      public void AwaitUnsafeOnCompleted<TAwaiter, TStateMachine>(ref
TAwaiter awaiter, ref TStateMachine stateMachine)
          where TAwaiter :
System.Runtime.CompilerServices.ICriticalNotifyCompletion
          where TStateMachine :
System.Runtime.CompilerServices.IAsyncStateMachine
```

Vous pouvez en savoir plus sur les générateurs de méthodes asynchrones en lisant les générateurs suivants fournis par .NET :

- System.Runtime.CompilerServices.AsyncTaskMethodBuilder

- System.Runtime.CompilerServices.AsyncTaskMethodBuilder<TResult>
- System.Runtime.CompilerServices.AsyncValueTaskMethodBuilder
- System.Runtime.CompilerServices.AsyncValueTaskMethodBuilder<TResult>

Dans C# 10 et versions ultérieures, l'attribut `AsyncMethodBuilder` peut être appliqué à une méthode asynchrone pour remplacer le générateur pour ce type.

Attributs `InterpolatedStringHandler` et `InterpolatedStringHandlerArguments`

À compter de C# 10, utilisez ces attributs pour spécifier qu'un type est un *gestionnaire de chaîne interpolé*. La bibliothèque .NET 6 inclut déjà System.Runtime.CompilerServices.DefaultInterpolatedStringHandler pour les scénarios où vous utilisez une chaîne interpolée comme argument pour un paramètre `string`. Vous pouvez avoir d'autres instances pour lesquelles vous souhaitez contrôler la façon dont les chaînes interpolées sont traitées. Vous appliquez System.Runtime.CompilerServices.InterpolatedStringHandlerAttribute au type qui implémente votre gestionnaire. Vous appliquez System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute aux paramètres du constructeur de ce type.

Vous pouvez en savoir plus sur la création d'un gestionnaire de chaînes interpolées dans la spécification de fonctionnalité C# 10 pour les améliorations de chaîne interpolée.

Attribut `ModuleInitializer`

À compter de C# 9, l'attribut `ModuleInitializer` marque une méthode que le runtime appelle lorsque l'assembly se charge. `ModuleInitializer` est un alias pour ModuleInitializerAttribute.

L'attribut `ModuleInitializer` ne peut être appliqué qu'à une méthode qui :

- Est statique.
- Est sans paramètre.
- Retourne `void`.
- Est accessible à partir du module conteneur, c'est à dire, `internal` ou `public`.
- N'est pas une méthode générique.
- N'est pas contenue dans une classe générique.
- N'est pas une fonction locale.

L'attribut `ModuleInitializer` peut être appliqué à plusieurs méthodes. Dans ce cas, l'ordre dans lequel le runtime les appelle est déterministe, mais pas spécifié.

L'exemple suivant illustre l'utilisation de plusieurs méthodes d'initialiseur de module. Les méthodes `Init1` et `Init2` s'exécutent avant `Main` et chacune ajoute une chaîne à la propriété `Text`. Par conséquent, lors de l'exécution de `Main`, la propriété `Text` a déjà des chaînes provenant des deux méthodes d'initialiseur.

C#

```
using System;

internal class ModuleInitializerExampleMain
{
    public static void Main()
    {
        Console.WriteLine(ModuleInitializerExampleModule.Text);
        //output: Hello from Init1! Hello from Init2!
    }
}
```

C#

```
using System.Runtime.CompilerServices;

internal class ModuleInitializerExampleModule
{
    public static string? Text { get; set; }

    [ModuleInitializer]
    public static void Init1()
    {
        Text += "Hello from Init1! ";
    }

    [ModuleInitializer]
    public static void Init2()
    {
        Text += "Hello from Init2! ";
    }
}
```

Les générateurs de code source doivent parfois générer du code d'initialisation. Les initialiseurs de module fournissent un emplacement standard pour ce code. Dans la plupart des autres cas, vous devez écrire un constructeur statique au lieu d'un initialiseur de module.

Attribut `SkipLocalsInit`

À compter de C# 9, l'attribut `SkipLocalsInit` empêche le compilateur de définir l'indicateur `.locals init` lors de l'émission en métadonnées. L'attribut `SkipLocalsInit` est un attribut à usage unique qui peut être appliqué à une méthode, une propriété, une classe, un struct, une interface ou un module, mais pas à un assembly. `SkipLocalsInit` est un alias pour SkipLocalsInitAttribute.

L'indicateur `.locals init` fait que le CLR initialise toutes les variables locales déclarées dans une méthode sur leurs valeurs par défaut. Étant donné que le compilateur s'assure également que vous n'utilisez jamais une variable avant de lui attribuer une valeur, `.locals init` n'est généralement pas nécessaire. Toutefois, l'initialisation zéro supplémentaire peut avoir un impact mesurable sur les performances dans certains scénarios, par exemple, lorsque vous utilisez stackalloc pour allouer un tableau sur la pile. Dans ces cas, vous pouvez ajouter l'attribut `SkipLocalsInit`. S'il est appliqué directement à une méthode, l'attribut affecte cette méthode et toutes ses fonctions imbriquées, y compris les fonctions lambda et locales. S'il est appliqué à un type ou à un module, il affecte toutes les méthodes imbriquées à l'intérieur. Cet attribut n'affecte pas les méthodes abstraites, mais il affecte le code généré pour l'implémentation.

Cet attribut nécessite l'option de compilateur AllowUnsafeBlocks. Cette exigence signale que dans certains cas, le code peut afficher la mémoire non affectée (par exemple, la lecture à partir de la mémoire allouée à la pile non initialisée).

L'exemple suivant illustre l'effet de l'attribut `SkipLocalsInit` sur une méthode qui utilise `stackalloc`. La méthode affiche ce qui était en mémoire lorsque le tableau d'entiers a été alloué.

C#

```
[SkipLocalsInit]
static void ReadUninitializedMemory()
{
    Span<int> numbers = stackalloc int[120];
    for (int i = 0; i < 120; i++)
    {
        Console.WriteLine(numbers[i]);
    }
}
// output depends on initial contents of memory, for example:
//0
//0
//0
//168
//0
```

```
//-1271631451
//32767
//38
//0
//0
//0
//38
// Remaining rows omitted for brevity.
```

Pour essayer ce code vous-même, définissez l'option du compilateur `AllowUnsafeBlocks` dans votre fichier *.csproj* :

XML

```
<PropertyGroup>
  ...
  <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
</PropertyGroup>
```

Voir aussi

- Attribute
- System.Reflection
- Attributs
- Réflexion

Code, types de pointeurs et pointeurs de fonction non sécurisés

Article • 10/05/2023

La plupart du code C# que vous écrivez est un « code sûr vérifiable ». *Le code vérifiable sécurisé* signifie que les outils .NET peuvent vérifier que le code est sécurisé. En général, le code sécurisé n'accède pas directement à la mémoire à l'aide de pointeurs. Il n'alloue pas non plus de mémoire brute. Il crée des objets managés à la place.

C# prend en charge un contexte unsafe dans lequel vous pouvez écrire du code *non vérifiable* . Dans un contexte `unsafe`, le code peut utiliser des pointeurs, allouer et libérer des blocs de mémoire, et appeler des méthodes à l'aide de pointeurs de fonction. Le code non sécurisé en C# n'est pas nécessairement dangereux : il s'agit simplement de code dont la sécurité ne peut pas être vérifiée.

Le code non sécurisé a les propriétés suivantes :

- Les méthodes, les types et les blocs de code peuvent être définis comme non sécurisés.
- Dans certains cas, le code non sécurisé peut augmenter les performances d'une application en supprimant les vérifications des limites des tableaux.
- Du code non sécurisé est obligatoire quand vous appelez des fonctions natives nécessitant des pointeurs.
- L'utilisation de code non sécurisé introduit des risques pour la sécurité et la stabilité.
- Le code qui contient des blocs unsafe doit être compilé avec l'option de compilateur **AllowUnsafeBlocks**.

Types de pointeur

Dans un contexte non sécurisé, un type peut être un type pointeur, en plus d'un type valeur ou d'un type référence. La déclaration d'un type pointeur peut prendre l'une des formes suivantes :

C#

```
type* identifier;
void* identifier; //allowed but not recommended
```

Le type spécifié avant `*` dans un type de pointeur est appelé **type référent**. Seul un type non managé peut être un type référent.

Les types pointeur n'héritent pas de object, et aucune conversion n'est possible entre les types pointeur et `object`. Par ailleurs, le boxing et l'unboxing ne prennent pas en charge les pointeurs. Cependant, vous pouvez effectuer des conversions entre différents types pointeur ainsi qu'entre des types pointeur et des types intégraux.

Lorsque vous déclarez plusieurs pointeurs dans la même déclaration, vous écrivez l'astérisque (`*`) avec le type sous-jacent uniquement. Il n'est pas utilisé comme préfixe de chaque nom de pointeur. Par exemple :

```
C#

int* p1, p2, p3;   // Ok
int *p1, *p2, *p3;   // Invalid in C#
```

Un pointeur ne peut pas pointer vers une référence ou vers un struct qui contient des références, car une référence d'objet peut être collectée par le récupérateur de mémoire, même si un pointeur pointe vers elle. Le récupérateur de mémoire ne se préoccupe pas de savoir si un objet est pointé par des types pointeur.

La valeur de la variable pointeur de type `MyType*` est l'adresse d'une variable de type `MyType`. Les éléments suivants sont des exemples de déclarations de type pointeur :

- `int* p` : `p` est un pointeur vers un entier.
- `int** p` : `p` est un pointeur vers un pointeur vers un entier.
- `int*[] p` : `p` est un tableau unidimensionnel de pointeurs vers des entiers.
- `char* p` : `p` est un pointeur vers un caractère.
- `void* p` : `p` est un pointeur vers un type inconnu.

L'opérateur d'indirection de pointeur `*` peut être utilisé pour accéder au contenu à l'emplacement vers lequel pointe la variable pointeur. Observez par exemple la déclaration suivante :

```
C#

int* myVariable;
```

L'expression `*myVariable` désigne la variable `int` trouvée à l'adresse contenue dans `myVariable`.

Il existe plusieurs exemples de pointeurs dans les articles sur l'instructionfixed. L'exemple suivant utilise le mot clé `unsafe` et les instructions `fixed`, et montre comment incrémenter un pointeur intérieur. Vous pouvez coller ce code dans la fonction Main d'une application console pour l'exécuter. Ces exemples doivent être compilés avec l'ensemble d'options de compilateur **AllowUnsafeBlocks**.

C#

```
// Normal pointer to an object.
int[] a = new int[5] { 10, 20, 30, 40, 50 };
// Must be in unsafe code to use interior pointers.
unsafe
{
    // Must pin object on heap so that it doesn't move while using interior
pointers.
    fixed (int* p = &a[0])
    {
        // p is pinned as well as object, so create another pointer to show
incrementing it.
        int* p2 = p;
        Console.WriteLine(*p2);
        // Incrementing p2 bumps the pointer by four bytes due to its type
...
        p2 += 1;
        Console.WriteLine(*p2);
        p2 += 1;
        Console.WriteLine(*p2);
        Console.WriteLine("--------");
        Console.WriteLine(*p);
        // Dereferencing p and incrementing changes the value of a[0] ...
        *p += 1;
        Console.WriteLine(*p);
        *p += 1;
        Console.WriteLine(*p);
    }
}

Console.WriteLine("--------");
Console.WriteLine(a[0]);

/*
Output:
10
20
30
--------
10
11
12
--------
12
*/
```

L'opérateur d'indirection ne peut pas être appliqué à un pointeur de type `void*`. Toutefois, vous pouvez utiliser un cast pour convertir un pointeur void en n'importe quel autre type pointeur, et inversement.

Un pointeur peut être `null`. Le fait d'appliquer un opérateur d'indirection à un pointeur Null donne lieu à un comportement défini par l'implémentation.

Le passage de pointeurs entre méthodes peut engendrer un comportement non défini. Supposons une méthode qui retourne un pointeur à une variable locale par le biais d'un paramètre `in`, `out` ou `ref`, ou comme résultat de fonction. Si le pointeur a été défini dans un bloc fixed, la variable vers laquelle il pointe peut ne plus être fixed.

Le tableau suivant répertorie les opérateurs et les instructions qui peuvent fonctionner sur des pointeurs dans un contexte unsafe :

Opérateur/Instruction	Utilisation
`*`	Exécute l'indirection de pointeur.
`->`	Accède à un membre d'un struct via un pointeur.
`[]`	Indexe un pointeur.
`&`	Obtient l'adresse d'une variable.
`++` et `--`	Incrémente et décrémente les pointeurs.
`+` et `-`	Exécute des opérations arithmétiques sur les pointeurs.
`==`, `!=`, `<`, `>`, `<=` et `>=`	Compare des pointeurs.
stackalloc	Alloue de la mémoire sur la pile.
instruction fixed	Résout temporairement une variable afin de pouvoir rechercher son adresse.

Pour plus d'informations sur les opérateurs associés au pointeur, consultez Opérateurs associés au pointeur.

N'importe quel type de pointeur peut être converti implicitement en type `void*`. La valeur `null` peut être attribuée à n'importe quel type de pointeur. N'importe quel type de pointeur peut être explicitement converti en n'importe quel autre type de pointeur à l'aide d'une expression cast. Vous pouvez également convertir n'importe quel type intégral en type de pointeur ou n'importe quel type de pointeur en type intégral. Ces conversions nécessitent un cast explicite.

L'exemple suivant convertit un `int*` en `byte*`. Notez que le pointeur pointe vers l'octet traité le plus faible de la variable. Quand vous incrémentez successivement le résultat, jusqu'à la taille de `int` (4 octets), vous pouvez afficher les octets restants de la variable.

C#

```
int number = 1024;

unsafe
{
    // Convert to byte:
    byte* p = (byte*)&number;

    System.Console.Write("The 4 bytes of the integer:");

    // Display the 4 bytes of the int variable:
    for (int i = 0 ; i < sizeof(int) ; ++i)
    {
        System.Console.Write(" {0:X2}", *p);
        // Increment the pointer:
        p++;
    }
    System.Console.WriteLine();
    System.Console.WriteLine("The value of the integer: {0}", number);

    /* Output:
        The 4 bytes of the integer: 00 04 00 00
        The value of the integer: 1024
    */
}
```

Mémoires tampons de taille fixe

Vous pouvez utiliser le mot clé `fixed` pour créer une mémoire tampon avec un tableau de taille fixe dans une structure de données. Les mémoires tampons de taille fixe sont utiles quand vous écrivez des méthodes qui sont interopérables avec les sources de données d'autres langages ou plateformes. La mémoire tampon fixe peut accepter tous les attributs ou modificateurs qui sont autorisés pour les membres de structures régulières. La seule restriction est que le tableau doit être de type `bool`, `byte`, `char`, `short`, `int`, `long`, `sbyte`, `ushort`, `uint`, `ulong`, `float` ou `double`.

C#

```
private fixed char name[30];
```

Dans du code sûr, un struct C# qui contient un tableau ne contient pas les éléments du tableau. Le struct contient une référence aux éléments du tableau à la place. Vous pouvez incorporer un tableau de taille fixe dans un struct quand il est utilisé dans un bloc de code unsafe.

La taille des éléments suivants `struct` ne dépend pas du nombre d'éléments dans le tableau, car `pathName` il s'agit d'une référence :

C#

```
public struct PathArray
{
    public char[] pathName;
    private int reserved;
}
```

Un struct peut contenir un tableau incorporé dans du code unsafe. Dans l'exemple suivant, le tableau `fixedBuffer` a une taille fixe. Vous utilisez une fixed instruction pour obtenir un pointeur vers le premier élément. Accédez aux éléments du tableau par le biais de ce pointeur. L'instruction `fixed` épingle le champ de l'instance `fixedBuffer` à un emplacement spécifique de la mémoire.

C#

```
internal unsafe struct Buffer
{
    public fixed char fixedBuffer[128];
}

internal unsafe class Example
{
    public Buffer buffer = default;
}

private static void AccessEmbeddedArray()
{
    var example = new Example();

    unsafe
    {
        // Pin the buffer to a fixed location in memory.
        fixed (char* charPtr = example.buffer.fixedBuffer)
        {
            *charPtr = 'A';
        }
        // Access safely through the index:
        char c = example.buffer.fixedBuffer[0];
        Console.WriteLine(c);
```

```
        // Modify through the index:
        example.buffer.fixedBuffer[0] = 'B';
        Console.WriteLine(example.buffer.fixedBuffer[0]);
    }
}
```

La taille du tableau `char` de 128 éléments est de 256 octets. Les mémoires tampons char de taille fixe acceptent toujours 2 octets par caractère, quel que soit l'encodage. Cette taille de tableau est la même lorsque les mémoires tampons char sont marshalées vers des méthodes ou des structs d'API avec `CharSet = CharSet.Auto` ou `CharSet = CharSet.Ansi`. Pour plus d'informations, consultez CharSet.

L'exemple précédent illustre l'accès aux champs `fixed` sans épinglage. Un autre tableau courant de taille fixe est le tableau bool. Les éléments d'un tableau `bool` sont toujours d'un octet. Les tableaux `bool` ne conviennent pas à la création de tableaux d'octets ou de mémoires tampons.

Les mémoires tampons de taille fixe sont compilées avec System.Runtime.CompilerServices.UnsafeValueTypeAttribute qui indique au CLR qu'un type contient un tableau non managé susceptible de dépasser sa capacité. La mémoire allouée à l'aide de stackalloc active également automatiquement les fonctionnalités de détection de dépassement de mémoire tampon dans le CLR. L'exemple précédent montre comment une mémoire tampon de taille fixe peut exister dans un `unsafe struct`.

C#

```
internal unsafe struct Buffer
{
    public fixed char fixedBuffer[128];
}
```

Le C# généré par le compilateur pour `Buffer` est attribué comme suit :

C#

```
internal struct Buffer
{
    [StructLayout(LayoutKind.Sequential, Size = 256)]
    [CompilerGenerated]
    [UnsafeValueType]
    public struct <fixedBuffer>e__FixedBuffer
    {
        public char FixedElementField;
    }
```

```
    [FixedBuffer(typeof(char), 128)]
    public <fixedBuffer>e__FixedBuffer fixedBuffer;
}
```

Les mémoires tampons fixes diffèrent des tableaux normaux des manières suivantes :

- Ne peuvent être utilisées que dans un contexte `unsafe`.
- Il ne peut s'agir que de champs d'instance de structs.
- Ce sont toujours des vecteurs ou des tableaux unidimensionnels.
- La déclaration doit inclure la longueur, par exemple `fixed char id[8]`. Vous ne pouvez pas utiliser `fixed char id[]`.

Comment utiliser des pointeurs pour copier un tableau d'octets

L'exemple suivant utilise des pointeurs pour copier des octets d'un tableau à un autre.

Cet exemple utilise le mot clé unsafe, qui vous permet d'utiliser des pointeurs dans la méthode `Copy`. L'instruction fixed permet de déclarer des pointeurs vers les tableaux source et de destination. L'instruction `fixed` *épingle* l'emplacement des tableaux source et de destination en mémoire pour que ceux-ci ne soient pas déplacés par l'opération de garbage collection. Les blocs de mémoire des tableaux sont libérés quand le bloc `fixed` est effectué. Comme la méthode `Copy` dans cet exemple utilise le mot clé `unsafe`, elle doit être compilée avec l'option de compilateur **AllowUnsafeBlocks**.

Cet exemple accède aux éléments des deux tableaux au moyen d'index et non au moyen d'un second pointeur non managé. La déclaration des pointeurs `pSource` et `pTarget` épingle les tableaux.

C#

```
static unsafe void Copy(byte[] source, int sourceOffset, byte[] target,
    int targetOffset, int count)
{
    // If either array is not instantiated, you cannot complete the copy.
    if ((source == null) || (target == null))
    {
        throw new System.ArgumentException("source or target is null");
    }

    // If either offset, or the number of bytes to copy, is negative, you
    // cannot complete the copy.
    if ((sourceOffset < 0) || (targetOffset < 0) || (count < 0))
    {
        throw new System.ArgumentException("offset or bytes to copy is
```

```
negative");
    }

    // If the number of bytes from the offset to the end of the array is
    // less than the number of bytes you want to copy, you cannot complete
    // the copy.
    if ((source.Length - sourceOffset < count) ||
        (target.Length - targetOffset < count))
    {
        throw new System.ArgumentException("offset to end of array is less
than bytes to be copied");
    }

    // The following fixed statement pins the location of the source and
    // target objects in memory so that they will not be moved by garbage
    // collection.
    fixed (byte* pSource = source, pTarget = target)
    {
        // Copy the specified number of bytes from source to target.
        for (int i = 0; i < count; i++)
        {
            pTarget[targetOffset + i] = pSource[sourceOffset + i];
        }
    }
}

static void UnsafeCopyArrays()
{
    // Create two arrays of the same length.
    int length = 100;
    byte[] byteArray1 = new byte[length];
    byte[] byteArray2 = new byte[length];

    // Fill byteArray1 with 0 - 99.
    for (int i = 0; i < length; ++i)
    {
        byteArray1[i] = (byte)i;
    }

    // Display the first 10 elements in byteArray1.
    System.Console.WriteLine("The first 10 elements of the original are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray1[i] + " ");
    }
    System.Console.WriteLine("\n");

    // Copy the contents of byteArray1 to byteArray2.
    Copy(byteArray1, 0, byteArray2, 0, length);

    // Display the first 10 elements in the copy, byteArray2.
    System.Console.WriteLine("The first 10 elements of the copy are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray2[i] + " ");
```

```
    }
    System.Console.WriteLine("\n");

    // Copy the contents of the last 10 elements of byteArray1 to the
    // beginning of byteArray2.
    // The offset specifies where the copying begins in the source array.
    int offset = length - 10;
    Copy(byteArray1, offset, byteArray2, 0, length - offset);

    // Display the first 10 elements in the copy, byteArray2.
    System.Console.WriteLine("The first 10 elements of the copy are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray2[i] + " ");
    }
    System.Console.WriteLine("\n");
    /* Output:
        The first 10 elements of the original are:
        0 1 2 3 4 5 6 7 8 9

        The first 10 elements of the copy are:
        0 1 2 3 4 5 6 7 8 9

        The first 10 elements of the copy are:
        90 91 92 93 94 95 96 97 98 99
    */
}
```

Pointeurs fonction

C# fournit des types delegate pour définir des objets pointeurs de fonction sécurisés. L'appel d'un délégué implique l'instanciation d'un type dérivé de System.Delegate et l'appel d'une méthode virtuelle à sa méthode `Invoke`. Cet appel virtuel utilise l'instruction IL `callvirt`. Dans les chemins du code critiques pour les performances, l'utilisation de l'instruction IL `calli` est plus efficace.

Vous pouvez définir un pointeur de fonction à l'aide de la syntaxe `delegate*`. Le compilateur appelle la fonction à l'aide de l'instruction `calli` plutôt que d'instancier un objet `delegate` et d'appeler `Invoke`. Le code suivant déclare deux méthodes qui utilisent un `delegate` ou un `delegate*` pour combiner deux objets du même type. La première méthode utilise un type délégué System.Func<T1,T2,TResult>. La deuxième méthode utilise une déclaration `delegate*` avec les mêmes paramètres et le même type de retour :

```
C#
```

```
public static T Combine<T>(Func<T, T, T> combinator, T left, T right) =>
    combinator(left, right);

public static T UnsafeCombine<T>(delegate*<T, T, T> combinator, T left, T
right) =>
    combinator(left, right);
```

Le code suivant montre comment déclarer une fonction locale statique et appeler la méthode à l'aide d'un pointeur `UnsafeCombine` vers cette fonction locale :

C#

```
static int localMultiply(int x, int y) => x * y;
int product = UnsafeCombine(&localMultiply, 3, 4);
```

Le code précédent illustre plusieurs des règles sur la fonction accessible en tant que pointeur de fonction :

- Les pointeurs de fonction ne peuvent être déclarés que dans un contexte `unsafe`.
- Les méthodes qui prennent un `delegate*` (ou retournent un `delegate*`) ne peuvent être appelées que dans un contexte `unsafe`.
- L'opérateur `&` permettant d'obtenir l'adresse d'une fonction est autorisé uniquement sur les fonctions `static`. (Cette règle s'applique aux fonctions membres et aux fonctions locales).

La syntaxe comporte des parallèles avec la déclaration de types `delegate` et l'utilisation de pointeurs. Le suffixe `*` sur `delegate` indique que la déclaration est un *pointeur de fonction*. L'affectation d'un groupe de méthodes à un pointeur de fonction indique que l'opération `&` prend l'adresse de la méthode.

Vous pouvez spécifier la convention d'appel pour un `delegate*` à l'aide des mots clés `managed` et `unmanaged`. En outre, pour les pointeurs de fonction `unmanaged`, vous pouvez spécifier la convention d'appel. Les déclarations suivantes présentent des exemples de chacune d'elles. La première déclaration utilise la convention d'appel `managed`, qui est la valeur par défaut. Les quatre suivantes utilisent une convention d'appel `unmanaged`. Chacune spécifie l'une des conventions d'appel ECMA 335 : `Cdecl`, `Stdcall`, `Fastcall` ou `Thiscall`. La dernière déclaration utilise la convention d'appel `unmanaged`, ce qui indique au CLR de choisir la convention d'appel par défaut pour la plateforme. Le CLR choisit la convention d'appel au moment de l'exécution.

C#

```
public static T ManagedCombine<T>(delegate* managed<T, T, T> combinator, T
left, T right) =>
    combinator(left, right);
public static T CDeclCombine<T>(delegate* unmanaged[Cdecl]<T, T, T>
combinator, T left, T right) =>
    combinator(left, right);
public static T StdcallCombine<T>(delegate* unmanaged[Stdcall]<T, T, T>
combinator, T left, T right) =>
    combinator(left, right);
public static T FastcallCombine<T>(delegate* unmanaged[Fastcall]<T, T, T>
combinator, T left, T right) =>
    combinator(left, right);
public static T ThiscallCombine<T>(delegate* unmanaged[Thiscall]<T, T, T>
combinator, T left, T right) =>
    combinator(left, right);
public static T UnmanagedCombine<T>(delegate* unmanaged<T, T, T> combinator,
T left, T right) =>
    combinator(left, right);
```

Pour en savoir plus sur les pointeurs de fonction, consultez la proposition de pointeur de fonction pour C# 9.0.

spécification du langage C#

Pour en savoir plus, consultez le chapitre Code unsafe de la spécification du langage C#.

Directives de préprocesseur C#

Article • 10/05/2023

Bien que le compilateur n'ait pas de préprocesseur distinct, les directives décrites dans cette section sont traitées comme si c'était le cas. Vous les utilisez pour faciliter la compilation conditionnelle. Contrairement aux directives C et C++, vous ne pouvez pas utiliser ces directives pour créer des macros. Une directive de préprocesseur doit être la seule instruction sur une ligne.

Contexte pouvant accepter la valeur Null

La directive de préprocesseur `#nullable` définit le *contexte d'annotation pouvant accepter la valeur Null* et le *contexte d'avertissement pouvant accepter la valeur Null*. Cette directive contrôle si les annotations pouvant accepter la valeur Null ont un effet et si des avertissements d'acceptation de la valeur Null sont donnés. Chaque contexte est *désactivé* ou *activé*.

Les deux contextes peuvent être spécifiés au niveau du projet (en dehors du code source C#). La directive `#nullable` contrôle les contextes d'annotation et d'avertissement et elle est prioritaire sur les paramètres au niveau du projet. Une directive définit le ou les contextes qu'elle contrôle jusqu'à ce qu'une autre directive la remplace, ou jusqu'à la fin du fichier source.

L'effet des directives est le suivant :

- `#nullable disable` : définit les contextes d'avertissement et d'annotation pouvant accepter la valeur Null à *désactivés*.
- `#nullable enable` : définit les contextes d'avertissement et d'annotation pouvant accepter la valeur Null sur *activés*.
- `#nullable restore` : restaure les contextes d'avertissement et d'annotation pouvant accepter la valeur Null dans les paramètres du projet.
- `#nullable disable annotations` : définit le contexte d'annotation pouvant accepter la valeur Null sur *désactivé*.
- `#nullable enable annotations` : définit le contexte d'annotation pouvant accepter la valeur Null sur *activé*.
- `#nullable restore annotations` : restaure le contexte d'annotation pouvant accepter la valeur Null dans les paramètres du projet.
- `#nullable disable warnings` : définit le contexte d'avertissement nullable sur *désactivé*.
- `#nullable enable warnings` : définit le contexte d'avertissement nullable sur *activé*.

- `#nullable restore warnings` : restaure le contexte d'avertissement nullable avec les paramètres du projet.

Compilation conditionnelle

Vous utilisez quatre directives de préprocesseur pour contrôler la compilation conditionnelle :

- `#if` : ouvre une compilation conditionnelle, où le code est compilé uniquement si le symbole spécifié est défini.
- `#elif` : ferme la compilation conditionnelle précédente et ouvre une nouvelle compilation conditionnelle en fonction de la définition du symbole spécifié.
- `#else` : ferme la compilation conditionnelle précédente et ouvre une nouvelle compilation conditionnelle si le symbole spécifié précédent n'est pas défini.
- `#endif` : ferme la compilation conditionnelle précédente.

Le compilateur C# compile le code entre la directive `#if` et la directive `#endif` uniquement si le symbole spécifié est défini, ou non défini lorsque l'opérateur Not `!` est utilisé. Contrairement à C et C++, vous ne pouvez pas attribuer de valeur numérique à un symbole. L'instruction `#if` en C# est booléenne et teste uniquement si le symbole a été défini ou non. Par exemple, le code suivant est compilé quand `DEBUG` est défini :

```
C#

#if DEBUG
    Console.WriteLine("Debug version");
#endif
```

Le code suivant est compilé quand `MYTEST` n'est **pas** défini :

```
C#

#if !MYTEST
    Console.WriteLine("MYTEST is not defined");
#endif
```

Vous pouvez utiliser les opérateurs == (égalité) et != (inégalité) pour tester les valeurs bool `true` ou `false`. `true` signifie que le symbole est défini. L'instruction `#if DEBUG` a la même signification que `#if (DEBUG == true)`. Vous pouvez utiliser les opérateurs && (et), || (ou), et ! (non) pour déterminer si plusieurs symboles ont été définis. Vous pouvez également regrouper des symboles et des opérateurs à l'aide de parenthèses.

`#if`, ainsi que les directives `#else`, `#elif`, `#endif`, `#define` et `#undef`, vous permettent d'inclure ou d'exclure du code en fonction de l'existence d'un ou plusieurs symboles. La compilation conditionnelle peut être utile lors de la compilation du code pour une version Debug ou lors de la compilation d'une configuration spécifique.

Une directive conditionnelle commençant par une directive `#if` doit se terminer explicitement par une directive `#endif`. `#define` vous permet de définir un symbole. En utilisant ce dernier comme expression passée à la directive `#if`, l'expression correspond à `true`. Vous pouvez également définir un symbole avec l'option de compilateur DefineConstants. Vous pouvez annuler la définition d'un symbole avec `#undef`. La portée d'un symbole créé avec `#define` est le fichier dans lequel il a été défini. Un symbole que vous définissez avec **DefineConstants** ou `#define` n'est pas en conflit avec une variable du même nom. Autrement dit, un nom de variable ne doit pas être passé à une directive de préprocesseur, et un symbole ne peut être évalué que par une directive de préprocesseur.

`#elif` vous permet de créer une directive conditionnelle composée. L'expression `#elif` est évaluée si ni l'expression `#if` précédente ni une expression de directive `#elif` facultative précédente n'est évaluée à `true`. Si une expression `#elif` est évaluée à `true`, le compilateur évalue l'ensemble du code situé entre `#elif` et la directive conditionnelle suivante. Par exemple :

C#

```
#define VC7
//...
#if DEBUG
    Console.WriteLine("Debug build");
#elif VC7
    Console.WriteLine("Visual Studio 7");
#endif
```

`#else` vous permet de créer une directive conditionnelle composée de sorte que, si aucune des expressions contenues dans les précédentes directives `#if` ou `#elif` (facultative) n'a la valeur `true`, le compilateur évalue tout le code entre `#else` et la directive `#endif` suivante. `#endif`(#endif) doit être la directive de préprocesseur suivante après `#else`.

`#endif` spécifie la fin d'une directive conditionnelle, qui a commencé avec la directive `#if`.

Le système de génération tient également compte des symboles de préprocesseur prédéfinis représentant différents frameworks cibles dans les projets de type SDK. Ils sont utiles durant la création d'applications pouvant cibler plusieurs versions de .NET.

Versions cibles de .NET Framework	symboles	Symboles supplémentaires (disponibles dans les SDK .NET 5+)	Symboles de plateforme (disponibles uniquement lorsque vous spécifiez un moniker de framework cible spécifique au système d'exploitation)
.NET Framework	`NETFRAMEWORK`, `NET48`, `NET472`, `NET471`, `NET47`, `NET462`, `NET461`, `NET46`, `NET452`, `NET451`, `NET45`, `NET40`, `NET35`, `NET20`	`NET48_OR_GREATER`, `NET472_OR_GREATER`, `NET471_OR_GREATER`, `NET47_OR_GREATER`, `NET462_OR_GREATER`, `NET461_OR_GREATER`, `NET46_OR_GREATER`, `NET452_OR_GREATER`, `NET451_OR_GREATER`, `NET45_OR_GREATER`, `NET40_OR_GREATER`, `NET35_OR_GREATER`, `NET20_OR_GREATER`	
.NET Standard	`NETSTANDARD`, `NETSTANDARD2_1`, `NETSTANDARD2_0`, `NETSTANDARD1_6`, `NETSTANDARD1_5`, `NETSTANDARD1_4`, `NETSTANDARD1_3`, `NETSTANDARD1_2`, `NETSTANDARD1_1`, `NETSTANDARD1_0`	`NETSTANDARD2_1_OR_GREATER`, `NETSTANDARD2_0_OR_GREATER`, `NETSTANDARD1_6_OR_GREATER`, `NETSTANDARD1_5_OR_GREATER`, `NETSTANDARD1_4_OR_GREATER`, `NETSTANDARD1_3_OR_GREATER`, `NETSTANDARD1_2_OR_GREATER`, `NETSTANDARD1_1_OR_GREATER`, `NETSTANDARD1_0_OR_GREATER`	

Versions cibles de .NET Framework	symboles	Symboles supplémentaires (disponibles dans les SDK .NET 5+)	Symboles de plateforme (disponibles uniquement lorsque vous spécifiez un moniker de framework cible spécifique au système d'exploitation)
.NET 5+ (et .NET Core)	`NET`, `NET7_0`, `NET6_0`, `NET5_0`, `NETCOREAPP`, `NETCOREAPP3_1`, `NETCOREAPP3_0`, `NETCOREAPP2_2`, `NETCOREAPP2_1`, `NETCOREAPP2_0`, `NETCOREAPP1_1`, `NETCOREAPP1_0`	`NET7_0_OR_GREATER`, `NET6_0_OR_GREATER`, `NET5_0_OR_GREATER`, `NETCOREAPP3_1_OR_GREATER`, `NETCOREAPP3_0_OR_GREATER`, `NETCOREAPP2_2_OR_GREATER`, `NETCOREAPP2_1_OR_GREATER`, `NETCOREAPP2_0_OR_GREATER`, `NETCOREAPP1_1_OR_GREATER`, `NETCOREAPP1_0_OR_GREATER`	`ANDROID`, `IOS`, `MACCATALYST`, `MACOS`, `TVOS`, `WINDOWS`, `[OS][version]` (par exemple, `IOS15_1`), `[OS][version]_OR_GREATER` (par exemple `IOS15_1_OR_GREATER`)

> ⓘ Notes
>
> - Les symboles sans version sont définis indépendamment de la version que vous ciblez.
> - Les symboles spécifiques à la version sont définis uniquement pour la version que vous ciblez.
> - Les symboles `<framework>_OR_GREATER` sont définis pour la version que vous ciblez et toutes les versions antérieures. Par exemple, si vous ciblez .NET Framework 2.0, les symboles suivants sont définis : `NET20`, `NET20_OR_GREATER`, `NET11_OR_GREATER` et `NET10_OR_GREATER`.
> - Ils sont différents des monikers de framework cible utilisés par la propriété **MSBuild TargetFramework** et **NuGet**.

> ⓘ Notes
>
> Pour les projets non-SDK traditionnels, vous devez configurer manuellement les symboles de compilation conditionnelle pour les différents frameworks cibles dans Visual Studio via les pages de propriétés du projet.

D'autres symboles prédéfinis incluent les constantes `DEBUG` et `TRACE`. Vous pouvez remplacer les valeurs définies pour le projet à l'aide de `#define`. Le symbole DEBUG, par exemple, est automatiquement défini en fonction de vos propriétés de configuration de build (mode « Debug » ou « Release »).

L'exemple suivant montre comment définir un symbole `MYTEST` sur un fichier, puis tester les valeurs des symboles `MYTEST` et `DEBUG`. La sortie de cet exemple dépend du mode de configuration que vous avez utilisé pour créer le projet (**Debug** ou **Release**).

C#

```
#define MYTEST
using System;
public class MyClass
{
    static void Main()
    {
#if (DEBUG && !MYTEST)
        Console.WriteLine("DEBUG is defined");
#elif (!DEBUG && MYTEST)
        Console.WriteLine("MYTEST is defined");
#elif (DEBUG && MYTEST)
        Console.WriteLine("DEBUG and MYTEST are defined");
#else
        Console.WriteLine("DEBUG and MYTEST are not defined");
#endif
    }
}
```

L'exemple suivant montre comment tester différents frameworks cibles pour pouvoir utiliser les nouvelles API dans la mesure du possible :

C#

```
public class MyClass
{
    static void Main()
    {
#if NET40
        WebClient _client = new WebClient();
#else
        HttpClient _client = new HttpClient();
#endif
    }
    //...
}
```

Définition de symboles

Vous utilisez les deux directives de préprocesseur suivantes pour définir ou annuler la définition de symboles pour la compilation conditionnelle :

- `#define` : permet de définir un symbole.
- `#undef` : permet d'annuler la définition d'un symbole.

Vous utilisez `#define` pour définir un symbole. Quand vous utilisez le symbole sous forme d'expression passée à la directive `#if`, l'expression donne la valeur `true`, comme illustré dans l'exemple suivant :

```
C#

#define VERBOSE

#if VERBOSE
   Console.WriteLine("Verbose output version");
#endif
```

> ⓘ Notes
>
> La directive `#define` ne peut pas être utilisée pour déclarer des valeurs constantes comme c'est le cas en général en C et C++. Les constantes en C# sont définies plutôt comme des membres statiques d'une classe ou d'un struct. Si vous avez plusieurs constantes de ce type, créez une classe « Constantes » distincte pour les stocker.

Les symboles peuvent être utilisés pour spécifier des conditions de compilation. Vous pouvez tester le symbole avec `#if` ou `#elif`. Vous pouvez également utiliser ConditionalAttribute pour effectuer une compilation conditionnelle. Vous pouvez définir un symbole, mais vous ne pouvez pas attribuer de valeur à un symbole. La directive `#define` doit apparaître dans le fichier avant l'utilisation d'instructions qui ne sont pas également des directives de préprocesseur. Vous pouvez également définir un symbole avec l'option de compilateur **DefineConstants**. Vous pouvez annuler la définition d'un symbole avec `#undef`.

Définition de régions

Vous pouvez définir des régions de code qui peuvent être réduites dans un plan à l'aide des deux directives de préprocesseur suivantes :

- `#region` : permet de démarrer une région.
- `#endregion` : permet de mettre fin à une région.

`#region` vous permet de spécifier un bloc de code que vous pouvez développer ou réduire quand vous utilisez la fonctionnalité Mode Plan de l'éditeur de code. Dans les fichiers de code volumineux, il peut être pratique de réduire ou masquer une ou plusieurs régions pour vous concentrer sur la partie du fichier sur laquelle vous êtes en train de travailler. L'exemple suivant montre comment définir une région :

C#

```
#region MyClass definition
public class MyClass
{
    static void Main()
    {
    }
}
#endregion
```

Un bloc `#region` doit se terminer par une directive `#endregion`. Un bloc `#region` ne peut pas chevaucher un bloc `#if`. Toutefois, un bloc `#region` peut être imbriqué dans un bloc `#if` et, inversement, un bloc `#if` peut être imbriqué dans un bloc `#region`.

Informations d'erreur et d'avertissement

Vous demandez au compilateur de générer des erreurs et des avertissements du compilateur définis par l'utilisateur, ainsi que des informations de ligne de contrôle à l'aide des directives suivantes :

- `#error` : permet de générer une erreur du compilateur avec un message spécifié.
- `#warning` : permet de générer un avertissement du compilateur, avec un message spécifique.
- `#line` : permet de modifier le numéro de ligne imprimé avec les messages du compilateur.

`#error` vous permet de générer une erreur définie par l'utilisateur CS1029 à partir d'un emplacement spécifique dans votre code. Par exemple :

C#

```
#error Deprecated code in this method.
```

> ⓘ **Notes**
>
> Le compilateur traite `#error version` de manière spéciale et signale une erreur de compilateur CS8304, avec un message contenant les versions du compilateur et du langage utilisés.

`#warning` vous permet de générer un avertissement du compilateur de premier niveau CS1030 à partir d'un emplacement spécifique dans votre code. Par exemple :

C#

```
#warning Deprecated code in this method.
```

`#line` vous permet de changer la numérotation de lignes du compilateur et (éventuellement) le nom de fichier pour les erreurs et les avertissements.

L'exemple suivant montre comment signaler deux avertissements associés à des numéros de ligne. La directive `#line 200` assigne de force le numéro 200 à la ligne suivante (bien que le numéro par défaut soit 6) ; jusqu'à la prochaine directive `#line`, le nom de fichier indiqué est « Special ». La directive `#line default` rétablit la numérotation de lignes par défaut, qui compte les lignes renumérotées par la directive précédente.

C#

```
class MainClass
{
    static void Main()
    {
#line 200 "Special"
        int i;
        int j;
#line default
        char c;
        float f;
#line hidden // numbering not affected
        string s;
        double d;
    }
}
```

La compilation produit la sortie suivante :

Console

```
Special(200,13): warning CS0168: The variable 'i' is declared but never used
Special(201,13): warning CS0168: The variable 'j' is declared but never used
MainClass.cs(9,14): warning CS0168: The variable 'c' is declared but never
used
MainClass.cs(10,15): warning CS0168: The variable 'f' is declared but never
used
MainClass.cs(12,16): warning CS0168: The variable 's' is declared but never
used
MainClass.cs(13,16): warning CS0168: The variable 'd' is declared but never
used
```

La directive `#line` peut être utilisée dans une étape intermédiaire automatisée du processus de génération. Par exemple, si des lignes ont été supprimées du fichier de code source d'origine, mais que vous voulez que le compilateur continue de générer une sortie sur la base de la numérotation de lignes d'origine du fichier, vous pouvez supprimer les lignes et simuler ensuite la numérotation de lignes d'origine avec `#line`.

La directive `#line hidden` masque les lignes successives du débogueur, de sorte que quand le développeur parcourt le code, les lignes situées entre une directive `#line hidden` et la directive `#line` suivante (en supposant qu'il ne s'agit pas d'une autre directive `#line hidden`) sont ignorées. Cette option peut aussi être utilisée pour permettre à ASP.NET de différencier le code défini par l'utilisateur du code généré par l'ordinateur. Même si ASP.NET est le principal utilisateur de cette fonctionnalité, il est probable que d'autres générateurs de code source pourront l'utiliser.

Une directive `#line hidden` n'affecte ni les noms de fichiers ni les numéros de lignes dans les rapports d'erreurs. Autrement dit, si le compilateur repère une erreur dans un bloc masqué, il indique le nom du fichier et le numéro de ligne actuels de l'erreur.

La directive `#line filename` spécifie le nom de fichier que vous souhaitez voir apparaître dans la sortie du compilateur. Par défaut, le nom réel du fichier de code source est utilisé. Le nom de fichier doit être entre guillemets doubles (" ") et doit être précédé d'un numéro de ligne.

À compter de C# 10, vous pouvez utiliser une nouvelle forme de directive `#line` :

C#

```
#line (1, 1) - (5, 60) 10 "partial-class.cs"
/*34567*/int b = 0;
```

Les composants de cette forme sont les suivants :

- `(1, 1)` : ligne de début et colonne pour le premier caractère de la ligne qui suit la directive. Dans cet exemple, la ligne suivante est signalée comme ligne 1, colonne 1.
- `(5, 60)` : ligne de fin et colonne pour la région marquée.
- `10` : décalage de colonne pour que la directive `#line` prenne effet. Dans cet exemple, la 10e colonne serait signalée comme colonne 1. C'est là que commence la déclaration `int b = 0;`. Ce champ est facultatif. En cas d'omission, la directive prend effet sur la première colonne.
- `"partial-class.cs"` : nom du fichier de sortie.

L'exemple précédent génère l'avertissement suivant :

CLI .NET

```
partial-class.cs(1,5,1,6): warning CS0219: The variable 'b' is assigned but
its value is never used
```

Après le remappage, la variable `b` se trouve sur la première ligne, au caractère six, du fichier `partial-class.cs`.

Les langages spécifiques à un domaine utilisent généralement ce format pour fournir un meilleur mappage du fichier source à la sortie C# générée. L'utilisation la plus courante de cette directive `#line` étendue consiste à mapper des avertissements ou des erreurs qui apparaissent dans un fichier généré à la source d'origine. Prenons comme exemple cette page Razor :

razor

```
@page "/"
Time: @DateTime.NowAndThen
```

La propriété `DateTime.Now` a été tapée de manière incorrecte en tant que `DateTime.NowAndThen`. Le code C# généré pour cet extrait de code Razor ressemble à ce qui suit, dans `page.g.cs` :

C#

```
  _builder.Add("Time: ");
#line (2, 6) (2, 27) 15 "page.razor"
  _builder.Add(DateTime.NowAndThen);
```

La sortie du compilateur pour l'extrait de code précédent est la suivante :

CLI .NET

```
page.razor(2, 2, 2, 27)error CS0117: 'DateTime' does not contain a
definition for 'NowAndThen'
```

La ligne 2, colonne 6 dans `page.razor` est l'endroit où commence le texte `@DateTime.NowAndThen`. Voici ce qui est indiqué dans la directive : `(2, 6)`. Cette étendue de `@DateTime.NowAndThen` se termine à la ligne 2, colonne 27. Voici ce qui est indiqué dans la directive : `(2, 27)`. Le texte pour `DateTime.NowAndThen` commence dans la colonne 15 de `page.g.cs`. Voici ce qui est indiqué dans la directive : `15`. En plaçant tous les arguments ensemble, le compilateur signale l'erreur à son emplacement dans `page.razor`. Le développeur peut accéder directement à l'erreur dans son code source, et non dans la source générée.

Pour afficher d'autres exemples de ce format, consultez la spécification de fonctionnalité dans la section sur les exemples.

Pragmas

La directive `#pragma` fournit au compilateur des instructions spéciales pour la compilation du fichier dans lequel elle apparaît. Les instructions doivent être prises en charge par le compilateur. En d'autres termes, vous ne pouvez pas utiliser `#pragma` pour créer des instructions de prétraitement personnalisées.

- #pragma warning : permet d'activer ou de désactiver les avertissements.
- #pragma checksum : permet de générer une somme de contrôle.

C#

```
#pragma pragma-name pragma-arguments
```

Où `pragma-name` est le nom d'un pragma reconnu et `pragma-arguments` correspond aux arguments spécifiques au pragma.

#pragma warning

`#pragma warning` peut activer ou désactiver certains avertissements.

C#

```
#pragma warning disable warning-list
#pragma warning restore warning-list
```

Où `warning-list` est une liste séparée par des virgules de numéros d'avertissement. Le préfixe « CS » est facultatif. Quand aucun numéro d'avertissement n'est spécifié, `disable` désactive tous les avertissements et `restore` active tous les avertissements.

> ⓘ **Notes**
>
> Pour trouver les numéros d'avertissement dans Visual Studio, générez votre projet, puis recherchez les numéros d'avertissement dans la fenêtre **Sortie**.

`disable` prend effet à partir de la ligne suivante du fichier source. L'avertissement est restauré sur la ligne suivant l'élément `restore`. En l'absence d'élément `restore` dans le fichier, les avertissements sont restaurés à leur état par défaut à la première ligne de tous les fichiers ultérieurs de la même compilation.

C#

```
// pragma_warning.cs
using System;

#pragma warning disable 414, CS3021
[CLSCompliant(false)]
public class C
{
    int i = 1;
    static void Main()
    {
    }
}
#pragma warning restore CS3021
[CLSCompliant(false)]  // CS3021
public class D
{
    int i = 1;
    public static void F()
    {
    }
}
```

checksum #pragma

Génère des sommes de contrôle pour les fichiers sources afin de faciliter le débogage des pages ASP.NET.

C#

```
#pragma checksum "filename" "{guid}" "checksum bytes"
```

Où `"filename"` est le nom du fichier qui nécessite la surveillance des modifications ou des mises à jour, `"{guid}"` est l'identificateur global unique (GUID) de l'algorithme de hachage, et `"checksum_bytes"` est la chaîne de chiffres hexadécimaux représentant les octets de la somme de contrôle. Doit être un nombre pair de chiffres hexadécimaux. S'il y a un nombre impair de chiffres, un avertissement est généré au moment de la compilation et la directive est ignorée.

Le débogueur Visual Studio utilise une somme de contrôle pour s'assurer de toujours trouver la bonne source. Le compilateur calcule la somme de contrôle pour un fichier source, puis envoie la sortie vers le fichier de base de données du programme (PDB). Le débogueur utilise ensuite le fichier PDB à comparer avec la somme de contrôle qu'il calcule pour le fichier source.

Cette solution ne fonctionne pas pour les projets ASP.NET, car la somme de contrôle est calculée pour le fichier source généré, au lieu du fichier .aspx. Pour résoudre ce problème, `#pragma checksum` fournit une prise en charge de la somme de contrôle pour les pages ASP.NET.

Quand vous créez un projet ASP.NET dans Visual C#, le fichier source généré contient une somme de contrôle pour le fichier .aspx, à partir duquel la source est générée. Le compilateur écrit ensuite ces informations dans le fichier PDB.

Si le compilateur ne rencontre aucune directive `#pragma checksum` dans le fichier, il calcule la somme de contrôle et écrit la valeur dans le fichier PDB.

C#

```
class TestClass
{
    static int Main()
    {
        #pragma checksum "file.cs" "{406EA660-64CF-4C82-B6F0-42D48172A799}"
"ab007f1d23d9" // New checksum
    }
}
```

Options du compilateur C#

Article • 23/02/2023

Cette section décrit les options interprétées par le compilateur C#. Les options sont regroupées dans des articles distincts en fonction de ce qu'elles contrôlent, par exemple, des fonctionnalités de langage, la génération de code ou la sortie. Utilisez la table des matières pour naviguer parmi celles-ci.

Procédures pour définir des options

Il existe deux façons différentes de définir les options du compilateur dans les projets .NET :

- ***Dans votre fichier *.csproj***

  Vous pouvez ajouter des propriétés MSBuild pour n'importe quelle option de compilateur dans votre fichier **.csproj* au format XML. Le nom de la propriété est identique à l'option du compilateur. La valeur de la propriété définit la valeur de l'option du compilateur. Par exemple, l'extrait de code du fichier projet suivant définit la propriété `LangVersion`.

  ```xml
  <PropertyGroup>
    <LangVersion>preview</LangVersion>
  </PropertyGroup>
  ```

  Pour plus d'informations sur la définition d'options dans les fichiers projet, consultez l'article Propriétés MSBuild pour les projets de kit SDK .NET.

- ***Utilisation des pages de propriétés Visual Studio***

  Visual Studio fournit des pages de propriétés pour modifier les propriétés de build. Pour en savoir plus, consultez Gérer les propriétés du projet et de la solution - Windows ou Gérer les propriétés de projet et de solution – Mac.

Projets .NET Framework

> **ⓘ Important**
>
> Cette section s'applique uniquement aux projets .NET Framework.

En plus des mécanismes décrits ci-dessus, vous pouvez définir des options de compilateur à l'aide de deux méthodes supplémentaires pour les projets .NET Framework :

- **Arguments de ligne de commande pour les projets .NET Framework** : les projets .NET Framework utilisent *csc.exe* au lieu de `dotnet build` pour générer des projets. Vous pouvez spécifier des arguments de ligne de commande pour *csc.exe* pour les projets .NET Framework.
- **Pages ASP.NET compilées** : les projets .NET Framework utilisent une section du fichier *web.config* pour la compilation des pages. Pour le nouveau système de build et les projets ASP.NET Core, les options sont extraites du fichier projet.

Le mot pour certaines options de compilateur est passé des projets *csc.exe* et .NET Framework au nouveau système MSBuild. La nouvelle syntaxe est utilisée dans cette section. Les deux versions sont répertoriées en haut de chaque page. Pour *csc.exe*, tous les arguments sont répertoriés après l'option et un signe deux-points. Par exemple, l'option `-doc` apparaît sous la forme :

```
Console

-doc:DocFile.xml
```

Vous pouvez appeler le compilateur C# en tapant le nom de son fichier exécutable (*csc.exe*) dans une invite de commandes.

Pour les projets .NET Framework, vous pouvez également exécuter *csc.exe* à partir de la ligne de commande. Chaque option du compilateur est disponible sous deux formes : **-option** et **/option**. Dans les projets web .NET Framework, vous spécifiez des options pour compiler du code-behind dans le fichier *web.config*. Pour plus d'informations, consultez <compilateur> Élément.

Si vous utilisez la fenêtre **Invite de commandes développeur pour Visual Studio**, toutes les variables d'environnement nécessaires sont définies automatiquement. Pour obtenir des informations sur la façon d'accéder à cet outil, consultez Invite de commandes développeur pour Visual Studio.

Le fichier exécutable *csc.exe* se trouve généralement dans le dossier Microsoft.NET\Framework\<*Version*> sous le répertoire *Windows*. Son emplacement peut varier en fonction de la configuration exacte de l'ordinateur utilisé. Si plusieurs versions du .NET Framework sont installées sur votre ordinateur, vous trouverez plusieurs versions de ce fichier. Pour plus d'informations sur ces installations, consultez Guide pratique pour déterminer les versions installées du .NET Framework.

Options du compilateur C# pour les règles de fonctionnalité de langage

Article • 10/05/2023

Les options suivantes contrôlent la façon dont le compilateur interprète les fonctionnalités du langage. La nouvelle syntaxe MSBuild est affichée en **gras**. La syntaxe *csc.exe* plus ancienne est indiquée en `code style`.

- **CheckForOverflowUnderflow** / `-checked` : Générer des vérifications de dépassement de capacité.
- **AllowUnsafeBlocks** / `-unsafe` : autoriser le code « non sécurisé ».
- **DefineConstants** / `-define` : définir des symboles de compilation conditionnelle.
- **LangVersion** / `-langversion` : spécifier la version de langue telle que `default` (dernière version majeure) ou `latest` (dernière version, y compris les versions mineures).
- **Nullable** / `-nullable` : activer le contexte nullable ou les avertissements nullables.

CheckForOverflowUnderflow

L'option **CheckForOverflowUnderflow** contrôle le contexte de vérification de dépassement de capacité par défaut qui définit le comportement du programme si les dépassements arithmétiques entiers.

XML

```
<CheckForOverflowUnderflow>true</CheckForOverflowUnderflow>
```

Lorsque **CheckForOverflowUnderflow** est `true`, le contexte par défaut est un contexte vérifié et la vérification du dépassement de capacité est activée. Sinon, le contexte par défaut est un contexte non vérifié. La valeur par défaut de cette option est `false`. C'est-à-dire que la vérification du dépassement de capacité est désactivée.

Vous pouvez également contrôler explicitement le contexte de vérification de dépassement de capacité pour les parties de votre code à l'aide des instructions `checked` et `unchecked`.

Pour plus d'informations sur la façon dont le contexte de contrôle de dépassement de capacité affecte les opérations et les opérations affectées, consultez l'article sur les instructions checked et unchecked.

AllowUnsafeBlocks

L'option de compilateur **AllowUnsafeBlocks** permet la compilation de code utilisant le mot clé unsafe. La valeur par défaut de cette option est `false`, ce qui signifie que le code non sécurisé n'est pas autorisé.

XML

```
<AllowUnsafeBlocks>true</AllowUnsafeBlocks>
```

Pour plus d'informations sur le code unsafe, consultez Pointeurs et code unsafe.

DefineConstants

L'option **DefineConstants** définit des symboles dans tous les fichiers de code source de votre programme.

XML

```
<DefineConstants>name;name2</DefineConstants>
```

Cette option spécifie les noms d'un ou plusieurs symboles que vous souhaitez définir. L'option **DefineConstants** revient à utiliser une directive de préprocesseur #define, sauf que l'option de compilateur est appliquée à tous les fichiers du projet. Un symbole reste défini dans un fichier source jusqu'à ce qu'une directive #undef dans le fichier source en supprime la définition. Quand vous utilisez l'option `-define`, une directive `#undef` dans un fichier n'a aucun effet sur les autres fichiers de code source du projet. Vous pouvez utiliser les symboles créés par cette option avec #if, #else, #elif et #endif pour effectuer une compilation conditionnelle des fichiers sources. Le compilateur C# lui-même ne définit aucun symbole ni aucune macro que vous pouvez utiliser dans votre code source ; toutes les définitions de symbole doivent être définies par l'utilisateur.

> ⓘ **Notes**
>
> La directive C# `#define` ne permet de donner une valeur à un symbole, comme dans les langages tels que C++. Par exemple, l'option `#define` ne peut pas être utilisée pour créer une macro ni définir une constante. Si vous devez définir une constante, utilisez une variable `enum`. Si vous souhaitez créer une macro de style C++, envisagez l'alternative que constituent les génériques. Comme les macros sont notoirement sujettes aux erreurs, C# n'autorise pas leur utilisation, mais fournit des solutions plus sûres.

LangVersion

Force le compilateur à accepter uniquement la syntaxe incluse dans la spécification choisie du langage C#.

XML

```
<LangVersion>9.0</LangVersion>
```

Les valeurs suivantes sont valides :

Value	Signification
`preview`	Le compilateur accepte toute la syntaxe de langage valide de la dernière préversion.
`latest`	Le compilateur accepte la syntaxe de la dernière version publiée du compilateur (versions mineures incluses).
`latestMajor` ou `default`	Le compilateur accepte la syntaxe de la dernière version principale publiée du compilateur.
`11.0`	Le compilateur accepte uniquement la syntaxe incluse dans C# 11 ou une version antérieure.
`10.0`	Le compilateur accepte uniquement la syntaxe incluse dans C# 10 ou une version antérieure.
`9.0`	Le compilateur accepte uniquement la syntaxe incluse dans C# 9 ou une version antérieure.
`8.0`	Le compilateur accepte uniquement la syntaxe incluse dans C# 8.0 ou une version antérieure.
`7.3`	Le compilateur accepte uniquement la syntaxe incluse dans C# 7.3 ou une version antérieure.
`7.2`	Le compilateur accepte uniquement la syntaxe incluse dans C# 7.2 ou une version antérieure.
`7.1`	Le compilateur accepte uniquement la syntaxe incluse dans C# 7.1 ou une version antérieure.
`7`	Le compilateur accepte uniquement la syntaxe incluse dans C# 7.0 ou une version antérieure.

Value	Signification
`6`	Le compilateur accepte uniquement la syntaxe incluse dans C# 6.0 ou une version antérieure.
`5`	Le compilateur accepte uniquement la syntaxe incluse dans C# 5.0 ou une version antérieure.
`4`	Le compilateur accepte uniquement la syntaxe incluse dans C# 4.0 ou une version antérieure.
`3`	Le compilateur accepte uniquement la syntaxe incluse dans C# 3.0 ou une version antérieure.
`ISO-2` ou `2`	Le compilateur accepte uniquement la syntaxe incluse dans ISO/IEC 23270:2006 C# (2.0)
`ISO-1` ou `1`	Le compilateur accepte uniquement la syntaxe incluse dans ISO/IEC 23270:2003 C# (1.0/1.2)

La version du langage par défaut dépend du framework cible de votre application et de la version installée du kit SDK ou de Visual Studio. Ces règles sont définies dans le contrôle de version du langage C#.

> **ⓘ Important**
>
> La valeur `latest` n'est généralement pas recommandée. Avec elle, le compilateur active les fonctionnalités les plus récentes, même si ces fonctionnalités dépendent des mises à jour non incluses dans l'infrastructure cible configurée. Sans ce paramètre, votre projet utilise la version du compilateur recommandée pour votre framework cible. Vous pouvez mettre à jour l'infrastructure cible pour accéder aux fonctionnalités de langage plus récentes.

Les métadonnées référencées par votre application C# ne sont pas visées par l'option de compilateur **LangVersion**.

Sachant que chaque version du compilateur C# contient des extensions de la spécification du langage, **LangVersion** ne vous offre pas les fonctionnalités équivalentes d'une version antérieure du compilateur.

De plus, alors que les mises à jour de la version de C# coïncident généralement avec les versions principales de .NET, la nouvelle syntaxe et les nouvelles fonctionnalités ne sont pas nécessairement liées à cette version spécifique du framework. Chaque fonctionnalité spécifique a ses propres exigences minimales relatives à l'API .NET ou au Common

Language Runtime pour pouvoir s'exécuter sur des frameworks de bas niveau en incluant des packages NuGet ou d'autres bibliothèques.

Quel que soit le paramètre **LangVersion** que vous utilisez, utilisez la version active du common language runtime pour créer votre fichier .exe ou .dll. Les assemblys friend et **ModuleAssemblyName**, qui fonctionnent sous **-langversion:ISO-1** est une exception.

Pour obtenir d'autres façons de spécifier la version du langage C#, consultez Contrôle de version du langage C#.

Pour plus d'informations sur la façon de définir cette option du compilateur par programme, consultez LanguageVersion.

spécification du langage C#

Version	Lien	Description
C# 7.0 et versions ultérieures	ici.	Spécification du langage C# version 7 - Ébauche non officielle : .NET Foundation
C# 6.0	Télécharger le PDF	Norme ECMA-334 6e édition
C# 5.0	Télécharger le PDF	Norme ECMA-334 5e édition
C# 3.0	Télécharger DOC	Spécification du langage C# version 3.0 : Microsoft Corporation
C# 2.0	Télécharger le PDF	Norme ECMA-334 4e édition
C# 1.2	Télécharger DOC	Spécification du langage C# version 1.2 : Microsoft Corporation
C# 1.0	Télécharger DOC	Spécification du langage C# version 1.0 : Microsoft Corporation

Version minimale du SDK nécessaire pour prendre en charge toutes les fonctionnalités linguistiques

Le tableau suivant répertorie les versions minimales du SDK avec le compilateur C# qui prend en charge la version de langue correspondante :

Version de C#	SDK logiciel minimum

Version de C#	SDK logiciel minimum
C# 11	SDK Microsoft Visual Studio/Build Tools 2022 version 17.4 ou .NET 7
C# 10	SDK Microsoft Visual Studio/Build Tools 2022 ou .NET 6
C# 9.0	SDK Microsoft Visual Studio/Build Tools 2019 version 16.8 ou .NET 5
C# 8.0	Microsoft Visual Studio/Build Tools 2019, version 16.3 ou SDK .NET Core 3.0
C# 7.3	Microsoft Visual Studio/Build Tools 2017, version 15.7
C# 7.2	Microsoft Visual Studio/Build Tools 2017, version 15.5
C# 7.1	Microsoft Visual Studio/Build Tools 2017, version 15.3
C# 7.0	Microsoft Visual Studio/Build Tools 2017
C# 6	Microsoft Visual Studio/Build Tools 2015
C# 5	Microsoft Visual Studio/Build Tools 2012 ou compilateur .NET Framework 4.5 groupé
C# 4	Microsoft Visual Studio/Build Tools 2010 ou compilateur .NET Framework 4.0 groupé
C# 3	Microsoft Visual Studio/Build Tools 2008 ou compilateur .NET Framework 3.5 groupé
C# 2	Microsoft Visual Studio/Build Tools 2005 ou compilateur .NET Framework 2.0 groupé
C# 1.0/1.2	Compilateur Microsoft Visual Studio/Build Tools .NET 2002 ou groupé .NET Framework 1.0

Nullable

L'option **Nullable** vous permet de spécifier le contexte nullable. Elle peut être définie dans la configuration du projet à l'aide de la balise `<Nullable>` :

XML

```xml
<Nullable>enable</Nullable>
```

L'argument doit être l'un des arguments `enable`, `disable`, `warnings` ou `annotations`. L'argument `enable` active le contexte nullable. La spécification `disable` désactive le contexte nullable. Lorsque vous spécifiez l'argument `warnings`, le contexte

d'avertissement nullable est activé. Lorsque vous spécifiez l'argument `annotations`, le contexte d'annotation nullable est activé. Les valeurs sont décrites et expliquées dans l'article sur les contextes nullables. Vous pouvez en savoir plus sur les tâches impliquées dans l'activation des types de référence Nullable dans une base de code existante dans notre article sur les stratégies de migration nullables.

> ⓘ **Notes**
>
> Quand aucune valeur n'est définie, la valeur `disable` par défaut est appliquée, mais les modèles .NET 6 sont fournis par défaut avec la valeur **Nullable** définie sur `enable`.

L'analyse de flux est utilisée pour déduire la nullabilité des variables dans le code exécutable. La nullabilité déduite d'une variable est indépendante de la capacité null déclarée de la variable. Les appels de méthode sont analysés même lorsqu'ils sont omis de manière conditionnelle. Par exemple, Debug.Assert en mode mise en production.

L'appel de méthodes annotées avec les attributs suivants affecte également l'analyse des flux :

- Conditions préalables simples : AllowNullAttribute et DisallowNullAttribute
- Conditions post-conditions simples : MaybeNullAttribute et NotNullAttribute
- Conditions post-conditions conditionnelles : MaybeNullWhenAttribute et NotNullWhenAttribute
- DoesNotReturnIfAttribute (par exemple, `DoesNotReturnIf(false)` pour Debug.Assert) et DoesNotReturnAttribute
- NotNullIfNotNullAttribute
- Conditions post-membres : MemberNotNullAttribute(String) et MemberNotNullAttribute(String[])

> ⓘ **Important**
>
> Le contexte nullable global ne s'applique pas aux fichiers de code générés. Quel que soit ce paramètre, le contexte nullable est *désactivé* pour tout fichier source marqué comme généré. Un fichier est marqué comme généré de quatre façons :
>
> 1. Dans .editorconfig, spécifiez `generated_code = true` dans une section qui s'applique à ce fichier.
> 2. Placez `<auto-generated/>` ou `<auto-generated>` dans un commentaire en haut du fichier. Il peut se trouver sur n'importe quelle ligne de ce commentaire, mais le bloc de commentaires doit être le premier élément du fichier.

3. Le nom du fichier doit commencer par *TemporaryGeneratedFile_*.
4. Le nom du fichier doit se terminer par *.designer.cs*, *.generated.cs*, *.g.cs* ou *.g.i.cs*.

Les générateurs peuvent opter pour l'utilisation de la directive de préprocesseur **#nullable**.

Options du compilateur C# qui contrôlent la sortie du compilateur.

Article • 10/05/2023

Les options suivantes contrôlent la génération de sortie du compilateur.

MSBuild	*csc.exe*	Description
DocumentationFile	`-doc:`	Générez un fichier de documentation XML à partir des commentaires `///`.
OutputAssembly	`-out:`	Spécifiez le fichier d'assembly de sortie.
PlatformTarget	`-platform:`	Spécifiez le processeur de la plateforme cible.
ProduceReferenceAssembly	`-refout:`	Générez un assembly de référence.
TargetType	`-target:`	Spécifiez le type de l'assembly de sortie.

DocumentationFile

L'option **DocumentationFile** vous permet de placer des commentaires de documentation dans un fichier XML. Pour en savoir plus sur la documentation de votre code, consultez Balises recommandées pour les commentaires de documentation. La valeur spécifie le chemin d'accès au fichier XML de sortie. Le fichier XML contient les commentaires contenus dans les fichiers de code source de la compilation.

XML

```
<DocumentationFile>path/to/file.xml</DocumentationFile>
```

Le fichier de code source qui contient les instructions principales ou de niveau supérieur est généré en premier dans le code XML. Il est souvent souhaitable d'utiliser le fichier .xml généré avec IntelliSense. Le nom du fichier *.xml* doit être identique au nom de l'assembly. Le fichier *.xml* doit figurer dans le même répertoire que l'assembly. Quand l'assembly est référencé dans un projet Visual Studio, le fichier *.xml* est également trouvé. Pour plus d'informations sur la génération de commentaires de code, consultez Fourniture de commentaires de code. À moins que vous compiliez avec <TargetType:Module>, `file` contiendra des balises `<assembly>` et `</assembly>` spécifiant le nom du fichier qui contient le manifeste d'assembly pour le fichier de sortie.

Pour obtenir des exemples, consultez Comment utiliser les fonctionnalités de documentation XML.

> ⓘ **Notes**
>
> L'option **DocumentationFile** s'applique à tous les fichiers du projet. Pour désactiver les avertissements relatifs aux commentaires de documentation pour un fichier ou une section de code spécifique, utilisez **#pragma warning**.

Cette option peut être utilisée dans n'importe quel projet de style SDK .NET. Pour plus d'informations, consultez la propriété DocumentationFile.

OutputAssembly

L'option **OutputAssembly** spécifie le nom du fichier de sortie. Le chemin de sortie spécifie le dossier dans lequel la sortie du compilateur est placée.

```xml
<OutputAssembly>folder</OutputAssembly>
```

Spécifiez le nom complet et l'extension du fichier que vous voulez créer. Si vous ne spécifiez pas le nom du fichier de sortie, MSBuild utilise le nom du projet pour spécifier le nom de l'assembly de sortie. Les anciens projets de style utilisent les règles suivantes :

- Un fichier .exe prend son nom du fichier de code source qui contient la méthode `Main` ou les instructions de niveau supérieur.
- Un fichier .dll ou .netmodule prend le nom du premier fichier de code source.

Les modules générés dans le cadre d'une compilation deviennent des fichiers associés à un assembly également généré pendant la compilation. Utilisez ildasm.exe pour afficher le manifeste d'assembly et identifier les fichiers associés.

L'option de compilateur **OutputAssembly** est nécessaire pour faire d'un fichier exe la cible d'un assembly friend.

PlatformTarget

Spécifie la version du CLR qui peut exécuter l'assembly.

```xml
```

```
<PlatformTarget>anycpu</PlatformTarget>
```

- **anycpu** (valeur par défaut) compile votre assembly pour qu'il s'exécute sur n'importe quelle plateforme. Votre application s'exécute en tant que processus 64 bits dans la mesure du possible et repasse en 32 bits quand seul ce mode est disponible.
- **anycpu32bitpreferred** compile votre assembly pour qu'il s'exécute sur n'importe quelle plateforme. Votre application s'exécute en mode 32 bits sur les systèmes qui prennent en charge les applications 64 bits et 32 bits. Vous pouvez spécifier cette option uniquement pour les projets qui ciblent .NET Framework 4.5 ou version ultérieure.
- **ARM** compile votre assembly pour qu'il s'exécute sur un ordinateur doté d'un processeur ARM (Advanced RISC Machine).
- **ARM64** compile votre assembly pour s'exécuter par le CLR 64 bits sur un ordinateur qui dispose d'un processeur Advanced RISC Machine (ARM) qui prend en charge le jeu d'instructions A64.
- **x64** compile votre assembly pour qu'il soit exécuté par le CLR 64 bits sur un ordinateur qui prend en charge le jeu d'instructions AMD64 ou EM64T.
- **x86** compile votre assembly pour qu'il soit exécuté par le CLR 32 bits x86.
- **Itanium** compile votre assembly pour qu'il soit exécuté par le CLR 64 bits sur un ordinateur doté d'un processeur Itanium.

Sur un système d'exploitation Windows 64 bits :

- Les assemblys compilés avec **x86** s'exécutent sur le CLR 32 bits fonctionnant sous WOW64.
- Une DLL compilée avec **anycpu** s'exécute sur le même CLR que le processus dans lequel elle est chargée.
- Les fichiers exécutables compilés avec **anycpu** s'exécutent sur le CLR 64 bits.
- Les fichiers exécutables compilés avec **anycpu32bitpreferred** s'exécutent sur le CLR 32 bits.

Le paramètre **anycpu32bitpreferred** est valide uniquement pour les fichiers exécutables (.EXE) et nécessite .NET Framework 4.5 ou version ultérieure. Pour plus d'informations sur le développement d'une application s'exécutant sur un système d'exploitation Windows 64 bits, consultez Applications 64 bits.

Vous définissez l'option **PlatformTarget** à partir de la page de propriétés **Build** de votre projet dans Visual Studio.

Le comportement de **anycpu** présente des nuances supplémentaires sur .NET Core et .NET 5 et versions ultérieures. Lorsque vous définissez **anycpu**, publiez votre application

et exécutez-la avec le x86 `dotnet.exe` ou le x64 `dotnet.exe`. Pour les applications autonomes, l'étape `dotnet publish` empaquète l'exécutable pour le RID de configuration.

ProduceReferenceAssembly

L'option **ProduceReferenceAssembly** spécifie si le compilateur produit des assemblys de référence.

```xml
<ProduceReferenceAssembly>true</ProduceReferenceAssembly>
```

Les assemblys de référence sont un type spécial d'assembly qui contiennent uniquement la quantité minimale de métadonnées requise pour représenter la surface d'API publique de la bibliothèque. Ils incluent des déclarations pour tous les membres qui sont significatifs lors du référencement d'un assembly dans les outils de génération. Les assemblys de référence excluent toutes les implémentations de membres et toutes les déclarations de membres privés. Ces membres n'ont aucun impact observable sur leur contrat d'API. Pour plus d'informations, consultez Assemblys de référence dans le Guide de .NET.

Les options **ProduceReferenceAssembly** et **ProduceOnlyReferenceAssembly** s'excluent mutuellement.

En règle générale, vous n'avez pas besoin d'utiliser directement les fichiers d'assembly de référence. Par défaut, les assemblys de référence sont générés dans un sous-dossier `ref` du chemin intermédiaire (c'est-à-dire `obj/ref/`). Pour les générer plutôt sous le répertoire de sortie (c'est-à-dire `bin/ref/`), définissez `ProduceReferenceAssemblyInOutDir` sur `true` dans votre projet.

Le kit SDK .NET 6.0.200 a apporté une modification en déplaçant les assemblys de référence du répertoire de sortie vers le répertoire intermédiaire par défaut.

TargetType

L'option du compilateur **TargetType** peut être spécifiée sous l'une des formes différentes :

- **library** : pour créer une bibliothèque de code. **library** est la valeur par défaut.
- **exe** : pour créer un fichier .exe.

- **module** : pour créer un module.
- **winexe** : pour créer un programme Windows.
- **winmdobj** : pour créer un fichier *.winmdobj* intermédiaire.
- **appcontainerexe** : pour créer un fichier .exe pour les applications du Windows 8.x Store.

> ⓘ **Notes**
>
> Pour les cibles .NET Framework, sauf si vous spécifiez **module**, cette option entraîne le placement d'un manifeste d'assembly .NET Framework dans un fichier de sortie. Pour plus d'informations, consultez **Assemblys dans .NET** et **Attributs communs**.

XML

```
<TargetType>library</TargetType>
```

Le compilateur crée un seul manifeste d'assembly par compilation. Les informations sur tous les fichiers d'une compilation sont placées dans le manifeste de l'assembly. Lorsque vous générez plusieurs fichiers de sortie sur la ligne de commande, un seul manifeste d'assembly peut être créé et il doit aller dans le premier fichier de sortie spécifié sur la ligne de commande.

Si vous créez un assembly, vous pouvez indiquer que tout ou partie de votre code est conforme CLS avec l'attribut CLSCompliantAttribute.

bibliothèque

L'option **library** indique au compilateur de créer une bibliothèque de liens dynamiques (DLL) à la place d'un fichier exécutable (EXE). La DLL est créée avec l'extension *.dll*. À moins que l'option **OutputAssembly** spécifie autre chose, le fichier de sortie prend le nom du premier fichier d'entrée. Lors de la création d'un fichier *.dll*, une méthode Main n'est pas obligatoire.

exe

L'option **exe** indique au compilateur de créer une application console exécutable (EXE). Le fichier exécutable est créé avec l'extension .exe. Utilisez **winexe** pour créer un exécutable de programme Windows. À moins que l'option **OutputAssembly** spécifie autre chose, le fichier de sortie prend le nom du fichier d'entrée qui contient le point d'entrée (méthode Main ou instructions de niveau supérieur). Un seul et unique point d'entrée est requis dans les fichiers de code source qui sont compilés dans un

fichier .exe. L'option de compilateur **StartupObject** vous permet de spécifier la classe contenant la méthode `Main`, au cas où votre code possède plusieurs classes avec une méthode `Main`.

module

Cette option empêche le compilateur de générer un manifeste d'assembly. Par défaut, le fichier de sortie créé en effectuant une compilation avec cette option porte l'extension *.netmodule*. Un fichier qui n'a pas de manifeste d'assembly ne peut pas être chargé par le runtime .NET. Toutefois, un tel fichier peut être incorporé dans le manifeste d'un assembly avec **AddModules**. Si plusieurs modules sont créés dans une seule compilation, les types internal d'un module sont accessibles à d'autres modules dans la compilation. Quand le code d'un module référence des types `internal` dans un autre module, les deux modules doivent être incorporés dans un manifeste d'assembly, avec **AddModules**. La création d'un module n'est pas prise en charge dans l'environnement de développement Visual Studio.

winexe

L'option **winexe** indique au compilateur de créer un exécutable (EXE) de programme Windows. Le fichier exécutable est créé avec l'extension .exe. Un programme Windows est un programme qui fournit une interface utilisateur à partir de la bibliothèque .NET ou avec les API Windows. Utilisez **exe** pour créer une application console. À moins que l'option **OutputAssembly** spécifie autre chose, le fichier de sortie prend le nom du fichier d'entrée qui contient la méthode Main. Une seule et unique méthode `Main` est requise dans les fichiers de code source qui sont compilés dans un fichier .exe. L'option **StartupObject** vous permet de spécifier la classe contenant la méthode `Main`, au cas où votre code possède plusieurs classes avec une méthode `Main`.

winmdobj

Si vous utilisez l'option **winmdobj**, le compilateur crée un fichier *.winmdobj* intermédiaire que vous pouvez convertir en fichier binaire Windows Runtime (*.winmd*). Le fichier *.winmd* peut ensuite être consommé par des programmes JavaScript et C++, en plus des programmes en langage managé.

Le paramètre **winmdobj** signale au compilateur qu'un module intermédiaire est requis. Le fichier *.winmdobj* peut ensuite être acheminé via l'outil d'exportation WinMDExp pour produire un fichier de métadonnées Windows (*.winmd*). Le fichier *.winmd* contient à la fois le code de la bibliothèque d'origine et les métadonnées WinMD utilisées par

JavaScript ou C++ et par le Windows Runtime. La sortie d'un fichier compilé à l'aide de l'option de compilateur **winmdobj** est utilisée uniquement comme entrée pour l'outil d'exportation WimMDExp. Le fichier *.winmdobj* lui-même n'est pas référencé directement. À moins que vous utilisiez l'option **OutputAssembly**, le fichier de sortie adopte le nom du premier fichier d'entrée. Une méthode Main n'est pas nécessaire.

appcontainerexe

Si vous utilisez l'option de compilateur **appcontainerexe**, le compilateur crée un fichier exécutable Windows (*.exe*) qui doit être exécuté dans un conteneur d'application. Cette option est équivalente à -target:winexe, mais elle est destinée aux applications du Windows 8.x Store.

Pour exiger que l'application s'exécute dans un conteneur d'application, cette option définit un bit dans le fichier Portable Executable (PE). Lorsque ce bit est défini, une erreur se produit si la méthode CreateProcess tente de lancer l'exécutable en dehors d'un conteneur d'application. À moins que vous utilisiez l'option **OutputAssembly**, le fichier de sortie prend le nom du fichier d'entrée qui contient la méthode Main.

Options du compilateur C# qui spécifient les entrées

Article • 10/05/2023

Les options suivantes contrôlent les entrées du compilateur. La nouvelle syntaxe MSBuild est affichée en **gras**. La syntaxe *csc.exe* plus ancienne est présentée en `code style`.

- **References** / `-reference` ou `-references` : métadonnées de référence du fichier ou des fichiers d'assembly spécifiés.
- **AddModules** `-addmodule` / : ajoute un module (créé avec `target:module` sur cet assembly.)
- **EmbedInteropTypes** / `-link` : incorpore les métadonnées des fichiers d'assembly d'interopérabilité spécifiés.

Références

L'option **References** indique au compilateur d'importer les informations de type public du fichier spécifié dans le projet actuel, ce qui vous permet de référencer les métadonnées des fichiers d'assembly spécifiés.

XML

```xml
<Reference Include="filename" />
```

`filename` est le nom d'un fichier contenant un manifeste d'assembly. Pour importer plusieurs fichiers, incluez un élément **Reference** pour chaque fichier. Vous pouvez définir un alias en tant qu'élément enfant de l'élément **Reference** :

XML

```xml
<Reference Include="filename.dll">
  <Aliases>LS</Aliases>
</Reference>
```

Dans l'exemple précédent, `LS` est l'identificateur C# valide représentant un espace de noms racine qui contient tous les espaces de noms de l'assembly *filename.dll*. Les fichiers que vous importez doivent contenir un manifeste. Utilisez **AdditionalLibPaths** pour spécifier le répertoire dans lequel se trouvent une ou plusieurs références d'assembly. La rubrique **AdditionalLibPaths** décrit également les répertoires dans

lesquels le compilateur recherche les assemblys. Pour que le compilateur reconnaisse un type dans un assembly, et non dans un module, il doit être forcé à résoudre le type, ce que vous pouvez faire en définissant une instance du type. Il existe d'autres moyens de résoudre les noms de types dans un assembly pour le compilateur : par exemple, si vous héritez d'un type dans un assembly, le nom du type est ensuite reconnu par le compilateur. Il est parfois nécessaire de référencer deux versions différentes du même composant dans un assembly. Pour ce faire, utilisez l'élément **Aliases** sur l'élément **References** de chaque fichier afin de distinguer les deux fichiers. Cet alias est utilisé comme qualificateur pour le nom du composant et se traduit par le composant dans l'un des fichiers.

> ⓘ **Notes**
>
> Dans Visual Studio, utilisez la commande **Ajouter une référence**. Pour plus d'informations, consultez **Guide pratique pour ajouter ou supprimer des références à l'aide du gestionnaire de références**.

AddModules

Cette option ajoute à la compilation en cours un module créé à l'aide du commutateur `<TargetType>module</TargetType>`.

XML

```
<AddModule Include=file1 />
<AddModule Include=file2 />
```

Où `file`, `file2` sont des fichiers de sortie qui contiennent des métadonnées. Le fichier ne peut pas contenir de manifeste d'assembly. Pour importer plusieurs fichiers, séparez les noms des fichiers par une virgule ou un point-virgule. Tous les modules ajoutés avec **AddModules** doivent se trouver dans le même répertoire que le fichier de sortie au moment de l'exécution. Vous pouvez donc spécifier un module dans n'importe quel répertoire au moment de la compilation, mais le module doit se trouver dans le répertoire de l'application au moment de l'exécution. Si le module n'est pas dans le répertoire de l'application au moment de l'exécution, vous recevez une TypeLoadException. `file` ne peut pas contenir d'assembly. Par exemple, si le fichier de sortie a été créé avec l'option **TargetType** de **module**, ses métadonnées peuvent être importées avec **AddModules**.

Si le fichier de sortie a été créé avec une option **TargetType** autre que **module**, ses métadonnées ne peuvent pas être importées avec **AddModules** mais peuvent l'être avec l'option **References**.

EmbedInteropTypes

Fait que le compilateur rend disponible pour le projet en cours de compilation les informations de type COM des assemblys spécifiés.

XML

```
<References>
  <EmbedInteropTypes>file1;file2;file3</EmbedInteropTypes>
</References>
```

Où `file1;file2;file3` est une liste délimitée par des points-virgules de noms de fichiers d'assembly. Si le nom de fichier contient un espace, placez-le entre des guillemets. L'option **EmbedInteropTypes** vous permet de déployer une application dans laquelle des informations de type sont incorporées. L'application peut ensuite utiliser des types dans un assembly de runtime qui implémentent les informations de type incorporées sans nécessiter une référence à l'assembly de runtime. Si différentes versions de l'assembly de runtime sont publiées, l'application qui contient les informations de type incorporées peut fonctionner avec les différentes versions sans devoir être recompilée. Pour un exemple, consultez Procédure pas à pas : incorporation de types provenant d'assemblys managés.

L'option **EmbedInteropTypes** est particulièrement utile avec COM Interop. Vous pouvez incorporer des types COM afin que votre application ne nécessite plus un assembly PIA (Primary Interop Assembly) sur l'ordinateur cible. L'option **EmbedInteropTypes** indique au compilateur d'incorporer les informations de type COM provenant de l'assembly Interop référencé dans le code compilé résultant. Le type COM est identifié par la valeur du CLSID (GUID). Par conséquent, votre application peut s'exécuter sur un ordinateur cible où les mêmes types COM ont été installés avec les mêmes valeurs de CLSID. Les applications qui automatisent Microsoft Office sont un bon exemple. Étant donné que les applications comme Office gardent habituellement la même valeur de CLSID à travers différentes versions, votre application peut utiliser les types COM référencés dès lors que le .NET Framework 4 ou ultérieur est installé sur l'ordinateur cible, et que votre application utilise les méthodes, propriétés ou événements qui sont inclus dans les types COM référencés. L'option **EmbedInteropTypes** incorpore seulement les interfaces, les structures et les délégués. L'incorporation de classes COM n'est pas prise en charge.

> **Notes**
>
> Quand vous créez une instance d'un type COM incorporé dans votre code, vous devez créer l'instance à l'aide de l'interface appropriée. Une tentative de création d'une instance d'un type COM incorporé à l'aide de la coclasse provoque une erreur.

Comme l'option de compilateur **References**, l'option **EmbedInteropTypes** utilise le fichier réponse Csc.rsp, qui référence les assemblys .NET fréquemment utilisés. Utilisez l'option de compilateur **NoConfig** si vous ne voulez pas que le compilateur utilise le fichier Csc.rsp.

C#

```
// The following code causes an error if ISampleInterface is an embedded
interop type.
ISampleInterface<SampleType> sample;
```

Vous ne pouvez pas utiliser des types qui ont un paramètre générique dont le type est incorporé depuis un assembly d'interopérabilité si ce type provient d'un assembly externe. Cette restriction ne s'applique pas aux interfaces. Prenons l'exemple de l'interface Range qui est définie dans l'assembly Microsoft.Office.Interop.Excel. Si une bibliothèque incorpore les types interop de l'assembly Microsoft.Office.Interop.Excel et expose une méthode qui retourne un type générique qui a un paramètre dont le type est l'interface Range, cette méthode doit retourner une interface générique, comme dans l'exemple de code suivant.

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Office.Interop.Excel;

public class Utility
{
    // The following code causes an error when called by a client assembly.
    public List<Range> GetRange1()
    {
        return null;
    }

    // The following code is valid for calls from a client assembly.
    public IList<Range> GetRange2()
    {
```

```
        return null;
    }
}
```

Dans l'exemple suivant, le code client peut appeler la méthode qui retourne l'interface générique IList sans erreur.

C#

```
public class Client
{
    public void Main()
    {
        Utility util = new Utility();

        // The following code causes an error.
        List<Range> rangeList1 = util.GetRange1();

        // The following code is valid.
        List<Range> rangeList2 = (List<Range>)util.GetRange2();
    }
}
```

Options du compilateur C# pour signaler des erreurs et des avertissements

Article • 13/07/2023

Les options suivantes contrôlent la façon dont le compilateur signale des erreurs et des avertissements. La nouvelle syntaxe MSBuild est affichée en **gras**. La syntaxe *csc.exe* plus ancienne est indiquée dans `code style`.

- **WarningLevel** / `-warn`: définir le niveau d'avertissement.
- **AnalysisLevel**: définir le niveau d'avertissement facultatif.
- **TreatWarningsAsErrors** / `-warnaserror`: traiter tous les avertissements comme des erreurs
- **WarningsAsErrors** / `-warnaserror`: traiter un ou plusieurs avertissements comme des erreurs
- **WarningsNotAsErrors** / `-warnnotaserror`: na pas traiter un ou plusieurs avertissements comme des erreurs
- **NoWarn** / `-nowarn`: définir une liste d'avertissements désactivés.
- **CodeAnalysisRuleSet** / `-ruleset`: spécifiez un fichier d'ensemble de règles qui désactive des diagnostics spécifiques.
- **ErrorLog** / `-errorlog`: spécifiez un fichier pour enregistrer tous les diagnostics du compilateur et de l'analyseur.
- **ReportAnalyzer** / `-reportanalyzer`: signalez des informations supplémentaires sur l'analyseur, telles que le temps d'exécution.

WarningLevel

L'option **WarningLevel** spécifie le niveau d'avertissement pour que le compilateur s'affiche.

XML

```xml
<WarningLevel>3</WarningLevel>
```

La valeur de l'élément est le niveau d'avertissement que vous souhaitez afficher pour la compilation : les nombres inférieurs n'affichent que les avertissements de gravité élevée. Les nombres plus élevés affichent plus d'avertissements. La valeur doit être égale à zéro ou à un entier positif :

Niveau d'avertissement	Signification
0	Désactive l'émission de tous les messages d'avertissement.
1	Affiche les messages d'avertissement graves.
2	Affiche les avertissements de niveau 1, ainsi que certains avertissements moins graves, tels que les avertissements concernant le masquage de membres de classe.
3	Affiche les avertissements de niveau 2, ainsi que certains avertissements moins graves, tels que les avertissements concernant les expressions toujours évaluées à `true` ou `false`.
4 (défaut)	Affiche les avertissements de niveau 3, ainsi que les avertissements d'information.

> ⚠ **Avertissement**
>
> La ligne de commande du compilateur accepte les valeurs supérieures à 4 pour activer les **alertes d'onde d'avertissement**. Toutefois, le Kit de développement logiciel (SDK) .NET définit *WarningLevel* pour qu'il corresponde à *AnalysisLevel* dans votre fichier projet.

Pour obtenir des informations sur une erreur ou un avertissement, vous pouvez rechercher le code d'erreur dans l'index de l'aide. Il existe d'autres façons d'obtenir des informations sur une erreur ou un avertissement, qui sont décrites dans Erreurs du compilateur C#. Utilisez **TreatWarningsAsErrors** pour traiter tous les avertissements comme des erreurs. Utilisez **DisabledWarnings** pour désactiver certains avertissements.

Niveau d'analyse

L'option**AnalysisLevel** spécifie des ondes d'avertissement et des analyseurs supplémentaires à activer. Les avertissements sont des vérifications supplémentaires qui améliorent votre code ou garantissent sa compatibilité avec les versions à venir. Les analyseurs offrent une capacité similaire à celle d'une ligne de démarcation pour améliorer votre code.

XML

```
<AnalysisLevel>preview</AnalysisLevel>
```

Niveau d'analyse	Signification
5	Affiche tous les avertissements facultatifs de la vague d'avertissement 5.
6	Affiche tous les avertissements facultatifs de la vague d'avertissement 6.
7	Affiche tous les avertissements facultatifs de la vague d'avertissement 7.
Dernier (par défaut)	Affiche tous les avertissements d'information jusqu'à la version actuelle incluse.
preview	Affiche tous les avertissements d'information jusqu'à la dernière préversion incluse.
aucun	Désactive tous les avertissements d'information.

Pour plus d'informations sur les avertissements facultatifs, consultez Vagues d'avertissement.

Pour obtenir des informations sur une erreur ou un avertissement, vous pouvez rechercher le code d'erreur dans l'index de l'aide. Il existe d'autres façons d'obtenir des informations sur une erreur ou un avertissement, qui sont décrites dans Erreurs du compilateur C#. Utilisez **TreatWarningsAsErrors** pour traiter tous les avertissements comme des erreurs. Utilisez **NoWarn** pour désactiver certains avertissements.

TreatWarningsAsErrors

L'option **TreatWarningsAsErrors** traite tous les avertissements comme des erreurs. Vous pouvez également utiliser **TreatWarningsAsErrors** pour définir uniquement certains avertissements en tant qu'erreurs. Si vous activez **TreatWarningsAsErrors**, vous pouvez utiliser **WarningsNotAsErrors** pour répertorier les avertissements qui ne doivent pas être traités comme des erreurs.

XML

```
<TreatWarningsAsErrors>true</TreatWarningsAsErrors>
```

Tous les messages d'avertissement sont signalés comme des erreurs. Le processus de génération s'arrête (aucun fichier de sortie n'est généré). Par défaut, **TreatWarningsAsErrors** n'est pas en vigueur, ce qui signifie que les avertissements n'empêchent pas la génération d'un fichier de sortie. Le cas échéant, si vous voulez que seuls certains avertissements spécifiques soient considérés comme des erreurs, vous pouvez spécifier une liste séparée par des virgules qui liste les numéros d'avertissement à considérer comme des erreurs. L'ensemble des avertissements de valeur Null peut être

spécifié avec le raccourci **Nullable**. Utilisez **WarningLevel** pour spécifier le niveau d'avertissements que vous souhaitez faire afficher par le compilateur. Utilisez **NoWarn** pour désactiver certains avertissements.

> **ⓘ Important**
>
> Il existe deux différences subtiles entre l'utilisation de l'élément `<TreatWarningsAsErrors>` dans votre fichier *csproj* et l'utilisation du commutateur de ligne de commande MSBuild `warnaserror`. *TreatWarningsAsErrors* affecte uniquement le compilateur C#, et pas les autres tâches MSBuild dans votre fichier *csproj*. Le commutateur de ligne de commande `warnaserror` affecte toutes les tâches. Deuxièmement, le compilateur ne produit aucune sortie sur des avertissements lorsque *TreatWarningsAsErrors* est utilisé. Le compilateur produit une sortie lorsque le commutateur de ligne de commande `warnaserror` est utilisé.

WarningsAsErrors et WarningsNotAsErrors

Les **options WarningsAsErrors** et **WarningsNotAsErrors** remplacent l'option **TreatWarningsAsErrors** pour une liste d'avertissements. Cette option peut être utilisée avec tous les avertissements *CS*. Le préfixe « CS » est facultatif. Vous pouvez utiliser le nombre ou « CS » suivi du numéro d'erreur ou d'avertissement. Pour les autres éléments qui affectent les avertissements, consultez les propriétés MSBuild courantes.

Activez les avertissements 0219 et 0168 en tant qu'erreurs :

```xml
<WarningsAsErrors>0219,CS0168</WarningsAsErrors>
```

Désactivez les mêmes avertissements en tant qu'erreurs :

```xml
<WarningsNotAsErrors>0219,CS0168</WarningsNotAsErrors>
```

Vous utilisez **WarningsAsErrors** pour configurer un ensemble d'avertissements en tant qu'erreurs. Utilisez **WarningsNotAsErrors** pour configurer un ensemble d'avertissements qui ne doivent pas être des erreurs lorsque vous avez défini tous les avertissements comme des erreurs.

NoWarn

L'option **NoWarn** vous permet d'empêcher le compilateur d'afficher un ou plusieurs avertissements. Séparez les numéros des avertissements par une virgule.

```
XML

<NoWarn>number1, number2</NoWarn>
```

`number1`, `number2` nombre(s) d'avertissement que vous souhaitez faire supprimer par le compilateur. Vous spécifiez la partie numérique de l'identificateur d'avertissement. Par exemple, si vous souhaitez supprimer *CS0028*, spécifiez `<NoWarn>28</NoWarn>`. Le compilateur ignore silencieusement les numéros d'avertissement passés en **NoWarn** qui étaient valides dans les versions précédentes et qui ont été supprimés. Par exemple, *CS0679* était valide dans le compilateur dans Visual Studio .NET 2002, mais a été supprimé ultérieurement.

Les avertissements suivants ne peuvent pas être supprimés par l'option **NoWarn** :

- Avertissement du compilateur (niveau 1) CS2002
- Avertissement du compilateur (niveau 1) CS2023
- Avertissement du compilateur (niveau 1) CS2029

CodeAnalysisRuleSet

Spécifiez un fichier d'ensemble de règles qui configure des diagnostics spécifiques.

```
XML

<CodeAnalysisRuleSet>MyConfiguration.ruleset</CodeAnalysisRuleSet>
```

Où `MyConfiguration.ruleset` est le chemin d'accès au fichier d'ensemble de règles. Pour plus d'informations sur l'utilisation des ensembles de règles, consultez l'article de la documentation de Visual Studio sur les ensembles de règles.

ErrorLog

Spécifier un fichier pour journaliser tous les diagnostics du compilateur et de l'analyseur.

```
XML
```

```
<ErrorLog>compiler-diagnostics.sarif</ErrorLog>
```

L'option **ErrorLog** entraîne le compilateur à générer un journal SARIF (Static Analysis Results Interchange Format). Les journaux SARIF sont généralement lus par des outils qui analysent les résultats des diagnostics du compilateur et de l'analyseur.

Vous pouvez spécifier le format SARIF à l'aide de l'argument `version` sur l'élément `ErrorLog` :

XML

```
<ErrorLog>logVersion21.json,version=2.1</ErrorLog>
```

Le séparateur peut être une virgule (`,`) ou un point-virgule (`;`). Les valeurs valides pour la version sont : « 1 », « 2 » et « 2.1 ». La valeur par défaut est 1. « 2 » et « 2.1 » signifient tous deux SARIF version 2.1.0.

ReportAnalyzer

Créer un rapport sur les informations supplémentaires de l'analyseur, comme l'heure d'exécution.

XML

```
<ReportAnalyzer>true</ReportAnalyzer>
```

L'option **ReportAnalyzer** entraîne le compilateur à emmètre des informations supplémentaires du journal MSBuild qui détaillent les caractéristiques de performances des analyseurs de la compilation. Elle est généralement utilisée par les auteurs d'analyseurs dans le cadre de la validation de l'analyseur.

> **ⓘ Important**
>
> Les informations de journal supplémentaires générées par cet indicateur ne sont générées que lorsque l'option de ligne de commande `-verbosity:detailed` est utilisée. Pour plus d'informations, consultez l'article **des commutateurs** dans la documentation MSBuild.

Options du compilateur C# qui contrôlent la génération de code

Article • 13/03/2023

Les options suivantes contrôlent la génération du code par le compilateur. La nouvelle syntaxe MSBuild est affichée en **gras**. La syntaxe *csc.exe* plus ancienne est indiquée dans `code style`.

- **DebugType** / `-debug` : émettre (ou ne pas émettre) des informations de débogage.
- **Optimize** / `-optimize` : activer les optimisations.
- **Déterministic** / `-deterministic` : produire une sortie équivalente octet pour octet à partir de la même source d'entrée.
- **ProduceOnlyReferenceAssembly** / `-refonly` : produire un assembly de référence, au lieu d'un assembly complet, en tant que sortie principale.

DebugType

L'option **DebugType** fait en sorte que le compilateur génère des informations de débogage et les place dans les fichiers de sortie. Les informations de débogage sont ajoutées par défaut.

XML

```
<DebugType>pdbonly</DebugType>
```

Pour toutes les versions du compilateur à partir de C# 6.0, il n'existe aucune différence entre *pdbonly* et *full*. Choisissez *pdbonly*. Pour modifier l'emplacement du fichier *.pdb*, consultez **PdbFile**.

Les valeurs suivantes sont valides :

Value	Signification
`full`	Émettez des informations de débogage dans le fichier *.pdb* à l'aide du format par défaut pour la plateforme actuelle : **Windows** : un fichier pdb Windows. **Linux/macOS** : un fichier PDB portable.
`pdbonly`	Comme pour `full`. Pour plus d'informations, consultez la remarque ci-dessous.

Value	Signification
`portable`	Émettez des informations de débogage dans le fichier .pdb à l'aide du format PDB portable multiplateforme.
`embedded`	Émettez des informations de débogage dans le *.dll/.exe* lui-même (le fichier *.pdb* n'est pas produit) à l'aide du format PDB portable.

> **Important**
>
> Les informations suivantes s'appliquent uniquement aux compilateurs antérieurs à C# 6.0. La valeur de cet élément peut être `full` ou `pdbonly`. L'argument *full*, qui est en vigueur si vous ne spécifiez pas *pdbonly*, permet d'attacher un débogueur pour le programme en cours d'exécution. Le fait de spécifier *pdbonly* active le débogage du code source quand le programme est démarré dans le débogueur, mais affiche uniquement un assembleur quand le programme en cours d'exécution est attaché au débogueur. Utilisez cette option pour créer des versions Debug. Si vous utilisez *Full*, sachez qu'il y a un impact sur la vitesse et la taille du code optimisé JIT, ainsi qu'un faible impact sur la qualité du code avec *full*. Nous vous recommandons d'utiliser *pdbonly* ou aucun PDB pour la génération du code de version finale. L'une des différences entre *pdbonly* et *full* est qu'avec *full*, le compilateur émet un **DebuggableAttribute**, qui est utilisé pour signaler au compilateur JIT que des informations de débogage sont disponibles. Ainsi, une erreur se produit si votre code contient un **DebuggableAttribute** avec la valeur false si vous utilisez *full*. Pour plus d'informations sur la façon de configurer les performances de débogage d'une application, consultez **Simplification du débogage d'une image**.

Optimiser

L'option **Optimize** active ou désactive les optimisations effectuées par le compilateur pour réduire la taille de votre fichier de sortie et le rendre plus rapide et plus efficace. L'option *Optimize* est activée par défaut pour une configuration de build *Release*. Elle est désactivée par défaut pour une configuration de build *Debug*.

XML

```
<Optimize>true</Optimize>
```

Vous définissez l'option **Optimize** à partir de la page de propriétés **Build** pour votre projet dans Visual Studio.

Optimize indique aussi au Common Language Runtime d'optimiser le code au moment de l'exécution. Par défaut, les optimisations sont désactivées. Pour activer les optimisations, spécifiez **Optimize+**. Quand vous créez un module destiné à être utilisé par un assembly, utilisez les mêmes paramètres **Optimize** que ceux de l'assembly. Il est possible de combiner les options **Optimize** et **Debug**.

Déterministe

Indique au compilateur de générer un assembly dont la sortie octet-pour-octet est identique dans les compilations pour les entrées identiques.

XML

```
<Deterministic>true</Deterministic>
```

Par défaut, la sortie du compilateur d'un ensemble donné d'entrées est unique, car le compilateur ajoute un timestamp et un MVID (un Module.ModuleVersionId. Fondamentalement, il s'agit d'un GUID qui identifie de manière unique le module et la version.) qui est généré à partir de nombres aléatoires. Utilisez l'option `<Deterministic>` pour générer un *assembly déterministe*, dont le contenu binaire sera identique dans les compilations tant que l'entrée restera la même. Dans une telle build, les champs timestamp et MVID sont remplacés par des valeurs dérivées d'un hachage de toutes les entrées de compilation. Le compilateur prend en compte les entrées suivantes qui affectent le déterminisme :

- La séquence des paramètres de ligne de commande.
- Le contenu du fichier réponse .rsp du compilateur.
- La version précise du compilateur utilisée et ses assemblys référencés.
- Le chemin de répertoire actif.
- Le contenu binaire de tous les fichiers explicitement passés au compilateur directement ou indirectement, notamment :
  - Fichiers sources
  - Assemblys référencés
  - Modules référencés
  - Ressources
  - Fichier de clé de nom fort
  - Fichiers réponse @
  - Analyseurs
  - Ensembles de règles
  - Autres fichiers susceptibles d'être utilisés par les analyseurs

- La culture actuelle (pour le langage dans lequel les diagnostics et les messages d'exception sont produits).
- L'encodage par défaut (ou la page de codes active) si l'encodage n'est pas spécifié.
- L'existence, la non-existence et le contenu des fichiers sur les chemins de recherche du compilateur (spécifiés, par exemple, par `-lib` ou `-recurse`).
- Plateforme CLR (Common Language Runtime) sur laquelle le compilateur est exécuté.
- La valeur de `%LIBPATH%`, qui peut affecter le chargement des dépendances de l'analyseur.

La compilation déterministe peut servir à établir si un fichier binaire est compilé à partir d'une source approuvée. La sortie déterministe peut être utile lorsque la source est disponible publiquement. Elle peut également déterminer si les étapes de génération dépendent des modifications apportées au fichier binaire utilisé dans le processus de génération.

ProduceOnlyReferenceAssembly

L'option **ProduceOnlyReferenceAssembly** indique qu'un assembly de référence doit être généré à la place d'un assembly d'implémentation, en tant que sortie principale. Le paramètre **ProduceOnlyReferenceAssembly** désactive sans assistance la génération de fichiers PDB, car les assemblys de référence ne peuvent pas être exécutés.

XML

```
<ProduceOnlyReferenceAssembly>true</ProduceOnlyReferenceAssembly>
```

Les assemblys de référence sont un type spécial d'assembly. Les assemblys de référence contiennent uniquement la quantité minimale de métadonnées requises pour représenter la surface d'API publique de la bibliothèque. Ils incluent des déclarations pour tous les membres qui sont significatifs lors du référencement d'un assembly dans les outils de génération, mais excluent toutes les implémentations de membres et déclarations de membres privés qui n'ont aucun impact observable sur leur contrat d'API. Pour plus d'informations, consultez Assemblys de référence.

Les options **ProduceOnlyReferenceAssembly** et **ProduceReferenceAssembly** s'excluent mutuellement.

Options du compilateur C# pour la sécurité

Article • 10/05/2023

Les options suivantes contrôlent les options de sécurité du compilateur. La nouvelle syntaxe MSBuild est affichée en **gras**. La syntaxe *csc.exe* plus ancienne est présentée en `code style`.

- **PublicSign** / `-publicsign` : signez publiquement l'assembly.
- **DelaySign** / `-delaysign` : différez la signature de l'assembly en utilisant uniquement la partie publique de la clé de nom fort.
- **KeyFile** / `-keyfile` : spécifiez un fichier de clé de nom fort.
- **KeyContainer** / `-keycontainer` : spécifiez un conteneur de clé de nom fort.
- **HighEntropyVA** / `-highentropyva` : activez la randomisation du layout de l'espace d'adressage haute entropie (ASLR)

PublicSign

Cette option force le compilateur à appliquer une clé publique sans signer l'assembly. L'option **PublicSign** définit également un bit dans l'assembly qui indique au runtime que le fichier est signé.

XML

```
<PublicSign>true</PublicSign>
```

L'option **PublicSign** nécessite l'utilisation de l'option **KeyFile** ou **KeyContainer**. Les options **KeyFile** ou **KeyContainer** spécifient la clé publique. Les options **PublicSign** et **DelaySign** s'excluent mutuellement. Parfois appelé « faux signe » ou « signature OSS », la signature publique comprend la clé publique dans un assembly de sortie et définit l'indicateur « signé ». La signature publique ne signe pas réellement l'assembly avec une clé privée. Les développeurs utilisent la signature publique pour les projets open source. Les personnes créent des assemblys qui sont compatibles avec les assemblys « entièrement signés » publiés lorsqu'ils n'ont pas accès à la clé privée utilisée pour signer les assemblys. Puisque peu de consommateurs doivent vérifier si l'assembly est totalement signé, ces assemblys générés publiquement peuvent être utilisés dans la quasi-totalité des scénarios employant des assemblys totalement signés.

DelaySign

Cette option fait en sorte que le compilateur réserve de l'espace dans le fichier de sortie afin qu'une signature numérique puisse être ajoutée ultérieurement.

XML

```
<DelaySign>true</DelaySign>
```

Utilisez **DelaySign-** si vous souhaitez obtenir un assembly complètement signé. Utilisez **DelaySign** si vous souhaitez uniquement placer la clé publique dans l'assembly. L'option **DelaySign** est sans effet sauf si elle est utilisée avec **KeyFile** ou **KeyContainer**. Les options **KeyContainer** et **PublicSign** s'excluent mutuellement. Quand vous demandez un assembly totalement signé, le compilateur hache le fichier qui contient le manifeste (métadonnées de l'assembly) et signe ce hachage avec la clé privée. L'opération crée une signature numérique qui est stockée dans le fichier contenant le manifeste. Lorsqu'un assembly est signé en différé, le compilateur ne calcule pas et ne stocke pas la signature. Au lieu de cela, le compilateur réserve de l'espace dans le fichier pour que la signature puisse être ajoutée ultérieurement.

Par exemple, l'utilisation de **DelaySign** permet à un testeur de placer l'assembly dans le cache global. Après un test, vous pouvez signer complètement l'assembly en plaçant la clé privée dans l'assembly à l'aide de l'utilitaire Assembly Linker. Pour plus d'informations, consultez Création et utilisation d'assemblys avec nom fort et Différer la signature d'un assembly.

KeyFile

Spécifie le nom du fichier contenant la clé de chiffrement.

XML

```
<KeyFile>filename</KeyFile>
```

`file` est le nom du fichier contenant la clé de nom fort. Quand cette option est utilisée, le compilateur insère la clé publique du fichier spécifié dans le manifeste d'assembly, puis signe l'assembly final avec la clé privée. Pour générer un fichier de clé, tapez `sn -k file` à la ligne de commande. Si vous compilez avec **-target:module**, le nom du fichier de clé est conservé dans le module et incorporé dans l'assembly créé quand vous compilez un assembly avec **AddModules**. Vous pouvez également passer vos informations de chiffrement au compilateur avec **KeyContainer**. Utilisez **-DelaySign** si

vous voulez obtenir un assembly partiellement signé. Si **KeyFile** et **KeyContainer** sont spécifiés dans la même compilation, le compilateur essaie d'abord d'utiliser le conteneur de clé. Si cette tentative réussit, l'assembly est signé avec les informations figurant dans le conteneur de clé. Si le compilateur ne trouve pas le conteneur de clé, il essaie d'utiliser le fichier spécifié avec **KeyFile**. En cas de tentative, l'assembly est signé avec les informations contenues dans le fichier de clé et les informations sur la clé sont installées dans le conteneur de clé. Lors de la compilation suivante, le conteneur de clé sera valide. Un fichier de clé peut contenir uniquement la clé publique. Pour plus d'informations, consultez Création et utilisation d'assemblys avec nom fort et Différer la signature d'un assembly.

KeyContainer

Spécifie le nom du conteneur de la clé de chiffrement.

```
XML

<KeyContainer>container</KeyContainer>
```

`container` est le nom du conteneur de clé de nom fort. Quand l'option **KeyContainer** est utilisée, le compilateur crée un composant pouvant être partagé. Le compilateur insère une clé publique à partir du conteneur spécifié dans le manifeste d'assembly et signe l'assembly final avec la clé privée. Pour générer un fichier de clé, tapez `sn -k file` à la ligne de commande. `sn -i` installe la paire de clés dans un conteneur. Cette option n'est pas prise en charge quand le compilateur s'exécute sur CoreCLR. Pour signer un assembly en cas de génération sur CoreCLR, utilisez l'option **KeyFile**. Si vous compilez avec **TargetType**, le nom du fichier de clé est conservé dans le module et incorporé dans l'assembly quand vous compilez ce module dans un assembly avec **AddModules**. Vous pouvez également spécifier cette option comme attribut personnalisé (System.Reflection.AssemblyKeyNameAttribute) dans le code source de n'importe quel module MSIL (Microsoft Intermediate Language). Vous pouvez également passer vos informations de chiffrement au compilateur avec **KeyFile**. Utilisez **DelaySign** pour ajouter la clé publique au manifeste d'assembly, mais différer la signature de l'assembly tant qu'il n'a pas été testé. Pour plus d'informations, consultez Création et utilisation d'assemblys avec nom fort et Différer la signature d'un assembly.

HighEntropyVA

L'option de compilateur **HighEntropyVA** indique au noyau Windows si un fichier exécutable particulier prend en charge la randomisation du layout de l'espace

d’adressage (ASLR) de forte entropie.

XML

```
<HighEntropyVA>true</HighEntropyVA>
```

Cette option spécifie qu’un exécutable 64 bits ou un exécutable marqué par l’option du compilateur **PlatformTarget** prend en charge un espace d’adressage virtuel d’entropie élevée. L’option est activée par défaut pour toutes les versions de .NET Standard et .NET Core, ainsi que pour les versions de .NET Framework à compter de .NET Framework 4.5.

L’option **HighEntropyVA** permet à des versions compatibles du noyau Windows d’utiliser un degré d’entropie plus élevé lors de la randomisation du layout de l’espace d’adressage d’un processus. En utilisant un degré d’entropie plus élevé, vous pouvez allouer un plus grand nombre d’adresses aux zones de mémoire, telles que les piles et les tas. Par conséquent, il est plus difficile de deviner l’emplacement d’une zone de mémoire. L’option du compilateur **HighEntropyVA** nécessite que l’exécutable cible et tous les modules dont il dépend puissent être capables de gérer les valeurs de pointeur supérieures à 4 Go lorsqu’ils sont exécutés en tant que processus 64 bits.

Options de compilateur C# qui spécifient les ressources

Article • 09/05/2023

Les options suivantes contrôlent la façon dont le compilateur C# crée ou importe des ressources Win32. La nouvelle syntaxe MSBuild est affichée en **gras**. La syntaxe *csc.exe* plus ancienne est indiquée en `code style`.

- **Win32Resource** / `-win32res` : spécifiez un fichier de ressources Win32 (.res).
- **Win32Icon** / `-win32icon` : référencez les métadonnées du fichier ou des fichiers d'assembly spécifiés.
- **Win32Manifest** / `-win32manifest` : spécifiez un fichier manifeste Win32 (.xml).
- **NoWin32Manifest** / `-nowin32manifest` : n'incluez pas le manifeste Win32 par défaut.
- **Ressources** / `-resource` : incorporez la ressource spécifiée (forme abrégée : /res).
- **LinkResources** / `-linkresources` : liez la ressource spécifiée à cet assembly.

Win32Resource

L'option **Win32Resource** insère une ressource Win32 dans le fichier de sortie.

XML

```
<Win32Resource>filename</Win32Resource>
```

`filename` est le fichier de ressources que vous voulez ajouter à votre fichier de sortie. Une ressource Win32 peut contenir des informations sur la version ou le fichier bitmap (icône) qui permettent d'identifier votre application dans l'Explorateur de fichiers. Si vous ne spécifiez pas cette option, le compilateur génère des informations de version en fonction de la version de l'assembly.

Win32Icon

L'option **Win32Icon** insère un fichier .ico dans le fichier de sortie, ce qui lui donne l'apparence souhaitée dans l'Explorateur de fichiers.

XML

```
<Win32Icon>filename</Win32Icon>
```

`filename` est le fichier *.ico* que vous voulez ajouter à votre fichier de sortie. Un fichier *.ico* peut être créé avec le compilateur de ressources. Le compilateur de ressources est appelé quand vous compilez un programme Visual C++ ; un fichier *.ico* est alors créé à partir du fichier *.rc*.

Win32Manifest

Utilisez l'option **Win32Manifest** pour spécifier un fichier manifeste d'application Win32 défini par l'utilisateur à incorporer dans le fichier exécutable portable (PE) d'un projet.

```
XML

<Win32Manifest>filename</Win32Manifest>
```

`filename` est le nom et l'emplacement du fichier manifeste personnalisé. Par défaut, le compilateur C# incorpore un manifeste d'application qui spécifie le niveau d'exécution demandé « asInvoker ». Il crée le manifeste dans le même dossier que celui dans lequel l'exécutable est généré. Pour fournir un manifeste personnalisé, par exemple pour spécifier le niveau d'exécution requis « highestAvailable » ou « requireAdministrator », utilisez cette option pour indiquer le nom du fichier.

> **Notes**
>
> Cette option et l'option **Win32Resources** s'excluent mutuellement. Si vous tentez d'utiliser ces deux options dans la même ligne de commande, une erreur de build se produira.

Une application sans manifeste d'application pour spécifier le niveau d'exécution demandé est soumise à une virtualisation des fichiers et des registres sous la fonctionnalité Contrôle de compte d'utilisateur de Windows. Pour plus d'informations, consultez Contrôle de compte d'utilisateur.

Votre application sera soumise à la virtualisation si l'une de ces conditions est vérifiée :

- Vous utilisez l'option **NoWin32Manifest** sans fournir de manifeste au cours d'une étape de génération ultérieure ou en tant que partie du fichier de ressources Windows (*.res*) à l'aide de l'option **Win32Resource**.
- Vous fournissez un manifeste personnalisé qui ne spécifie pas de niveau d'exécution demandé.

Visual Studio crée un fichier *.manifest* par défaut et le stocke dans les répertoires de débogage et de mise en production avec le fichier exécutable. Vous pouvez ajouter un manifeste personnalisé en en créant un dans un éditeur de texte et en ajoutant le fichier au projet. Vous pouvez également cliquer avec le bouton droit sur l'icône **Projet** dans **Explorateur de solutions**, sélectionner **Ajouter un nouvel élément**, puis sélectionner **Fichier manifeste de l'application** . Une fois que vous avez ajouté votre fichier manifeste nouveau ou existant, il apparaît dans la liste déroulante **Manifeste**. Pour plus d'informations, consultez Page Application, Concepteur de projets (C#).

Vous pouvez fournir le manifeste de l'application en tant qu'étape après génération personnalisée ou en tant que partie d'un fichier de ressources Win32 à l'aide de l'option **NoWin32Manifest**. Utilisez cette même option pour que votre application soit soumise à la virtualisation des fichiers ou des registres dans Windows Vista.

NoWin32Manifest

Utilisez l'option **NoWin32Manifest** pour indiquer au compilateur de ne pas incorporer le manifeste de l'application dans le fichier exécutable.

XML

```
<NoWin32Manifest />
```

Quand cette option est utilisée, l'application est soumise à la virtualisation sur Windows Vista, sauf si vous fournissez un manifeste de l'application dans un fichier de ressources Win32 ou au cours d'une étape de génération ultérieure.

Dans Visual Studio, définissez cette option dans la page de propriétés **Application** en sélectionnant l'option **Créer une application sans fichier manifeste** dans la zone de liste déroulante **Manifeste**. Pour plus d'informations, consultez Page Application, Concepteur de projets (C#).

Ressources

Incorpore la ressource spécifiée dans le fichier de sortie.

XML

```
<Resources Include=filename>
  <LogicalName>identifier</LogicalName>
  <Access>accessibility-modifier</Access>
</Resources>
```

`filename` est le fichier de ressources .NET que vous voulez incorporer dans le fichier de sortie. `identifier` (facultatif) est le nom logique de la ressource. Il s'agit du nom utilisé pour charger la ressource. La valeur par défaut est le nom du fichier. `accessibility-modifier` (facultatif) est l'accessibilité de la ressource : publique ou privée. La valeur par défaut est public. Par défaut, les ressources sont publiques dans l'assembly quand elles sont créées à l'aide du compilateur C#. Pour rendre les ressources privées, spécifiez le modificateur d'accessibilité `private`. Aucune accessibilité autre `public` ou `private` n'est autorisée. Si `filename` est un fichier de ressources .NET créé, par exemple, par Resgen.exe ou dans l'environnement de développement, il est possible d'y accéder à l'aide des membres de l'espace de noms System.Resources. Pour plus d'informations, consultez System.Resources.ResourceManager. Pour toutes les autres ressources, utilisez les méthodes `GetManifestResource` dans la classe Assembly pour accéder à la ressource au moment de l'exécution. L'ordre des ressources dans le fichier de sortie est déterminé par l'ordre spécifié dans le fichier projet.

LinkResources

Crée un lien vers une ressource .NET dans le fichier de sortie. Le fichier de ressources n'est pas ajouté au fichier de sortie. **LinkResources** diffère de l'option **Ressource** qui incorpore un fichier de ressources dans le fichier de sortie.

XML

```
<LinkResources Include=filename>
  <LogicalName>identifier</LogicalName>
  <Access>accessibility-modifier</Access>
</LinkResources>
```

`filename` est le fichier de ressources .NET que vous voulez lier à partir de l'assembly. `identifier` (facultatif) est le nom logique de la ressource. Il s'agit du nom utilisé pour charger la ressource. La valeur par défaut est le nom du fichier. `accessibility-modifier` (facultatif) est l'accessibilité de la ressource : publique ou privée. La valeur par défaut est public. Par défaut, les ressources liées sont publiques dans l'assembly quand elles sont créées avec le compilateur C#. Pour rendre les ressources privées, spécifiez le modificateur d'accessibilité `private`. Aucun autre modificateur que `public` ou `private` n'est autorisé. Si `filename` est un fichier de ressources .NET créé, par exemple, par Resgen.exe ou dans l'environnement de développement, il est possible d'y accéder à l'aide des membres de l'espace de noms System.Resources. Pour plus d'informations, consultez System.Resources.ResourceManager. Pour toutes les autres ressources, utilisez les méthodes `GetManifestResource` dans la classe Assembly pour accéder à la ressource

au moment de l'exécution. Le fichier spécifié dans `filename` peut avoir n'importe quel format. C'est le cas, par exemple, si vous voulez qu'une DLL native fasse partie de l'assembly pour qu'elle puisse être installée dans le Global Assembly Cache et accessible à partir du code managé dans l'assembly. Vous pouvez effectuer la même opération dans Assembly Linker. Pour plus d'informations, consultez Al.exe (Assembly Linker) et Utilisation d'assemblys et du Global Assembly Cache.

Options diverses du compilateur C#

Article • 10/05/2023

Les options suivantes contrôlent divers comportements du compilateur. La nouvelle syntaxe MSBuild est affichée en **gras**. L'ancienne syntaxe de ligne de commande *csc.exe* est indiquée dans `code style`.

- **ResponseFiles** / `@CustomOpts.RSP` : lisez le fichier de réponse spécifié pour plus d'options.
- **NoLogo** / `-nologo` : supprimez le message de copyright du compilateur.
- **NoConfig** / `-noconfig` : n'incluez pas automatiquement le fichier *CSC.RSP*.

ResponseFiles

L'option **ResponseFiles** vous permet de spécifier un fichier qui contient les options du compilateur et les fichiers de code source à compiler.

```
XML

<ResponseFiles>response_file</ResponseFiles>
```

`response_file` spécifie le fichier qui répertorie des options du compilateur ou des fichiers de code source à compiler. Les options du compilateur et les fichiers de code source seront traités par le compilateur comme s'ils avaient été spécifiés sur la ligne de commande. Pour spécifier plusieurs fichiers réponse dans une compilation, spécifiez plusieurs options de fichier réponse. Dans un fichier réponse, plusieurs options du compilateur et fichiers de code source peuvent apparaître sur une même ligne. Une spécification d'option de compilateur unique doit apparaître sur une seule ligne (elle ne peut pas s'étendre sur plusieurs lignes). Les fichiers réponse peuvent contenir des commentaires, précédés du symbole #. La spécification d'options du compilateur à partir d'un fichier réponse est identique à la spécification de ces commandes sur la ligne de commande. Le compilateur traite les options de commande à mesure qu'elles sont lues. Les arguments de ligne de commande peuvent remplacer les options répertoriées précédemment dans les fichiers de réponse. Inversement, les options d'un fichier réponse substituent les options affichées précédemment dans la ligne de commande ou dans d'autres fichiers réponse. C# fournit le fichier csc.rsp, qui se trouve dans le même répertoire que le fichier csc.exe. Pour plus d'informations sur le format du fichier réponse, consultez NoConfig. Cette option du compilateur ne peut pas être définie dans l'environnement de développement Visual Studio, ni être modifiée par programmation. Voici quelques lignes extraites d'un exemple de fichier réponse :

Console

```
# build the first output file
-target:exe -out:MyExe.exe source1.cs source2.cs
```

NoLogo

L'option **NoLogo** supprime l'affichage de la bannière de démarrage pendant le démarrage du compilateur et l'affichage des messages d'informations pendant la compilation.

XML

```
<NoLogo>true</NoLogo>
```

NoConfig

L'option **NoConfig** indique au compilateur de ne pas compiler avec le fichier *csc.rsp*.

XML

```
<NoConfig>true</NoConfig>
```

Le fichier *csc.rsp* fait référence à tous les assemblys fournis avec le .NET Framework. Les références réelles qui sont incluses dans l'environnement de développement Visual Studio .NET varient en fonction du type de projet. Vous pouvez modifier le fichier *csc.rsp* et spécifier des options de compilateur supplémentaires qui doivent être incluses dans chaque compilation. Si vous ne voulez pas que le compilateur recherche et utilise les paramètres du fichier *csc.rsp*, spécifiez **NoConfig**. Cette option de compilateur n'est pas disponible dans Visual Studio et ne peut pas être changée par programmation.

Options du compilateur C# avancées

Article • 24/07/2023

Les options suivantes prennent en charge les scénarios avancés. La nouvelle syntaxe MSBuild est affichée en **gras**. L'ancienne syntaxe `csc.exe` s'affiche dans `code style`.

- **MainEntryPoint**, **StartupObject** / `-main` : spécifiez le type qui contient le point d'entrée.
- **PdbFile** / `-pdb` : spécifiez le nom du fichier d'informations de débogage.
- **PathMap** / `-pathmap` : spécifiez un mappage pour la sortie de noms de chemin d'accès source par le compilateur.
- **ApplicationConfiguration** / `-appconfig` : spécifiez un fichier de configuration d'application contenant des paramètres de liaison d'assembly.
- **AdditionalLibPaths** / `-lib` : spécifiez des répertoires supplémentaires dans lesquels rechercher les références.
- **GenerateFullPaths** / `-fullpath` : le compilateur génère des chemins d'accès complets.
- **PreferredUILang** / `-preferreduilang` : spécifiez le nom de la langue de sortie préférée.
- **BaseAddress** / `-baseaddress` : spécifiez l'adresse de base de la bibliothèque à générer.
- **ChecksumAlgorithm** / `-checksumalgorithm` : spécifiez l'algorithme de calcul de la somme de contrôle du fichier source stockée dans le fichier PDB.
- **CodePage** / `-codepage` : spécifiez la page de codes à utiliser à l'ouverture des fichiers sources.
- **Utf8Output** / `-utf8output` : messages du compilateur de sortie encodés en UTF-8.
- **FileAlignment** / `-filealign` : spécifier l'alignement utilisé pour les sections du fichier de sortie.
- **ErrorEndLocation** / `-errorendlocation` : ligne et colonne de sortie de l'emplacement de fin de chaque erreur.
- **NoStandardLib** / `-nostdlib` : ne référencez pas la bibliothèque standard *mscorlib.dll*.
- **SubsystemVersion** / `-subsystemversion` : spécifiez la version du sous-système de cet assembly.
- **ModuleAssemblyName** / `-moduleassemblyname` : Nom de l'assembly dont ce module doit faire partie.
- **ReportIVTs** / `-reportivts` : produire des informations supplémentaires sur System.Runtime.CompilerServices.InternalsVisibleToAttribute les informations.

Vous ajoutez l'une de ces options dans un `<PropertyGroup>` élément de votre `*.csproj` fichier :

```xml
<PropertyGroup>
    <StartupObject>...</StartupObject>
    ...
</PropertyGroup>
```

MainEntryPoint ou StartupObject

Cette option spécifie la classe qui contient le point d'entrée du programme dans le cas où plusieurs classes contiennent une méthode `Main`.

```xml
<StartupObject>MyNamespace.Program</StartupObject>
```

ou

```xml
<MainEntryPoint>MyNamespace.Program</MainEntryPoint>
```

Où `Program` est le type qui contient la méthode `Main`. Le nom de classe fourni doit être complet, c'est-à-dire qu'il doit inclure l'espace de noms complet contenant la classe, suivi du nom de classe. Par exemple, si la méthode `Main` se trouve dans la classe `Program` dans l'espace de noms `MyApplication.Core`, l'option du compilateur doit être `-main:MyApplication.Core.Program`. Si votre compilation inclut plusieurs types avec une méthode Main, vous pouvez spécifier le type qui contient la méthode `Main`.

> ⓘ **Notes**
>
> Cette option ne peut pas être utilisée pour un projet qui inclut des **instructions de niveau supérieur**, même si ce projet contient une ou plusieurs méthodes `Main`.

PdbFile

L'option de compilateur **PdbFile** spécifie le nom et l'emplacement du fichier de symboles de débogage. La valeur `filename` spécifie le nom et l'emplacement du fichier de symboles de débogage.

```
XML
```

```xml
<PdbFile>filename</PdbFile>
```

Lorsque vous spécifiez **DebugType**, le compilateur crée un fichier *.pdb* dans le même répertoire que celui du fichier de sortie créé par le compilateur (.exe ou .dll). Le fichier *.pdb* a le même nom de fichier de base que le nom du fichier de sortie. **PdbFile** vous permet de spécifier un nom de fichier autre que celui par défaut et un emplacement pour le fichier .pdb. Cette option du compilateur ne peut pas être définie dans l'environnement de développement Visual Studio, ni être modifiée par programmation.

PathMap

L'option de compilateur **PathMap** spécifie comment mapper des chemins physiques aux noms de chemin source générés en sortie par le compilateur. Cette option mappe chaque chemin physique de l'ordinateur sur lequel le compilateur s'exécute à un chemin correspondant qui doit être écrit dans les fichiers de sortie. Dans l'exemple suivant, `path1` est le chemin d'accès complet aux fichiers sources dans l'environnement actuel et `sourcePath1` est le chemin d'accès source remplacé par `path1` dans tous les fichiers de sortie. Pour spécifier plusieurs chemins sources mappés, séparez-les par des virgules.

```
XML
```

```xml
<PathMap>path1=sourcePath1,path2=sourcePath2</PathMap>
```

Le compilateur écrit le chemin source dans sa sortie dans les cas suivants :

1. Le chemin source est remplacé par un argument quand CallerFilePathAttribute est appliqué à un paramètre facultatif.
2. Le chemin source est incorporé dans un fichier PDB.
3. Le chemin du fichier PDB est incorporé dans un fichier exécutable portable (PE).

ApplicationConfiguration

L'option du compilateur **ApplicationConfiguration** permet à une application C# d'indiquer l'emplacement du fichier de configuration de l'application (app.config) d'un assembly au CLR (Common Language Runtime) au moment de la liaison de l'assembly.

XML

```xml
<ApplicationConfiguration>file</ApplicationConfiguration>
```

Où `file` est le fichier de configuration de l'application qui contient des paramètres de liaison de l'assembly. **ApplicationConfiguration** est notamment utilisé dans des scénarios avancés dans lesquels un assembly doit simultanément référencer à la fois la version du .NET Framework et la version du .NET Framework pour Silverlight d'un assembly de référence particulier. Par exemple, un concepteur XAML écrit dans WPF (Windows Presentation Foundation) devra peut-être référencer à la fois le bureau WPF pour l'interface utilisateur du concepteur et le sous-ensemble de WPF qui est fourni avec Silverlight. Le même assembly de concepteur doit accéder aux deux assemblys. Par défaut, les références séparées provoquent une erreur du compilateur parce que la liaison d'assembly considère les deux assemblys comme équivalents. L'option du compilateur **ApplicationConfiguration** vous permet de spécifier l'emplacement d'un fichier app.config qui désactive le comportement par défaut à l'aide d'une balise `<supportPortability>`, comme indiqué dans l'exemple suivant.

XML

```xml
<supportPortability PKT="7cec85d7bea7798e" enable="false"/>
```

Le compilateur passe l'emplacement du fichier à la logique de liaison d'assembly du CLR.

> **Notes**
>
> Pour utiliser le fichier app.config qui est déjà défini dans le projet, ajoutez la balise de propriété `<UseAppConfigForCompiler>` au fichier *.csproj* et affectez-lui la valeur `true`. Pour spécifier un autre fichier app.config, ajoutez la balise de propriété `<AppConfigForCompiler>` et définissez sa valeur sur l'emplacement du fichier.

L'exemple suivant affiche un fichier app.config qui permet à une application d'avoir des références à la fois à l'implémentation .NET Framework et à l'implémentation .NET Framework pour Silverlight de tout assembly .NET Framework qui existe dans les deux implémentations. L'option du compilateur **ApplicationConfiguration** spécifie l'emplacement de ce fichier app.config.

XML

```
<configuration>
  <runtime>
    <assemblyBinding>
      <supportPortability PKT="7cec85d7bea7798e" enable="false"/>
      <supportPortability PKT="31bf3856ad364e35" enable="false"/>
    </assemblyBinding>
  </runtime>
</configuration>
```

AdditionalLibPaths

L'option **AdditionalLibPaths** spécifie l'emplacement des assemblys référencés avec l'option **Références**.

XML

```
<AdditionalLibPaths>dir1[,dir2]</AdditionalLibPaths>
```

Où `dir1` est un répertoire dans lequel le compilateur doit effectuer la recherche si un assembly référencé ne se trouve pas dans le répertoire de travail actuel (le répertoire à partir duquel vous appelez le compilateur) ou dans le répertoire système du common language runtime. `dir2` est un ou plusieurs répertoires supplémentaires où rechercher des références d'assembly. Séparez les noms des répertoires par une virgule, sans espace. Le compilateur recherche les références d'assembly qui ne sont pas complètes dans l'ordre suivant :

1. Répertoire de travail actuel.
2. Répertoire système du common language runtime.
3. Répertoires spécifiés par **AdditionalLibPaths**.
4. Répertoires spécifiés par la variable d'environnement LIB.

Utilisez **Reference** pour spécifier une référence d'assembly. **AdditionalLibPaths** est additif. Si vous le spécifiez plusieurs fois, il effectue un ajout aux valeurs précédentes. Comme le chemin de l'assembly dépendant n'est pas spécifié dans le manifeste de l'assembly, l'application recherche et utilise l'assembly dans le Global Assembly Cache. Le compilateur référençant l'assembly n'implique pas que le Common Language Runtime puisse chercher et charger l'assembly au moment de l'exécution. Consultez Méthode de localisation des assemblys par le runtime pour plus d'informations sur la façon dont le runtime recherche des assemblys référencés.

GenerateFullPaths

Si vous utilisez l'option **GenerateFullPaths**, le compilateur spécifie le chemin complet du fichier quand il répertorie les erreurs de compilation et les avertissements.

```Xml
<GenerateFullPaths>true</GenerateFullPaths>
```

Par défaut, les erreurs et les avertissements générés par la compilation spécifient le nom du fichier dans lequel une erreur a été trouvée. Si vous utilisez l'option **GenerateFullPaths**, le compilateur spécifie le chemin complet du fichier. Cette option de compilateur n'est pas disponible dans Visual Studio et ne peut pas être changée par programmation.

PreferredUILang

L'option de compilateur **PreferredUILang** vous permet de spécifier la langue dans laquelle le compilateur C# affiche la sortie, comme les messages d'erreur.

```XML
<PreferredUILang>language</PreferredUILang>
```

Où `language` est le nom de la langue à utiliser pour la sortie du compilateur. Vous pouvez utiliser l'option de compilateur **PreferredUILang** pour spécifier la langue que le compilateur C# doit utiliser pour les messages d'erreur et d'autres types de sortie de ligne de commande. Si le module linguistique de la langue n'est pas installé, le paramètre de langue du système d'exploitation est utilisé.

BaseAddress

L'option **BaseAddress** vous permet de spécifier l'adresse de base préférée à laquelle doit être chargée une DLL. Pour plus d'informations sur le moment et la raison de l'utilisation de cette option, consultez Larry Osterman's WebLog.

```XML
<BaseAddress>address</BaseAddress>
```

Où `address` est l'adresse de base de la DLL. Cette adresse peut être spécifiée sous forme d'un nombre décimal, hexadécimal ou octal. L'adresse de base par défaut d'une DLL est définie par le common language runtime .NET. Le dernier chiffre de cette adresse sera

arrondi. Par exemple, si vous spécifiez l'adresse `0x11110001`, elle est arrondie à `0x11110000`. Pour terminer le processus de signature d'une DLL, utilisez SN.EXE avec l'option -R.

ChecksumAlgorithm

Cette option contrôle l'algorithme checksum que nous utilisons pour encoder les fichiers sources dans le fichier PDB.

```
XML

<ChecksumAlgorithm>algorithm</ChecksumAlgorithm>
```

`algorithm` doit être `SHA1` (par défaut) ou `SHA256`.

CodePage

Cette option spécifie la page de codes à utiliser pendant la compilation si la page exigée n'est pas la page de codes par défaut actuelle du système.

```
XML

<CodePage>id</CodePage>
```

Où `id` est l'identificateur de la page de codes à utiliser pour tous les fichiers de code source de la compilation. Le compilateur tente tout d'abord d'interpréter tous les fichiers sources comme des fichiers UTF-8. Si vos fichiers de code source sont encodés dans un format autre que UTF-8 et s'ils utilisent des caractères autres que des caractères ASCII 7 bits, utilisez l'option **CodePage** pour spécifier la page de codes à utiliser. **CodePage** s'applique à tous les fichiers de code source inclus dans votre compilation. Pour plus d'informations sur la façon de savoir quelles pages de codes sont prises en charge sur votre système, consultez GetCPInfo.

Utf8Output

L'option **Utf8Output** affiche la sortie du compilateur en utilisant un encodage UTF-8.

```
XML

<Utf8Output>true</Utf8Output>
```

Dans certaines configurations internationales, la sortie du compilateur ne peut pas s'afficher correctement dans la console. Utilisez **Utf8Output** et redirigez la sortie du compilateur vers un fichier.

FileAlignment

L'option **FileAlignment** permet de spécifier la taille des sections de votre fichier de sortie. Les valeurs valides sont 512, 1024, 2048, 4096 et 8192. Ces valeurs sont exprimées en octets.

```
XML

<FileAlignment>number</FileAlignment>
```

Vous définissez l'option **FileAlignment** à partir de la page **Avancé** des propriétés **Build** de votre projet dans Visual Studio. Chaque section est alignée sur une limite qui est un multiple de la valeur **FileAlignment**. Il n'existe aucune valeur fixe par défaut. Si la valeur **FileAlignment** n'est pas spécifiée, le Common Language Runtime choisit une valeur par défaut au moment de la compilation. En spécifiant la taille de la section, vous affectez la taille du fichier de sortie. Il peut être utile de modifier la taille de la section pour les programmes qui sont exécutés sur des appareils de petite taille. Utilisez DUMPBIN pour afficher des informations sur les sections de votre fichier de sortie.

ErrorEndLocation

Indique au compilateur de sortir la ligne et la colonne de l'emplacement de fin de chaque erreur.

```
XML

<ErrorEndLocation>true</ErrorEndLocation>
```

Par défaut, le compilateur écrit l'emplacement de départ dans la source pour toutes les erreurs et tous les avertissements. Lorsque cette option est définie sur true, le compilateur écrit à la fois l'emplacement de début et de fin pour chaque erreur et chaque avertissement.

NoStandardLib

NoStandardLib empêche l'importation du fichier mscorlib.dll, qui définit l'espace de noms System tout entier.

```XML
<NoStandardLib>true</NoStandardLib>
```

Utilisez cette option si vous souhaitez définir ou créer vos propres objets et espace de noms System. Si vous ne spécifiez pas **NoStandardLib**, mscorlib.dll est importé dans votre programme (ce qui équivaut à spécifier `<NoStandardLib>false</NoStandardLib>`).

SubsystemVersion

Spécifie la version minimale du sous-système sur lequel le fichier exécutable s'exécute. En règle générale, cette option garantit que le fichier exécutable peut utiliser des fonctionnalités de sécurité particulières qui ne sont pas disponibles avec des versions antérieures de Windows.

> ⓘ **Notes**
>
> Pour spécifier le sous-système lui-même, utilisez l'option du compilateur **TargetType**.

```XML
<SubsystemVersion>major.minor</SubsystemVersion>
```

`major.minor` spécifie la version minimale requise du sous-système, telle qu'elle est exprimée dans une notation par points pour les versions majeure et mineure. Par exemple, vous pouvez spécifier qu'une application ne peut pas s'exécuter sur un système d'exploitation antérieur à Windows 7. Définissez la valeur de cette option sur 6.01, comme décrit plus loi dans le tableau de cet article. Spécifiez les valeurs pour `major` et `minor` en tant qu'entiers. Les zéros à gauche du numéro de version `minor` ne modifient pas la version, contrairement aux zéros de droite. Par exemple, 6.1 et 6.01 font référence à la même version, mais 6.10 fait référence à une version différente. Nous vous recommandons d'exprimer la version mineure avec deux chiffres pour éviter toute confusion.

Le tableau suivant répertorie les versions courantes de sous-système de Windows.

Version de Windows	Version de sous-système
Windows Server 2003	5.02
Windows Vista	6,00
Windows 7	6.01
Windows Server 2008	6.01
Windows 8	6.02

La valeur par défaut de l'option du compilateur **SubsystemVersion** dépend des conditions répertoriées dans la liste suivante :

- La valeur par défaut est 6.02 si l'une quelconque des options du compilateur de la liste suivante est définie :
  - -target:appcontainerexe
  - -target:winmdobj
  - -platform:arm
- La valeur par défaut est 6.00 si vous utilisez MSBuild, si vous ciblez .NET Framework 4.5 et si vous n'avez défini aucune des options du compilateur spécifiées plus haut dans cette liste.
- La valeur par défaut est 4.00 si aucune des conditions précédentes n'est vraie.

ModuleAssemblyName

Spécifie le nom d'un assembly dont les types non publics sont accessibles par un *.netmodule*.

XML

```
<ModuleAssemblyName>assembly_name</ModuleAssemblyName>
```

ModuleAssemblyName doit être utilisée au moment de créer un fichier *.netmodule*et quand les conditions suivantes sont réunies :

- Le fichier *.netmodule* doit accéder à des types non publics dans un assembly existant.
- Vous connaissez le nom de l'assembly dans lequel le fichier .netmodule sera généré.
- L'assembly existant a accordé un accès d'assembly friend à l'assembly dans lequel le fichier *netmodule* sera généré.

Pour plus d'informations sur la création d'un fichier .netmodule, consultez l'option du **moduleTargetType**. Pour plus d'informations sur les assemblys friend, consultez Assemblys friend.

ReportIVTs

Activez ou désactivez les informations de diagnostic supplémentaires trouvées System.Runtime.CompilerServices.InternalsVisibleToAttribute pendant la compilation :

```XML
<ReportIVTs>true</ReportIVTs>
```

Les diagnostics sont activés si le contenu de l'élément est `true`, désactivé si `false`, ou non présent.

ReportIVTs signale les informations suivantes lorsqu'il est activé :

1. Tout membre inaccessible diagnostics inclure son assembly source, s'il est différent de l'assembly actuel.
2. Le compilateur imprime l'identité d'assembly du projet en cours de compilation, son nom d'assembly et sa clé publique.
3. Pour chaque référence passée au compilateur, elle s'imprime ;
   a. Identité d'assembly de la référence
   b. Si la référence accorde le projet actuel `InternalsVisibleTo`
   c. Nom et toutes les clés publiques de tous les assemblys accordés `InternalsVisibleTo` à partir de cet assembly

Commentaires sur la documentation XML

Article • 23/02/2023

Les fichiers sources C# peuvent avoir des commentaires structurés qui produisent une documentation d'API pour les types définis dans ces fichiers. Le compilateur C# produit un fichier *XML* qui contient des données structurées représentant les commentaires et les signatures d'API. D'autres outils peuvent traiter cette sortie XML pour créer une documentation lisible par l'homme sous la forme de pages web ou de fichiers PDF, par exemple.

Ce processus offre de nombreux avantages et vous permet d'ajouter de la documentation d'API dans votre code :

- Le compilateur C# combine la structure du code C# avec le texte des commentaires dans un document XML unique.
- Le compilateur C# vérifie que les commentaires correspondent aux signatures d'API pour les balises pertinentes.
- Les outils qui traitent les fichiers de documentation XML peuvent définir des éléments et des attributs XML spécifiques à ces outils.

Les outils tels que Visual Studio fournissent IntelliSense pour de nombreux éléments XML courants utilisés dans les commentaires de documentation.

Cet article traite des sujets suivants :

- Commentaires de documentation et génération de fichiers XML
- Balises validées par le compilateur C# et Visual Studio
- Format du fichier XML généré

Créer une sortie de documentation XML

Vous créez la documentation pour votre code en écrivant des champs de commentaire spéciaux indiqués par trois barres obliques. Les champs de commentaire incluent des éléments XML qui décrivent le bloc de code qui suit les commentaires. Par exemple :

C#

```
/// <summary>
///  This class performs an important function.
/// </summary>
public class MyClass {}
```

Vous définissez l'option **GenerateDocumentationFile** ou **DocumentationFile**, et le compilateur recherche tous les champs de commentaire avec des balises XML dans le code source et crée un fichier de documentation XML à partir de ces commentaires. Quand cette option est activée, le compilateur génère l'avertissement CS1591 pour tout membre publiquement visible déclaré dans votre projet sans commentaires de documentation XML.

Formats de commentaire XML

L'utilisation de commentaires de documentation XML exige des délimiteurs, qui indiquent où un commentaire de documentation commence et se termine. Vous utilisez les délimiteurs suivants avec les balises de documentation XML :

- Délimiteur à ligne unique `///` : les exemples de documentation et les modèles de projet C# utilisent cette forme. Si un espace blanc suit ce délimiteur, il n'est pas inclus dans la sortie XML.

  > ⓘ **Notes**
  >
  > Visual Studio insère automatiquement les balises `<summary>` et `</summary>`, et positionne votre curseur dans ces balises une fois que vous avez tapé le délimiteur `///` dans l'éditeur de code. Vous pouvez activer/désactiver cette fonctionnalité dans la **boîte de dialogue Options**.

- Délimiteurs multilignes `/** */` : les délimiteurs `/** */` ont les règles de mise en forme suivantes :

  - Sur la ligne qui contient le délimiteur `/**`, si le reste de la ligne est constitué d'espaces blancs, la ligne n'est pas traitée pour les commentaires. Si le premier caractère après le délimiteur `/**` est un espace blanc, celui-ci est ignoré et le reste de la ligne est traité. Sinon, tout le texte de la ligne après le délimiteur `/**` est traité comme faisant partie du commentaire.

  - Sur la ligne qui contient le délimiteur `*/`, s'il n'y a que des espaces blancs jusqu'au délimiteur `*/`, la ligne est ignorée. Sinon, le texte de la ligne jusqu'au délimiteur `*/` est traité comme faisant partie du commentaire.

  - Pour les lignes situées après celle qui commence par le délimiteur `/**`, le compilateur recherche un modèle commun au début de chaque ligne. Le modèle peut se composer d'un espace blanc facultatif et d'un astérisque (`*`),

suivi d'autres espaces blancs facultatifs. Si le compilateur détecte un modèle commun au début de chaque ligne qui ne commence pas par le délimiteur `/**` ou qui ne finit pas par le délimiteur `*/`, il ignore ce modèle pour chaque ligne.

- La seule partie du commentaire suivant qui sera traitée est la ligne qui commence par `<summary>`. Les trois formats de balise produisent les mêmes commentaires.

```
C#

/** <summary>text</summary> */

/**
<summary>text</summary>
*/

/**
* <summary>text</summary>
*/
```

- Le compilateur identifie un modèle commun « * » au début des deuxième et troisième lignes. Le modèle n'est pas inclus dans la sortie.

```
C#

/**
* <summary>
* text </summary>*/
```

- Le compilateur ne détecte aucun modèle commun dans le commentaire suivant, car le deuxième caractère sur la troisième ligne n'est pas un astérisque. Tout le texte des deuxième et troisième lignes est traité comme faisant partie du commentaire.

```
C#

/**
* <summary>
   text </summary>
*/
```

- Le compilateur ne détecte aucun modèle dans le commentaire suivant pour deux raisons. Tout d'abord, le nombre d'espaces avant l'astérisque n'est pas constant. Ensuite, la cinquième ligne commence par une tabulation, qui ne

correspond pas à des espaces. Tout le texte des lignes 2 à 5 est traité comme faisant partie du commentaire.

```
C#

/**
  * <summary>
  * text
*  text2
    *  </summary>
*/
```

Pour faire référence à des éléments XML (par exemple, votre fonction traite des éléments XML spécifiques que vous souhaitez décrire dans un commentaire de documentation XML), vous pouvez utiliser le mécanisme de citation standard (`<` et `>`). Pour faire référence aux identificateurs génériques dans les éléments de référence de code (`cref`), vous pouvez utiliser des caractères d'échappement (par exemple, `cref="List<T>"`) ou des accolades (`cref="List{T}"`). En tant que cas particulier, le compilateur analyse les accolades comme des crochets pointus pour rendre le commentaire de documentation moins fastidieux à créer lorsqu'il s'agit de faire référence aux identificateurs génériques.

> **Notes**
>
> Les commentaires de documentation XML ne sont pas des métadonnées. Ils ne sont pas inclus dans l'assembly compilé et ne sont donc pas accessibles par réflexion.

Outils qui acceptent l'entrée de documentation XML

Les outils suivants créent une sortie à partir de commentaires XML :

- DocFX : *DocFX* est un générateur de documentation d'API pour .NET, qui prend actuellement en charge C#, Visual Basic et F#. Il vous permet également de personnaliser la documentation de référence générée. DocFX génère un site web HTML statique à partir de votre code source et de vos fichiers Markdown. DocFX vous offre également la possibilité de personnaliser la mise en page et le style de votre site web à l'aide de modèles. Vous pouvez également créer des modèles personnalisés.

- Sandcastle : les *outils Sandcastle* créent des fichiers d'aide pour les bibliothèques de classes managées contenant à la fois des pages de référence sur les concepts et les API. Les outils Sandcastle sont basés sur la ligne de commande et ne disposent d'aucune interface utilisateur graphique frontale, d'aucune fonctionnalité de gestion de projet ni d'aucun processus de génération automatisé. Sandcastle Help File Builder fournit une interface utilisateur graphique autonome et des outils basés sur la ligne de commande pour créer un fichier d'aide de manière automatisée. Un package d'intégration Visual Studio est également disponible pour cela afin que les projets d'aide puissent être créés et gérés entièrement depuis Visual Studio.
- Doxygen : *Doxygen* génère un navigateur de documentation en ligne (en HTML) ou un manuel de référence hors ligne (dans LaTeX) à partir d'un ensemble de fichiers sources documentés. Il existe également une prise en charge de la génération d'une sortie dans les pages de manuel RTF (MS Word), PostScript, PDF avec lien hypertexte, HTML compressé, DocBook et Unix. Vous pouvez configurer Doxygen pour extraire la structure du code à partir de fichiers sources non documentés.

Chaînes d'ID

Chaque type ou membre est stocké dans un élément dans le fichier XML de sortie. Chacun de ces éléments a une chaîne d'ID unique qui identifie le type ou le membre. La chaîne d'ID doit prendre en compte les opérateurs, les paramètres, les valeurs de retour, les paramètres de type générique et les paramètres `ref`, `in` et `out`. Pour encoder tous ces éléments potentiels, le compilateur suit des règles clairement définies pour générer les chaînes d'ID. Les programmes qui traitent le fichier XML utilisent la chaîne d'ID pour identifier l'élément de métadonnées ou de réflexion .NET correspondant auquel s'applique la documentation.

Le compilateur respecte les règles suivantes quand il génère les chaînes d'ID :

- La chaîne ne contient aucun espace blanc.
- La première partie de cette chaîne identifie le type de membre avec un caractère unique suivi d'un signe deux-points. Les types de membres suivants sont utilisés :

Caractère	Type de membre	Notes
N	espace de noms	Vous ne pouvez pas ajouter de commentaires de documentation à un espace de noms, mais vous pouvez insérer des références cref à des commentaires, là où cela est pris en charge.

Caractère	Type de membre	Notes
T	type	Un type est une classe, une interface, un struct, une énumération ou un délégué.
F	field	
P	propriété	Inclut des indexeurs ou d'autres propriétés indexées.
M	method	Inclut des méthodes spéciales, telles que des constructeurs et des opérateurs.
E	événement	
!	chaîne d'erreur	Le reste de la chaîne fournit des informations sur l'erreur. Le compilateur C# génère des informations d'erreur pour les liens qui ne peuvent pas être résolus.

- La deuxième partie de la chaîne est le nom qualifié complet de l'élément, en commençant à la racine de l'espace de noms. Le nom de l'élément, ses types englobants et l'espace de noms sont séparés par des points. Si le nom de l'élément lui-même comporte des points, ceux-ci sont remplacés par un signe dièse (« # »). Il est supposé qu'aucun élément n'a de signe dièse directement dans son nom. Par exemple, le nom complet du constructeur String est « System.String.#ctor ».

- Pour les propriétés et les méthodes, la liste de paramètres entre parenthèses suit. S'il n'y a pas de paramètres, les parenthèses ne sont pas présentes. Les paramètres sont séparés par des virgules. L'encodage de chaque paramètre suit directement la façon dont il est encodé dans une signature .NET (consultez Microsoft.VisualStudio.CorDebugInterop.CorElementType pour obtenir les définitions des éléments en majuscules listés ci-dessous) :
  - Types de base. Les types réguliers (`ELEMENT_TYPE_CLASS` ou `ELEMENT_TYPE_VALUETYPE`) sont représentés en tant que nom complet du type.
  - Les types intrinsèques (par exemple, `ELEMENT_TYPE_I4`, `ELEMENT_TYPE_OBJECT`, `ELEMENT_TYPE_STRING`, `ELEMENT_TYPE_TYPEDBYREF` et `ELEMENT_TYPE_VOID`) sont représentés en tant que nom complet du type complet correspondant. Par exemple, `System.Int32` ou `System.TypedReference`.
  - `ELEMENT_TYPE_PTR` est représenté par un « * » après le type modifié.
  - `ELEMENT_TYPE_BYREF` est représenté par un « @ » après le type modifié.
  - `ELEMENT_TYPE_CMOD_OPT` est représenté par un « ! » et le nom complet de la classe de modificateur, après le type modifié.
  - `ELEMENT_TYPE_SZARRAY` est représenté par « [] » après le type d'élément du tableau.

    - `ELEMENT_TYPE_ARRAY` est représenté par [*limite_inférieure*:`size`,*limite_inférieure*:`size`], où le nombre de virgules correspond au rang – 1, et les limites inférieures et la taille de chaque dimension, si elles sont connues, sont représentées sous forme décimale. Si une limite inférieure ou une taille n'est pas spécifiée, elle est omise. Si la limite inférieure et la taille d'une dimension particulière sont omises, le « : » est également omis. Par exemple, un tableau à deux dimensions avec 1 comme limites inférieures et des tailles non spécifiées est représenté par [1:,1:].

- Pour les opérateurs de conversion uniquement (`op_Implicit` et `op_Explicit`), la valeur de retour de la méthode est encodée par un `~`, suivi du type de retour. Par exemple : `<member name="M:System.Decimal.op_Explicit(System.Decimal arg)~System.Int32">` est la balise de l'opérateur de cast `public static explicit operator int (decimal value);` déclaré dans la classe `System.Decimal`.

- Pour les types génériques, le nom du type est suivi d'un accent grave, puis d'un chiffre qui indique le nombre de paramètres de type générique. Par exemple : `<member name="T:SampleClass``2">` est la balise pour un type qui est défini en tant que `public class SampleClass<T, U>`. Pour les méthodes qui prennent des types génériques comme paramètres, les paramètres de types génériques sont spécifiés sous forme de nombres précédés d'accents graves (par exemple `0,`1). Chaque nombre représente une notation de tableau de base zéro pour les paramètres génériques du type.
    - `ELEMENT_TYPE_PINNED` est représenté par un « ^ » après le type modifié. Le compilateur C# ne génère jamais ce codage.
    - `ELEMENT_TYPE_CMOD_REQ` est représenté par un « | » et le nom complet de la classe de modificateur, après le type modifié. Le compilateur C# ne génère jamais ce codage.
    - `ELEMENT_TYPE_GENERICARRAY` est représenté par « [?] » après le type d'élément du tableau. Le compilateur C# ne génère jamais ce codage.
    - `ELEMENT_TYPE_FNPTR` est représenté par « =FUNC:`type`(*signature*) », où `type` est le type de retour et *signature* correspond aux arguments de la méthode. S'il n'y a pas d'argument, les parenthèses sont omises. Le compilateur C# ne génère jamais ce codage.
    - Les composants de signature suivants ne sont pas représentés, car ils ne sont pas utilisés pour différencier les méthodes surchargées :
        - convention d'appel
        - type de retour
        - `ELEMENT_TYPE_SENTINEL`

Les exemples suivants montrent comment les chaînes d'ID pour une classe et ses membres sont générées :

C#

```
namespace MyNamespace
{
    /// <summary>
    /// Enter description here for class X.
    /// ID string generated is "T:MyNamespace.MyClass".
    /// </summary>
    public unsafe class MyClass
    {
        /// <summary>
        /// Enter description here for the first constructor.
        /// ID string generated is "M:MyNamespace.MyClass.#ctor".
        /// </summary>
        public MyClass() { }

        /// <summary>
        /// Enter description here for the second constructor.
        /// ID string generated is
"M:MyNamespace.MyClass.#ctor(System.Int32)".
        /// </summary>
        /// <param name="i">Describe parameter.</param>
        public MyClass(int i) { }

        /// <summary>
        /// Enter description here for field message.
        /// ID string generated is "F:MyNamespace.MyClass.message".
        /// </summary>
        public string? message;

        /// <summary>
        /// Enter description for constant PI.
        /// ID string generated is "F:MyNamespace.MyClass.PI".
        /// </summary>
        public const double PI = 3.14;

        /// <summary>
        /// Enter description for method func.
        /// ID string generated is "M:MyNamespace.MyClass.func".
        /// </summary>
        /// <returns>Describe return value.</returns>
        public int func() { return 1; }

        /// <summary>
        /// Enter description for method someMethod.
        /// ID string generated is
"M:MyNamespace.MyClass.someMethod(System.String,System.Int32@,System.Void*)"
.
        /// </summary>
        /// <param name="str">Describe parameter.</param>
        /// <param name="num">Describe parameter.</param>
```

```
        /// <param name="ptr">Describe parameter.</param>
        /// <returns>Describe return value.</returns>
        public int someMethod(string str, ref int nm, void* ptr) { return 1;
}

        /// <summary>
        /// Enter description for method anotherMethod.
        /// ID string generated is
"M:MyNamespace.MyClass.anotherMethod(System.Int16[],System.Int32[0:,0:])".
        /// </summary>
        /// <param name="array1">Describe parameter.</param>
        /// <param name="array">Describe parameter.</param>
        /// <returns>Describe return value.</returns>
        public int anotherMethod(short[] array1, int[,] array) { return 0; }

        /// <summary>
        /// Enter description for operator.
        /// ID string generated is
"M:MyNamespace.MyClass.op_Addition(MyNamespace.MyClass,MyNamespace.MyClass)"
.
        /// </summary>
        /// <param name="first">Describe parameter.</param>
        /// <param name="second">Describe parameter.</param>
        /// <returns>Describe return value.</returns>
        public static MyClass operator +(MyClass first, MyClass second) {
return first; }

        /// <summary>
        /// Enter description for property.
        /// ID string generated is "P:MyNamespace.MyClass.prop".
        /// </summary>
        public int prop { get { return 1; } set { } }

        /// <summary>
        /// Enter description for event.
        /// ID string generated is "E:MyNamespace.MyClass.OnHappened".
        /// </summary>
        public event Del? OnHappened;

        /// <summary>
        /// Enter description for index.
        /// ID string generated is
"P:MyNamespace.MyClass.Item(System.String)".
        /// </summary>
        /// <param name="str">Describe parameter.</param>
        /// <returns></returns>
        public int this[string s] { get { return 1; } }

        /// <summary>
        /// Enter description for class Nested.
        /// ID string generated is "T:MyNamespace.MyClass.Nested".
        /// </summary>
        public class Nested { }

        /// <summary>
```

```
        /// Enter description for delegate.
        /// ID string generated is "T:MyNamespace.MyClass.Del".
        /// </summary>
        /// <param name="i">Describe parameter.</param>
        public delegate void Del(int i);

        /// <summary>
        /// Enter description for operator.
        /// ID string generated is
"M:MyNamespace.MyClass.op_Explicit(MyNamespace.MyClass)~System.Int32".
        /// </summary>
        /// <param name="myParameter">Describe parameter.</param>
        /// <returns>Describe return value.</returns>
        public static explicit operator int(MyClass myParameter) { return 1;
}
    }
}
```

spécification du langage C#

Pour plus d'informations, consultez l'annexe Spécification du langage C# sur les commentaires de documentation.

Balises XML recommandées pour les commentaires de documentation C#

Article • 10/05/2023

Les commentaires de documentation C# utilisent des éléments XML pour définir la structure de la documentation de sortie. L'une des conséquences de cette fonctionnalité est que vous pouvez ajouter n'importe quel code XML valide dans vos commentaires de documentation. Le compilateur C# copie ces éléments dans le fichier XML de sortie. Bien que vous puissiez utiliser n'importe quel code XML valide dans vos commentaires (y compris tout élément HTML valide), il est recommandé de documenter le code pour de nombreuses raisons.

Voici quelques recommandations, des scénarios de cas d'usage généraux et des éléments que vous devez connaître quand vous utilisez des balises de documentation XML dans votre code C#. Bien que vous puissiez placer n'importe quelles balises dans vos commentaires de documentation, cet article décrit les balises recommandées pour les constructions de langage les plus courantes. Dans tous les cas, vous devez respecter ces recommandations :

- Par souci de cohérence, tous les types visibles publiquement et leurs membres respectifs doivent être documentés.
- Les membres privés peuvent également être documentés à l'aide de commentaires XML. Toutefois, cela expose le fonctionnement interne (potentiellement confidentiel) de votre bibliothèque.
- Au minimum, les types et leurs membres doivent avoir une balise `<summary>`, car son contenu est nécessaire pour IntelliSense.
- Le texte de la documentation doit être écrit à l'aide de phrases complètes se terminant par un point.
- Les classes partielles sont entièrement prises en charge, et les informations de documentation seront concaténées dans une seule entrée pour chaque type.

Le début de la documentation XML est symbolisé par `///`. Lorsque vous créez un projet, les modèles insèrent quelques lignes de début `///` pour vous. Le traitement de ces commentaires présente certaines restrictions :

- La documentation doit être dans un format XML correct. Si le code XML n'est pas correctement formé, le compilateur génère un avertissement. Le fichier de documentation contiendra un commentaire indiquant qu'une erreur a été trouvée.
- Certaines des balises recommandées ont des significations spéciales :

    - La balise `<param>` est utilisée pour décrire les paramètres. Quand elle est utilisée, le compilateur vérifie que le paramètre existe et que tous les paramètres sont décrits dans la documentation. Si la vérification échoue, le compilateur émet un avertissement.
    - L'attribut `cref` peut être joint à n'importe quelle balise pour référencer un élément de code. Le compilateur vérifie l'existence de cet élément de code. Si la vérification échoue, le compilateur émet un avertissement. Le compilateur respecte toutes les instructions `using` lorsqu'il recherche un type décrit dans l'attribut `cref`.
    - La balise `<summary>` est utilisée par IntelliSense dans Visual Studio pour afficher des informations supplémentaires sur un type ou un membre.

      > ⓘ **Notes**
      >
      > Le fichier XML ne fournit pas des informations complètes sur le type et les membres (par exemple, il ne contient pas d'informations sur le type). Pour obtenir des informations complètes sur un type ou sur un membre, utilisez le fichier de documentation avec la réflexion sur le type ou sur le membre en question.

- Les développeurs sont libres de créer leur propre jeu de balises. Le compilateur copie ces informations dans le fichier de sortie.

Certaines des balises recommandées peuvent être utilisées sur n'importe quel élément de langage. D'autres ont une utilisation plus spécialisée. Enfin, certaines balises sont utilisées pour mettre en forme du texte dans votre documentation. Cet article décrit les balises recommandées organisées en fonction de leur utilisation.

Le compilateur vérifie la syntaxe des éléments suivis d'un seul * dans la liste suivante. Visual Studio fournit IntelliSense pour les balises vérifiées par le compilateur et toutes les balises suivies de ** dans la liste suivante. En plus des balises répertoriées ici, le compilateur et Visual Studio valident les balises `<b>`, `<i>`, `<u>`, `<br/>` et `<a>`. Le compilateur valide `<tt>` également, qui est du code HTML déconseillé.

- Balises générales utilisées pour plusieurs éléments : ces balises sont l'ensemble minimal pour n'importe quelle API.
    - <summary> : la valeur de cet élément s'affiche dans IntelliSense dans Visual Studio.
    - <remarks> **
- Balises utilisées pour les membres : ces balises sont utilisées lors de la documentation des méthodes et des propriétés.

    - <returns> : la valeur de cet élément s'affiche dans IntelliSense dans Visual Studio.
    - <param> * : la valeur de cet élément s'affiche dans IntelliSense dans Visual Studio.
    - <paramref>
    - <exception> *
    - <value> : la valeur de cet élément s'affiche dans IntelliSense dans Visual Studio.
- Mettre en forme la sortie de la documentation : ces balises fournissent des instructions de mise en forme pour les outils qui génèrent de la documentation.
    - <para>
    - <list>
    - <c>
    - <code>
    - <example> **
- Réutiliser le texte de la documentation : ces balises fournissent des outils qui facilitent la réutilisation des commentaires XML.
    - <inheritdoc> **
    - <include> *
- Générer des liens et des références : ces balises génèrent des liens vers d'autres documents.
    - <see> *
    - <seealso> *
    - cref
    - href
- Balises pour les types et méthodes génériques : ces balises sont utilisées uniquement sur les types et méthodes génériques
    - <typeparam> * : la valeur de cet élément s'affiche dans IntelliSense dans Visual Studio.
    - <typeparamref>

> ⓘ **Notes**
>
> Il n'est pas possible d'appliquer des commentaires de documentation à un espace de noms.

Si vous voulez que des crochets angulaires apparaissent dans le texte d'un commentaire de documentation, utilisez le codage HTML de `<` et `>`, respectivement `<` et `>`. Ce codage est illustré dans l'exemple suivant.

```
C#
```

```
/// <summary>
/// This property always returns a value &lt; 1.
/// </summary>
```

Balises générales

<summary>

XML

```
<summary>description</summary>
```

La balise `<summary>` doit être utilisée pour décrire un type ou un membre de type. Utilisez <remarks> pour ajouter des informations supplémentaires à une description de type. Utilisez l'attribut cref pour activer des outils de documentation, tels que DocFX et Sandcastle, afin de créer des liens hypertexte internes aux pages de documentation pour les éléments de code. Le texte de la balise `<summary>` est la seule source d'informations sur le type dans IntelliSense et il est également affiché dans la fenêtre Explorateur d'objets.

<remarks>

XML

```
<remarks>
description
</remarks>
```

La balise `<remarks>` permet d'ajouter des informations sur un type ou un membre de type en complétant les informations spécifiées par le <résumé>. Ces informations sont affichées dans la fenêtre Explorateur d'objets. Cette balise peut inclure des explications plus longues. Vous constaterez peut-être que l'utilisation de sections `CDATA` pour markdown rend l'écriture plus pratique. Des outils tels que docfx traitent le texte markdown dans les sections `CDATA`.

Membres du document

<returns>

XML

```xml
<returns>description</returns>
```

La balise `<returns>` doit être utilisée dans le commentaire relatif à une déclaration de méthode pour décrire la valeur renvoyée.

<param>

XML

```xml
<param name="name">description</param>
```

- `name` : nom d'un paramètre de méthode. Mettez le nom entre guillemets doubles (" "). Les noms des paramètres doivent correspondre à la signature d'API. Si un ou plusieurs paramètres ne sont pas couverts, le compilateur émet un avertissement. Le compilateur émet également un avertissement si la valeur de `name` ne correspond pas à un paramètre formel dans la déclaration de méthode.

La balise `<param>` doit être utilisée dans le commentaire d'une déclaration de méthode pour décrire l'un des paramètres de la méthode. Pour documenter plusieurs paramètres, utilisez plusieurs balises `<param>`. Le texte de la balise `<param>` s'affiche dans IntelliSense, dans la fenêtre Explorateur d'objets et dans le rapport web de commentaire de code.

<paramref>

XML

```xml
<paramref name="name"/>
```

- `name` : nom du paramètre auquel faire référence. Mettez le nom entre guillemets doubles (" ").

La balise `<paramref>` vous offre un moyen d'indiquer qu'un mot dans les commentaires du code, par exemple dans un bloc `<summary>` ou `<remarks>`, fait référence à un paramètre. Le fichier XML peut être traité de manière à mettre en forme ce mot d'une façon particulière, par exemple en gras ou en italique.

<exception>

XML

```xml
<exception cref="member">description</exception>
```

- cref = « `member` » : référence à une exception qui est disponible à partir de l'environnement de compilation actuel. Le compilateur vérifie que l'exception donnée existe, et il traduit `member` en nom d'élément canonique dans le fichier XML de sortie. `member` doit apparaître entre guillemets doubles (" ").

La balise `<exception>` vous permet de spécifier quelles exceptions peuvent être levées. Cette balise peut être appliquée à des définitions de méthodes, de propriétés, d'événements et d'indexeurs.

<value>

XML

```xml
<value>property-description</value>
```

La balise `<value>` vous permet de décrire la valeur représentée par une propriété. Lorsque vous ajoutez une propriété via l'assistant de code de l'environnement de développement Visual Studio .NET, il ajoute une balise <summary> pour la nouvelle propriété. Vous ajoutez manuellement une balise `<value>` pour décrire la valeur représentée par la propriété.

Mettre en forme la sortie de la documentation

<para>

XML

```xml
<remarks>
    <para>
        This is an introductory paragraph.
    </para>
    <para>
        This paragraph contains more details.
    </para>
</remarks>
```

La balise `<para>` est prévue pour une utilisation à l'intérieur d'une balise, telle que `<summary>`, `<remarks>` ou `<returns>`, et vous permet d'ajouter une structure au texte. La balise `<para>` crée un paragraphe avec interligne double. Utilisez la balise `<br/>` si vous souhaitez un paragraphe avec une seule interligne.

<list>

XML

```
<list type="bullet|number|table">
    <listheader>
        <term>term</term>
        <description>description</description>
    </listheader>
    <item>
        <term>Assembly</term>
        <description>The library or executable built from a compilation.
</description>
    </item>
</list>
```

Le bloc `<listheader>` est utilisé pour définir la ligne d'en-tête d'une table ou d'une liste de définitions. Au moment de définir une table, il vous suffit de fournir une entrée pour `term` figurant dans l'en-tête. Chaque élément de la liste est spécifié avec un bloc `<item>`. Au moment de créer une liste de définitions, vous devez spécifier à la fois `term` et `description`. Cependant, pour une table, une liste à puces ou une liste numérotée, il vous suffit de fournir une entrée pour `description`. Une liste ou une table peut comporter autant de blocs `<item>` que nécessaire.

<c>

XML

```
<c>text</c>
```

La balise `<c>` vous permet d'indiquer que le texte d'une description doit être marqué comme étant du code. Utilisez `<code>` pour indiquer plusieurs lignes comme étant du code.

<code>

XML

```
<code>
    var index = 5;
    index++;
</code>
```

La balise `<code>` est utilisée pour indiquer plusieurs lignes de code. Utilisez `<c>` pour indiquer qu'une ligne de texte d'une description doit être marquée comme étant du code.

<example>

XML

```
<example>
This shows how to increment an integer.
<code>
    var index = 5;
    index++;
</code>
</example>
```

La balise `<example>` vous permet de spécifier un exemple d'utilisation d'une méthode ou de tout autre membre de la bibliothèque. Cela implique généralement l'utilisation de la balise `<code>`.

Réutiliser le texte de la documentation

<inheritdoc>

XML

```
<inheritdoc [cref=""] [path=""]/>
```

Héritez des commentaires XML des classes de base, des interfaces et de méthodes similaires. Utiliser `inheritdoc` élimine les actions de copier-coller indésirables des commentaires XML en double et maintient automatiquement les commentaires XML synchronisés. Notez que lorsque vous ajoutez la balise `<inheritdoc>` à un type, tous les membres héritent également des commentaires.

- `cref` : spécifie le membre duquel hériter la documentation. Les balises déjà définies sur le membre actuel ne sont pas remplacées par les balises héritées.

- `path` : requête d'expression XPath qui aboutira à un ensemble de nœuds à afficher. Vous pouvez utiliser cet attribut pour filtrer les balises à inclure ou exclure de la documentation héritée.

Ajoutez vos commentaires XML dans des classes de base ou des interfaces et laissez inheritdoc copier les commentaires dans les classes d'implémentation. Ajoutez vos commentaires XML à vos méthodes synchrones et laissez inheritdoc copier les commentaires dans vos versions asynchrones des mêmes méthodes. Si vous souhaitez copier les commentaires d'un membre spécifique, vous utilisez l'attribut `cref` pour spécifier le membre.

<include>

XML

```
<include file='filename' path='tagpath[@name="id"]' />
```

- `filename` : nom du fichier XML contenant la documentation. Le nom de fichier peut être qualifié avec un chemin relatif au fichier de code source. Mettez `filename` entre guillemets simples (' ').
- `tagpath` : chemin des balises contenues dans `filename` qui mène à la balise `name`. Mettez le chemin entre guillemets simples (' ').
- `name` : spécificateur de nom contenu dans la balise qui précède les commentaires. `name` possède un `id`.
- `id` : ID de la balise qui précède les commentaires. Mettez l'ID entre guillemets doubles (" ").

La balise `<include>` vous permet de faire référence à des commentaires dans un autre fichier qui décrivent les types et les membres dans votre code source. Inclure un fichier externe constitue une solution alternative au placement direct des commentaires de la documentation dans votre fichier de code source. En plaçant la documentation dans un fichier distinct, vous pouvez appliquer un contrôle de code source à la documentation indépendamment du code source. Ainsi, une personne peut extraire le fichier de code source et une autre personne peut extraire le fichier de documentation. La balise `<include>` utilise la syntaxe XPath XML. Reportez-vous à la documentation de XPath pour savoir comment personnaliser votre utilisation de `<include>`.

Générer des liens et des références

<see>

XML

```xml
<see cref="member"/>
<!-- or -->
<see cref="member">Link text</see>
<!-- or -->
<see href="link">Link Text</see>
<!-- or -->
<see langword="keyword"/>
```

- `cref="member"` : référence à un membre ou un champ qu'il est possible d'appeler à partir de l'environnement de compilation actuel. Le compilateur vérifie que l'élément de code donné existe, et qu'il passe `member` au nom d'élément dans le code XML de sortie. Placez le *membre* entre guillemets doubles (" "). Vous pouvez fournir un texte de lien différent pour un « cref », à l'aide d'une balise fermante distincte.
- `href="link"` : lien cliquable vers une URL donnée. Par exemple, `<see href="https://github.com">GitHub</see>` produit un lien cliquable avec du texte GitHub qui mène à `https://github.com`.
- `langword="keyword"` : mot clé de langage, tel que `true` ou l'un des autres mots clés valides.

La balise `<see>` vous permet de spécifier un lien à partir de l'intérieur du texte. Utilisez <seealso> pour indiquer que le texte doit être placé dans une section Voir aussi. Utilisez l'attribut cref pour créer des liens hypertexte internes aux pages de documentation pour les éléments de code. Vous incluez les paramètres de type pour spécifier une référence à un type ou à une méthode générique, comme `cref="IDictionary{T, U}"`. `href` est également un attribut valide qui fonctionnera comme un lien hypertexte.

<seealso>

XML

```xml
<seealso cref="member"/>
<!-- or -->
<seealso href="link">Link Text</seealso>
```

- `cref="member"` : référence à un membre ou un champ qu'il est possible d'appeler à partir de l'environnement de compilation actuel. Le compilateur vérifie que

l'élément de code donné existe, et qu'il passe `member` au nom d'élément dans le code XML de sortie. `member` doit apparaître entre guillemets doubles (" ").

- `href="link"` : lien cliquable vers une URL donnée. Par exemple, `<seealso href="https://github.com">GitHub</seealso>` produit un lien cliquable avec du texte GitHub qui mène à `https://github.com`.

La balise `<seealso>` vous permet de spécifier le texte que vous souhaitez voir apparaître dans une section **Voir aussi**. Utilisez <see> pour spécifier un lien à partir de l'intérieur du texte. Vous ne pouvez pas imbriquer la balise `seealso` à l'intérieur de la balise `summary`.

cref, attribut

L'attribut `cref` dans une balise de documentation XML signifie « référence de code ». Il spécifie que le texte interne de la balise est un élément de code, tel qu'un type, une méthode ou une propriété. Les outils de documentation comme DocFX et Sandcastle utilisent les attributs `cref` pour générer automatiquement des liens hypertexte vers la page où le type ou le membre est documenté.

Attribut href

L'attribut `href` signifie une référence à une page web. Vous pouvez l'utiliser pour référencer directement la documentation en ligne sur votre API ou votre bibliothèque.

Types et méthodes génériques

<typeparam>

```xml
<typeparam name="TResult">The type returned from this method</typeparam>
```

- `TResult` : nom du paramètre de type. Mettez le nom entre guillemets doubles (" ").

La balise `<typeparam>` doit être utilisée dans le commentaire d'une déclaration de méthode ou de type générique pour décrire un paramètre de type. Ajoutez une balise pour chaque paramètre de type de la méthode et du type générique. Le texte de la balise `<typeparam>` s'affiche dans IntelliSense.

<typeparamref>

```xml
<typeparamref name="TKey"/>
```

- `TKey` : nom du paramètre de type. Mettez le nom entre guillemets doubles (" ").

Utilisez cette balise pour permettre aux consommateurs du fichier de documentation d'appliquer une mise en forme particulière au mot, par exemple l'italique.

Balises définies par l'utilisateur

Toutes les balises décrites ci-dessus représentent celles qui sont reconnues par le compilateur C#. Toutefois, un utilisateur est libre de définir ses propres balises. Des outils tels que Sandcastle permettent de prendre en charge des balises supplémentaires telles que <event> et <note>, et prennent même en charge la documentation des espaces de noms. Des outils de génération de documentation personnalisés ou internes peuvent également être utilisés avec les balises standard, et plusieurs formats de sortie, du format HTML au format PDF, peuvent être pris en charge.

Exemples de commentaires de documentation XML

Article • 28/03/2023

Cet article contient trois exemples d'ajout de commentaires de documentation XML à la plupart des éléments de langage C#. Le premier exemple montre comment documenter une classe avec différents membres. Le deuxième montre comment réutiliser des explications pour une hiérarchie de classes ou d'interfaces. Le troisième montre les balises à utiliser pour les classes et les membres génériques. Les deuxième et troisième exemples utilisent des concepts abordés dans le premier exemple.

Documenter une classe, une structure ou une interface

L'exemple suivant montre les éléments de langage courants et les balises que vous utiliserez probablement pour décrire ces éléments. Les commentaires de documentation décrivent l'utilisation des balises, plutôt que la classe elle-même.

C#

```
    /// <summary>
    /// Every class and member should have a one sentence
    /// summary describing its purpose.
    /// </summary>
    /// <remarks>
    /// You can expand on that one sentence summary to
    /// provide more information for readers. In this case,
    /// the <c>ExampleClass</c> provides different C#
    /// elements to show how you would add documentation
    ///comments for most elements in a typical class.
    /// <para>
    /// The remarks can add multiple paragraphs, so you can
    /// write detailed information for developers that use
    /// your work. You should add everything needed for
    /// readers to be successful. This class contains
    /// examples for the following:
    /// </para>
    /// <list type="table">
    /// <item>
    /// <term>Summary</term>
    /// <description>
    /// This should provide a one sentence summary of the class or member.
    /// </description>
    /// </item>
    /// <item>
```

```
    /// <term>Remarks</term>
    /// <description>
    /// This is typically a more detailed description of the class or member
    /// </description>
    /// </item>
    /// <item>
    /// <term>para</term>
    /// <description>
    /// The para tag separates a section into multiple paragraphs
    /// </description>
    /// </item>
    /// <item>
    /// <term>list</term>
    /// <description>
    /// Provides a list of terms or elements
    /// </description>
    /// </item>
    /// <item>
    /// <term>returns, param</term>
    /// <description>
    /// Used to describe parameters and return values
    /// </description>
    /// </item>
    /// <item>
    /// <term>value</term>
    /// <description>Used to describe properties</description>
    /// </item>
    /// <item>
    /// <term>exception</term>
    /// <description>
    /// Used to describe exceptions that may be thrown
    /// </description>
    /// </item>
    /// <item>
    /// <term>c, cref, see, seealso</term>
    /// <description>
    /// These provide code style and links to other
    /// documentation elements
    /// </description>
    /// </item>
    /// <item>
    /// <term>example, code</term>
    /// <description>
    /// These are used for code examples
    /// </description>
    /// </item>
    /// </list>
    /// <para>
    /// The list above uses the "table" style. You could
    /// also use the "bullet" or "number" style. Neither
    /// would typically use the "term" element.
    /// <br/>
    /// Note: paragraphs are double spaced. Use the *br*
    /// tag for single spaced lines.
    /// </para>
```

```
    /// </remarks>
    public class ExampleClass
    {
        /// <value>
        /// The <c>Label</c> property represents a label
        /// for this instance.
        /// </value>
        /// <remarks>
        /// The <see cref="Label"/> is a <see langword="string"/>
        /// that you use for a label.
        /// <para>
        /// Note that there isn't a way to provide a "cref" to
        /// each accessor, only to the property itself.
        /// </para>
        /// </remarks>
        public string? Label
        {
            get;
            set;
        }

        /// <summary>
        /// Adds two integers and returns the result.
        /// </summary>
        /// <returns>
        /// The sum of two integers.
        /// </returns>
        /// <param name="left">
        /// The left operand of the addition.
        /// </param>
        /// <param name="right">
        /// The right operand of the addition.
        /// </param>
        /// <example>
        /// <code>
        /// int c = Math.Add(4, 5);
        /// if (c > 10)
        /// {
        ///     Console.WriteLine(c);
        /// }
        /// </code>
        /// </example>
        /// <exception cref="System.OverflowException">
        /// Thrown when one parameter is
        /// <see cref="Int32.MaxValue">MaxValue</see> and the other is
        /// greater than 0.
        /// Note that here you can also use
        /// <see
href="https://learn.microsoft.com/dotnet/api/system.int32.maxvalue"/>
        ///  to point a web page instead.
        /// </exception>
        /// <see cref="ExampleClass"/> for a list of all
        /// the tags in these examples.
        /// <seealso cref="ExampleClass.Label"/>
        public static int Add(int left, int right)
```

```
        {
            if ((left == int.MaxValue && right > 0) || (right ==
int.MaxValue && left > 0))
                throw new System.OverflowException();

            return left + right;
        }
    }

    /// <summary>
    /// This is an example of a positional record.
    /// </summary>
    /// <remarks>
    /// There isn't a way to add XML comments for properties
    /// created for positional records, yet. The language
    /// design team is still considering what tags should
    /// be supported, and where. Currently, you can use
    /// the "param" tag to describe the parameters to the
    /// primary constructor.
    /// </remarks>
    /// <param name="FirstName">
    /// This tag will apply to the primary constructor parameter.
    /// </param>
    /// <param name="LastName">
    /// This tag will apply to the primary constructor parameter.
    /// </param>
    public record Person(string FirstName, string LastName);
}
```

L'ajout de documentation peut encombrer votre code source avec de grands ensembles de commentaires destinés aux utilisateurs de votre bibliothèque. Vous utilisez la balise `<Include>` pour séparer vos commentaires XML de votre source. Votre code source fait référence à un fichier XML avec la balise `<Include>` :

C#

```
/// <include file='xml_include_tag.xml'
path='MyDocs/MyMembers[@name="test"]/*' />
class Test
{
    static void Main()
    {
    }
}

/// <include file='xml_include_tag.xml'
path='MyDocs/MyMembers[@name="test2"]/*' />
class Test2
{
    public void Test()
    {
```

```
    }
}
```

Le deuxième fichier (*xml_include_tag.xml*) contient les commentaires de documentation.

XML

```
<MyDocs>
    <MyMembers name="test">
        <summary>
        The summary for this type.
        </summary>
    </MyMembers>
    <MyMembers name="test2">
        <summary>
        The summary for this other type.
        </summary>
    </MyMembers>
</MyDocs>
```

Documenter une hiérarchie de classes et d'interfaces

L'élément `<inheritdoc>` signifie qu'un type ou un membre *hérite* des commentaires de documentation d'une classe ou d'une interface de base. Vous pouvez également utiliser l'élément `<inheritdoc>` avec l'attribut `cref` pour faire hériter les commentaires d'un membre du même type. L'exemple suivant montre comment utiliser cette balise. Notez que lorsque vous ajoutez l'attribut `inheritdoc` à un type, les commentaires des membres sont hérités. Vous pouvez empêcher l'utilisation de commentaires hérités en écrivant des commentaires sur les membres dans le type dérivé. Ceux-ci seront préférés aux commentaires hérités.

C#

```
/// <summary>
/// A summary about this class.
/// </summary>
/// <remarks>
/// These remarks would explain more about this class.
/// In this example, these comments also explain the
/// general information about the derived class.
/// </remarks>
public class MainClass
{
}
```

```
///<inheritdoc/>
public class DerivedClass : MainClass
{
}

/// <summary>
/// This interface would describe all the methods in
/// its contract.
/// </summary>
/// <remarks>
/// While elided for brevity, each method or property
/// in this interface would contain docs that you want
/// to duplicate in each implementing class.
/// </remarks>
public interface ITestInterface
{
    /// <summary>
    /// This method is part of the test interface.
    /// </summary>
    /// <remarks>
    /// This content would be inherited by classes
    /// that implement this interface when the
    /// implementing class uses "inheritdoc"
    /// </remarks>
    /// <returns>The value of <paramref name="arg" /> </returns>
    /// <param name="arg">The argument to the method</param>
    int Method(int arg);
}

///<inheritdoc cref="ITestInterface"/>
public class ImplementingClass : ITestInterface
{
    // doc comments are inherited here.
    public int Method(int arg) => arg;
}

/// <summary>
/// This class shows hows you can "inherit" the doc
/// comments from one method in another method.
/// </summary>
/// <remarks>
/// You can inherit all comments, or only a specific tag,
/// represented by an xpath expression.
/// </remarks>
public class InheritOnlyReturns
{
    /// <summary>
    /// In this example, this summary is only visible for this method.
    /// </summary>
    /// <returns>A boolean</returns>
    public static bool MyParentMethod(bool x) { return x; }

    /// <inheritdoc cref="MyParentMethod" path="/returns"/>
    public static bool MyChildMethod() { return false; }
}
```

```
/// <Summary>
/// This class shows an example ofsharing comments across methods.
/// </Summary>
public class InheritAllButRemarks
{
    /// <summary>
    /// In this example, this summary is visible on all the methods.
    /// </summary>
    /// <remarks>
    /// The remarks can be inherited by other methods
    /// using the xpath expression.
    /// </remarks>
    /// <returns>A boolean</returns>
    public static bool MyParentMethod(bool x) { return x; }

    /// <inheritdoc cref="MyParentMethod" path="//*[not(self::remarks)]"/>
    public static bool MyChildMethod() { return false; }
}
```

Types génériques

Utilisez la balise `<typeparam>` pour décrire les paramètres de type sur les types et méthodes génériques. La valeur de l'attribut `cref` nécessite une nouvelle syntaxe pour référencer une méthode ou classe générique :

C#

```
/// <summary>
/// This is a generic class.
/// </summary>
/// <remarks>
/// This example shows how to specify the <see cref="GenericClass{T}"/>
/// type as a cref attribute.
/// In generic classes and methods, you'll often want to reference the
/// generic type, or the type parameter.
/// </remarks>
class GenericClass<T>
{
    // Fields and members.
}

/// <Summary>
/// This shows examples of typeparamref and typeparam tags
/// </Summary>
public class ParamsAndParamRefs
{
    /// <summary>
    /// The GetGenericValue method.
    /// </summary>
    /// <remarks>
```

```
    /// This sample shows how to specify the <see cref="GetGenericValue"/>
    /// method as a cref attribute.
    /// The parameter and return value are both of an arbitrary type,
    /// <typeparamref name="T"/>
    /// </remarks>
    public static T GetGenericValue<T>(T para)
    {
        return para;
    }
}
```

Exemple de classe mathématique

Le code suivant montre un exemple réaliste d'ajout de commentaires de document à une bibliothèque mathématique.

C#

```
namespace TaggedLibrary
{
    /*
        The main Math class
        Contains all methods for performing basic math functions
    */
        /// <summary>
    /// The main <c>Math</c> class.
    /// Contains all methods for performing basic math functions.
    /// <list type="bullet">
    /// <item>
    /// <term>Add</term>
    /// <description>Addition Operation</description>
    /// </item>
    /// <item>
    /// <term>Subtract</term>
    /// <description>Subtraction Operation</description>
    /// </item>
    /// <item>
    /// <term>Multiply</term>
    /// <description>Multiplication Operation</description>
    /// </item>
    /// <item>
    /// <term>Divide</term>
    /// <description>Division Operation</description>
    /// </item>
    /// </list>
    /// </summary>
    /// <remarks>
    /// <para>
    /// This class can add, subtract, multiply and divide.
    /// </para>
    /// <para>
```

```
    /// These operations can be performed on both
    /// integers and doubles.
    /// </para>
    /// </remarks>
    public class Math
    {
        // Adds two integers and returns the result
        /// <summary>
        /// Adds two integers <paramref name="a"/> and <paramref name="b"/>
        ///  and returns the result.
        /// </summary>
        /// <returns>
        /// The sum of two integers.
        /// </returns>
        /// <example>
        /// <code>
        /// int c = Math.Add(4, 5);
        /// if (c > 10)
        /// {
        ///     Console.WriteLine(c);
        /// }
        /// </code>
        /// </example>
        /// <exception cref="System.OverflowException">
        /// Thrown when one parameter is <see cref="Int32.MaxValue"/> and
the other
        /// is greater than 0.
        /// </exception>
        /// See <see cref="Math.Add(double, double)"/> to add doubles.
        /// <seealso cref="Math.Subtract(int, int)"/>
        /// <seealso cref="Math.Multiply(int, int)"/>
        /// <seealso cref="Math.Divide(int, int)"/>
        /// <param name="a">An integer.</param>
        /// <param name="b">An integer.</param>
        public static int Add(int a, int b)
        {
            // If any parameter is equal to the max value of an integer
            // and the other is greater than zero
            if ((a == int.MaxValue && b > 0) ||
                (b == int.MaxValue && a > 0))
            {
                throw new System.OverflowException();
            }
            return a + b;
        }

        // Adds two doubles and returns the result
        /// <summary>
        /// Adds two doubles <paramref name="a"/> and <paramref name="b"/>
        /// and returns the result.
        /// </summary>
        /// <returns>
        /// The sum of two doubles.
        /// </returns>
        /// <example>
```

```
        /// <code>
        /// double c = Math.Add(4.5, 5.4);
        /// if (c > 10)
        /// {
        ///     Console.WriteLine(c);
        /// }
        /// </code>
        /// </example>
        /// <exception cref="System.OverflowException">
        /// Thrown when one parameter is max and the other
        /// is greater than 0.</exception>
        /// See <see cref="Math.Add(int, int)"/> to add integers.
        /// <seealso cref="Math.Subtract(double, double)"/>
        /// <seealso cref="Math.Multiply(double, double)"/>
        /// <seealso cref="Math.Divide(double, double)"/>
        /// <param name="a">A double precision number.</param>
        /// <param name="b">A double precision number.</param>
        public static double Add(double a, double b)
        {
            // If any parameter is equal to the max value of an integer
            // and the other is greater than zero
            if ((a == double.MaxValue && b > 0)
                || (b == double.MaxValue && a > 0))
            {
                throw new System.OverflowException();
            }

            return a + b;
        }

        // Subtracts an integer from another and returns the result
        /// <summary>
        /// Subtracts <paramref name="b"/> from <paramref name="a"/>
        /// and returns the result.
        /// </summary>
        /// <returns>
        /// The difference between two integers.
        /// </returns>
        /// <example>
        /// <code>
        /// int c = Math.Subtract(4, 5);
        /// if (c > 1)
        /// {
        ///     Console.WriteLine(c);
        /// }
        /// </code>
        /// </example>
        /// See <see cref="Math.Subtract(double, double)"/> to subtract
doubles.
        /// <seealso cref="Math.Add(int, int)"/>
        /// <seealso cref="Math.Multiply(int, int)"/>
        /// <seealso cref="Math.Divide(int, int)"/>
        /// <param name="a">An integer.</param>
        /// <param name="b">An integer.</param>
        public static int Subtract(int a, int b)
```

```
        {
            return a - b;
        }

        // Subtracts a double from another and returns the result
        /// <summary>
        /// Subtracts a double <paramref name="b"/> from another
        /// double <paramref name="a"/> and returns the result.
        /// </summary>
        /// <returns>
        /// The difference between two doubles.
        /// </returns>
        /// <example>
        /// <code>
        /// double c = Math.Subtract(4.5, 5.4);
        /// if (c > 1)
        /// {
        ///     Console.WriteLine(c);
        /// }
        /// </code>
        /// </example>
        /// See <see cref="Math.Subtract(int, int)"/> to subtract integers.
        /// <seealso cref="Math.Add(double, double)"/>
        /// <seealso cref="Math.Multiply(double, double)"/>
        /// <seealso cref="Math.Divide(double, double)"/>
        /// <param name="a">A double precision number.</param>
        /// <param name="b">A double precision number.</param>
        public static double Subtract(double a, double b)
        {
            return a - b;
        }

        // Multiplies two integers and returns the result
        /// <summary>
        /// Multiplies two integers <paramref name="a"/>
        /// and <paramref name="b"/> and returns the result.
        /// </summary>
        /// <returns>
        /// The product of two integers.
        /// </returns>
        /// <example>
        /// <code>
        /// int c = Math.Multiply(4, 5);
        /// if (c > 100)
        /// {
        ///     Console.WriteLine(c);
        /// }
        /// </code>
        /// </example>
        /// See <see cref="Math.Multiply(double, double)"/> to multiply
doubles.
        /// <seealso cref="Math.Add(int, int)"/>
        /// <seealso cref="Math.Subtract(int, int)"/>
        /// <seealso cref="Math.Divide(int, int)"/>
        /// <param name="a">An integer.</param>
```

```
        /// <param name="b">An integer.</param>
        public static int Multiply(int a, int b)
        {
            return a * b;
        }

        // Multiplies two doubles and returns the result
        /// <summary>
        /// Multiplies two doubles <paramref name="a"/> and
        /// <paramref name="b"/> and returns the result.
        /// </summary>
        /// <returns>
        /// The product of two doubles.
        /// </returns>
        /// <example>
        /// <code>
        /// double c = Math.Multiply(4.5, 5.4);
        /// if (c > 100.0)
        /// {
        ///     Console.WriteLine(c);
        /// }
        /// </code>
        /// </example>
        /// See <see cref="Math.Multiply(int, int)"/> to multiply integers.
        /// <seealso cref="Math.Add(double, double)"/>
        /// <seealso cref="Math.Subtract(double, double)"/>
        /// <seealso cref="Math.Divide(double, double)"/>
        /// <param name="a">A double precision number.</param>
        /// <param name="b">A double precision number.</param>
        public static double Multiply(double a, double b)
        {
            return a * b;
        }

        // Divides an integer by another and returns the result
        /// <summary>
        /// Divides an integer <paramref name="a"/> by another
        /// integer <paramref name="b"/> and returns the result.
        /// </summary>
        /// <returns>
        /// The quotient of two integers.
        /// </returns>
        /// <example>
        /// <code>
        /// int c = Math.Divide(4, 5);
        /// if (c > 1)
        /// {
        ///     Console.WriteLine(c);
        /// }
        /// </code>
        /// </example>
        /// <exception cref="System.DivideByZeroException">
        /// Thrown when <paramref name="b"/> is equal to 0.
        /// </exception>
        /// See <see cref="Math.Divide(double, double)"/> to divide doubles.
```

```
        /// <seealso cref="Math.Add(int, int)"/>
        /// <seealso cref="Math.Subtract(int, int)"/>
        /// <seealso cref="Math.Multiply(int, int)"/>
        /// <param name="a">An integer dividend.</param>
        /// <param name="b">An integer divisor.</param>
        public static int Divide(int a, int b)
        {
            return a / b;
        }

        // Divides a double by another and returns the result
        /// <summary>
        /// Divides a double <paramref name="a"/> by another double
        /// <paramref name="b"/> and returns the result.
        /// </summary>
        /// <returns>
        /// The quotient of two doubles.
        /// </returns>
        /// <example>
        /// <code>
        /// double c = Math.Divide(4.5, 5.4);
        /// if (c > 1.0)
        /// {
        ///     Console.WriteLine(c);
        /// }
        /// </code>
        /// </example>
        /// <exception cref="System.DivideByZeroException">
        /// Thrown when <paramref name="b"/> is equal to 0.
        /// </exception>
        /// See <see cref="Math.Divide(int, int)"/> to divide integers.
        /// <seealso cref="Math.Add(double, double)"/>
        /// <seealso cref="Math.Subtract(double, double)"/>
        /// <seealso cref="Math.Multiply(double, double)"/>
        /// <param name="a">A double precision dividend.</param>
        /// <param name="b">A double precision divisor.</param>
        public static double Divide(double a, double b)
        {
            return a / b;
        }
    }
}
```

Vous pouvez constater que le code est dissimulé par tous les commentaires. L'exemple final montre comment adapter cette bibliothèque afin d'utiliser la balise `include`. Vous déplacez toute la documentation vers un fichier XML :

XML

```
<docs>
  <members name="math">
   <Math>
```

```
    <summary>
      The main <c>Math</c> class.
      Contains all methods for performing basic math functions.
      </summary>
      <remarks>
      <para>This class can add, subtract, multiply and divide.</para>
      <para>These operations can be performed on both integers and doubles.
</para>
      </remarks>
    </Math>
    <AddInt>
      <summary>
      Adds two integers <paramref name="a"/> and <paramref name="b"/>
      and returns the result.
      </summary>
      <returns>
      The sum of two integers.
      </returns>
      <example>
      <code>
      int c = Math.Add(4, 5);
      if (c > 10)
      {
        Console.WriteLine(c);
      }
      </code>
      </example>
      <exception cref="System.OverflowException">Thrown when one
      parameter is max
      and the other is greater than 0.</exception>
      See <see cref="Math.Add(double, double)"/> to add doubles.
      <seealso cref="Math.Subtract(int, int)"/>
      <seealso cref="Math.Multiply(int, int)"/>
      <seealso cref="Math.Divide(int, int)"/>
      <param name="a">An integer.</param>
      <param name="b">An integer.</param>
    </AddInt>
    <AddDouble>
      <summary>
      Adds two doubles <paramref name="a"/> and <paramref name="b"/>
      and returns the result.
      </summary>
      <returns>
      The sum of two doubles.
      </returns>
      <example>
      <code>
      double c = Math.Add(4.5, 5.4);
      if (c > 10)
      {
        Console.WriteLine(c);
      }
      </code>
      </example>
      <exception cref="System.OverflowException">Thrown when one parameter
```

```
is max
      and the other is greater than 0.</exception>
      See <see cref="Math.Add(int, int)"/> to add integers.
      <seealso cref="Math.Subtract(double, double)"/>
      <seealso cref="Math.Multiply(double, double)"/>
      <seealso cref="Math.Divide(double, double)"/>
      <param name="a">A double precision number.</param>
      <param name="b">A double precision number.</param>
    </AddDouble>
    <SubtractInt>
      <summary>
      Subtracts <paramref name="b"/> from <paramref name="a"/> and
      returns the result.
      </summary>
      <returns>
      The difference between two integers.
      </returns>
      <example>
      <code>
      int c = Math.Subtract(4, 5);
      if (c > 1)
      {
        Console.WriteLine(c);
      }
      </code>
      </example>
      See <see cref="Math.Subtract(double, double)"/> to subtract doubles.
      <seealso cref="Math.Add(int, int)"/>
      <seealso cref="Math.Multiply(int, int)"/>
      <seealso cref="Math.Divide(int, int)"/>
      <param name="a">An integer.</param>
      <param name="b">An integer.</param>
    </SubtractInt>
    <SubtractDouble>
      <summary>
      Subtracts a double <paramref name="b"/> from another
      double <paramref name="a"/> and returns the result.
      </summary>
      <returns>
      The difference between two doubles.
      </returns>
      <example>
      <code>
      double c = Math.Subtract(4.5, 5.4);
      if (c > 1)
      {
        Console.WriteLine(c);
      }
      </code>
      </example>
      See <see cref="Math.Subtract(int, int)"/> to subtract integers.
      <seealso cref="Math.Add(double, double)"/>
      <seealso cref="Math.Multiply(double, double)"/>
      <seealso cref="Math.Divide(double, double)"/>
      <param name="a">A double precision number.</param>
```

```
    <param name="b">A double precision number.</param>
  </SubtractDouble>
  <MultiplyInt>
    <summary>
    Multiplies two integers <paramref name="a"/> and
    <paramref name="b"/> and returns the result.
    </summary>
    <returns>
    The product of two integers.
    </returns>
    <example>
    <code>
    int c = Math.Multiply(4, 5);
    if (c > 100)
    {
      Console.WriteLine(c);
    }
    </code>
    </example>
    See <see cref="Math.Multiply(double, double)"/> to multiply doubles.
    <seealso cref="Math.Add(int, int)"/>
    <seealso cref="Math.Subtract(int, int)"/>
    <seealso cref="Math.Divide(int, int)"/>
    <param name="a">An integer.</param>
    <param name="b">An integer.</param>
  </MultiplyInt>
  <MultiplyDouble>
    <summary>
    Multiplies two doubles <paramref name="a"/> and
    <paramref name="b"/> and returns the result.
    </summary>
    <returns>
    The product of two doubles.
    </returns>
    <example>
    <code>
    double c = Math.Multiply(4.5, 5.4);
    if (c > 100.0)
    {
      Console.WriteLine(c);
    }
    </code>
    </example>
    See <see cref="Math.Multiply(int, int)"/> to multiply integers.
    <seealso cref="Math.Add(double, double)"/>
    <seealso cref="Math.Subtract(double, double)"/>
    <seealso cref="Math.Divide(double, double)"/>
    <param name="a">A double precision number.</param>
    <param name="b">A double precision number.</param>
  </MultiplyDouble>
  <DivideInt>
    <summary>
    Divides an integer <paramref name="a"/> by another integer
    <paramref name="b"/> and returns the result.
    </summary>
```

```
      <returns>
      The quotient of two integers.
      </returns>
      <example>
      <code>
      int c = Math.Divide(4, 5);
      if (c > 1)
      {
        Console.WriteLine(c);
      }
      </code>
      </example>
      <exception cref="System.DivideByZeroException">
      Thrown when <paramref name="b"/> is equal to 0.
      </exception>
      See <see cref="Math.Divide(double, double)"/> to divide doubles.
      <seealso cref="Math.Add(int, int)"/>
      <seealso cref="Math.Subtract(int, int)"/>
      <seealso cref="Math.Multiply(int, int)"/>
      <param name="a">An integer dividend.</param>
      <param name="b">An integer divisor.</param>
    </DivideInt>
    <DivideDouble>
      <summary>
      Divides a double <paramref name="a"/> by another
      double <paramref name="b"/> and returns the result.
      </summary>
      <returns>
      The quotient of two doubles.
      </returns>
      <example>
      <code>
      double c = Math.Divide(4.5, 5.4);
      if (c > 1.0)
      {
        Console.WriteLine(c);
      }
      </code>
      </example>
      <exception cref="System.DivideByZeroException">Thrown when <paramref
name="b"/> is equal to 0.</exception>
      See <see cref="Math.Divide(int, int)"/> to divide integers.
      <seealso cref="Math.Add(double, double)"/>
      <seealso cref="Math.Subtract(double, double)"/>
      <seealso cref="Math.Multiply(double, double)"/>
      <param name="a">A double precision dividend.</param>
      <param name="b">A double precision divisor.</param>
    </DivideDouble>
  </members>
</docs>
```

Dans le code XML ci-dessus, les commentaires de documentation de chaque membre apparaissent directement dans une balise nommée en fonction de ce qu'il fait. Vous

pouvez choisir votre propre stratégie. Le code utilise la balise `<include>` pour référencer l'élément approprié dans le fichier XML :

C#

```
namespace IncludeTag
{

    /*
        The main Math class
        Contains all methods for performing basic math functions
    */
    /// <include file='include.xml'
path='docs/members[@name="math"]/Math/*'/>
    public class Math
    {
        // Adds two integers and returns the result
        /// <include file='include.xml'
path='docs/members[@name="math"]/AddInt/*'/>
        public static int Add(int a, int b)
        {
            // If any parameter is equal to the max value of an integer
            // and the other is greater than zero
            if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a >
0))
                throw new System.OverflowException();

            return a + b;
        }

        // Adds two doubles and returns the result
        /// <include file='include.xml'
path='docs/members[@name="math"]/AddDouble/*'/>
        public static double Add(double a, double b)
        {
            // If any parameter is equal to the max value of an integer
            // and the other is greater than zero
            if ((a == double.MaxValue && b > 0) || (b == double.MaxValue &&
a > 0))
                throw new System.OverflowException();

            return a + b;
        }

        // Subtracts an integer from another and returns the result
        /// <include file='include.xml'
path='docs/members[@name="math"]/SubtractInt/*'/>
        public static int Subtract(int a, int b)
        {
            return a - b;
        }

        // Subtracts a double from another and returns the result
        /// <include file='include.xml'
```

```
path='docs/members[@name="math"]/SubtractDouble/*'/>
        public static double Subtract(double a, double b)
        {
            return a - b;
        }

        // Multiplies two integers and returns the result
        /// <include file='include.xml'
path='docs/members[@name="math"]/MultiplyInt/*'/>
        public static int Multiply(int a, int b)
        {
            return a * b;
        }

        // Multiplies two doubles and returns the result
        /// <include file='include.xml'
path='docs/members[@name="math"]/MultiplyDouble/*'/>
        public static double Multiply(double a, double b)
        {
            return a * b;
        }

        // Divides an integer by another and returns the result
        /// <include file='include.xml'
path='docs/members[@name="math"]/DivideInt/*'/>
        public static int Divide(int a, int b)
        {
            return a / b;
        }

        // Divides a double by another and returns the result
        /// <include file='include.xml'
path='docs/members[@name="math"]/DivideDouble/*'/>
        public static double Divide(double a, double b)
        {
            return a / b;
        }
    }
}
```

- L'attribut `file` représente le nom du fichier XML contenant la documentation.
- L'attribut `path` représente une requête XPath au `tag name` présent dans le `file` spécifié.
- L'attribut `name` représente le spécificateur de nom dans la balise qui précède les commentaires.
- L'attribut `id` qui peut être utilisé à la place de `name` représente l'ID de la balise qui précède les commentaires.

Erreurs du compilateur C#

Article • 07/04/2023

Certaines erreurs du compilateur C# possèdent des rubriques correspondantes qui expliquent pourquoi l'erreur est générée et, dans certains cas, comment corriger l'erreur. Suivez l'une des étapes suivantes pour savoir si une aide est fournie pour un message d'erreur donné.

- Si vous utilisez Visual Studio, choisissez le numéro d'erreur (par exemple, CS0029) dans la Fenêtre Sortie, puis appuyez sur la touche F1.
- Tapez le numéro d'erreur dans la zone *Filtrer par titre* dans la table des matières.

Si aucune de ces étapes n'aboutit à des informations sur l'erreur, envoyez des commentaires en bas de cette page en indiquant le numéro ou le texte de l'erreur.

Pour plus d'informations sur la configuration des options d'erreur et d'avertissement en C#, consultez la page Options du compilateur C# ou la page Générer, page du Concepteur de projets (C#).

> ⓘ **Notes**
>
> Il est possible que pour certains des éléments de l'interface utilisateur de Visual Studio, votre ordinateur affiche des noms ou des emplacements différents de ceux indiqués dans les instructions suivantes. L'édition de Visual Studio dont vous disposez et les paramètres que vous utilisez déterminent ces éléments. Pour plus d'informations, consultez **Personnalisation de l'IDE**.

Voir aussi

- Options du compilateur C#
- Générer, page du Concepteur de projets (C#)
- **WarningLevel** (Options du compilateur Visual C#)
- **NoWarn** (Options du compilateur C#)

Detailed table of contents

Article • 06/08/2023

- Foreword
- Introduction
- §1 Scope
- §2 Normative references
- §3 Terms and definitions
- §4 General description
- §5 Conformance
- §6 Lexical structure
  - §6.1 Programs
  - §6.2 Grammars
    - §6.2.1 General
    - §6.2.2 Grammar notation
    - §6.2.3 Lexical grammar
    - §6.2.4 Syntactic grammar
    - §6.2.5 Grammar ambiguities
  - §6.3 Lexical analysis
    - §6.3.1 General
    - §6.3.2 Line terminators
    - §6.3.3 Comments
    - §6.3.4 White space
  - §6.4 Tokens
    - §6.4.1 General
    - §6.4.2 Unicode character escape sequences
    - §6.4.3 Identifiers
    - §6.4.4 Keywords
    - §6.4.5 Literals
      - §6.4.5.1 General
      - §6.4.5.2 Boolean literals
      - §6.4.5.3 Integer literals
      - §6.4.5.4 Real literals
      - §6.4.5.5 Character literals
      - §6.4.5.6 String literals
      - §6.4.5.7 The null literal
    - §6.4.6 Operators and punctuators
  - §6.5 Pre-processing directives
    - §6.5.1 General
    - §6.5.2 Conditional compilation symbols

        - §6.5.3 Pre-processing expressions
        - §6.5.4 Definition directives
        - §6.5.5 Conditional compilation directives
        - §6.5.6 Diagnostic directives
        - §6.5.7 Region directives
        - §6.5.8 Line directives
        - §6.5.9 Pragma directives
- §7 Basic concepts
    - §7.1 Application startup
    - §7.2 Application termination
    - §7.3 Declarations
    - §7.4 Members
        - §7.4.1 General
        - §7.4.2 Namespace members
        - §7.4.3 Struct members
        - §7.4.4 Enumeration members
        - §7.4.5 Class members
        - §7.4.6 Interface members
        - §7.4.7 Array members
        - §7.4.8 Delegate members
    - §7.5 Member access
        - §7.5.1 General
        - §7.5.2 Declared accessibility
        - §7.5.3 Accessibility domains
        - §7.5.4 Protected access
        - §7.5.5 Accessibility constraints
    - §7.6 Signatures and overloading
    - §7.7 Scopes
        - §7.7.1 General
        - §7.7.2 Name hiding
            - §7.7.2.1 General
            - §7.7.2.2 Hiding through nesting
            - §7.7.2.3 Hiding through inheritance
    - §7.8 Namespace and type names
        - §7.8.1 General
        - §7.8.2 Unqualified names
        - §7.8.3 Fully qualified names
    - §7.9 Automatic memory management
    - §7.10 Execution order
- §8 Types

- §8.1 General
- §8.2 Reference types
  - §8.2.1 General
  - §8.2.2 Class types
  - §8.2.3 The object type
  - §8.2.4 The dynamic type
  - §8.2.5 The string type
  - §8.2.6 Interface types
  - §8.2.7 Array types
  - §8.2.8 Delegate types
- §8.3 Value types
  - §8.3.1 General
  - §8.3.2 The System.ValueType type
  - §8.3.3 Default constructors
  - §8.3.4 Struct types
  - §8.3.5 Simple types
  - §8.3.6 Integral types
  - §8.3.7 Floating-point types
  - §8.3.8 The Decimal type
  - §8.3.9 The Bool type
  - §8.3.10 Enumeration types
  - §8.3.11 Tuple types
  - §8.3.12 Nullable value types
  - §8.3.13 Boxing and unboxing
- §8.4 Constructed types
  - §8.4.1 General
  - §8.4.2 Type arguments
  - §8.4.3 Open and closed types
  - §8.4.4 Bound and unbound types
  - §8.4.5 Satisfying constraints
- §8.5 Type parameters
- §8.6 Expression tree types
- §8.7 The dynamic type
- §8.8 Unmanaged types

- §9 Variables
  - §9.1 General
  - §9.2 Variable categories
    - §9.2.1 General
    - §9.2.2 Static variables
    - §9.2.3 Instance variables

- §9.2.3.1 General
- §9.2.3.2 Instance variables in classes
- §9.2.3.3 Instance variables in structs
- §9.2.4 Array elements
- §9.2.5 Value parameters
- §9.2.6 Reference parameters
- §9.2.7 Output parameters
- §9.2.8 Input parameters
- §9.2.9 Local variables
  - §9.2.9.1 Discards
- §9.3 Default values
- §9.4 Definite assignment
  - §9.4.1 General
  - §9.4.2 Initially assigned variables
  - §9.4.3 Initially unassigned variables
  - §9.4.4 Precise rules for determining definite assignment
    - §9.4.4.1 General
    - §9.4.4.2 General rules for statements
    - §9.4.4.3 Block statements, checked, and unchecked statements
    - §9.4.4.4 Expression statements
    - §9.4.4.5 Declaration statements
    - §9.4.4.6 If statements
    - §9.4.4.7 Switch statements
    - §9.4.4.8 While statements
    - §9.4.4.9 Do statements
    - §9.4.4.10 For statements
    - §9.4.4.11 Break, continue, and goto statements
    - §9.4.4.12 Throw statements
    - §9.4.4.13 Return statements
    - §9.4.4.14 Try-catch statements
    - §9.4.4.15 Try-finally statements
    - §9.4.4.16 Try-catch-finally statements
    - §9.4.4.17 Foreach statements
    - §9.4.4.18 Using statements
    - §9.4.4.19 Lock statements
    - §9.4.4.20 Yield statements
    - §9.4.4.21 General rules for constant expressions
    - §9.4.4.22 General rules for simple expressions
    - §9.4.4.23 General rules for expressions with embedded expressions
    - §9.4.4.24 Invocation expressions and object creation expressions

        - §9.4.4.25 Simple assignment expressions
        - §9.4.4.26 && expressions
        - §9.4.4.27 || expressions
        - §9.4.4.28 ! expressions
        - §9.4.4.29 ?? expressions
        - §9.4.4.30 ?: expressions
        - §9.4.4.31 Anonymous functions
        - §9.4.4.32 Throw expressions
        - §9.4.4.33 Rules for variables in local functions
        - §9.4.4.34 is-pattern expressions
    - §9.5 Variable references
    - §9.6 Atomicity of variable references
    - §9.7 Reference variables and returns
        - §9.7.1 General
        - §9.7.2 Ref safe contexts
            - §9.7.2.1 General
            - §9.7.2.2 Local variable ref safe context
            - §9.7.2.3 Parameter ref safe context
            - §9.7.2.4 Field ref safe context
            - §9.7.2.5 Operators
            - §9.7.2.6 Function invocation
            - §9.7.2.7 Values
            - §9.7.2.8 Constructor invocations
            - §9.7.2.9 Limitations on reference variables
- §10 Conversions
    - §10.1 General
    - §10.2 Implicit conversions
        - §10.2.1 General
        - §10.2.2 Identity conversion
        - §10.2.3 Implicit numeric conversions
        - §10.2.4 Implicit enumeration conversions
        - §10.2.5 Implicit interpolated string conversions
        - §10.2.6 Implicit nullable conversions
        - §10.2.7 Null literal conversions
        - §10.2.8 Implicit reference conversions
        - §10.2.9 Boxing conversions
        - §10.2.10 Implicit dynamic conversions
        - §10.2.11 Implicit constant expression conversions
        - §10.2.12 Implicit conversions involving type parameters
        - §10.2.13 Implicit tuple conversions

    - §10.2.14 User-defined implicit conversions
    - §10.2.15 Anonymous function conversions and method group conversions
    - §10.2.16 Default literal conversions
    - §10.2.17 Implicit throw conversions
  - §10.3 Explicit conversions
    - §10.3.1 General
    - §10.3.2 Explicit numeric conversions
    - §10.3.3 Explicit enumeration conversions
    - §10.3.4 Explicit nullable conversions
    - §10.3.5 Explicit reference conversions
    - §10.3.6 Explicit tuple conversions
    - §10.3.7 Unboxing conversions
    - §10.3.8 Explicit dynamic conversions
    - §10.3.9 Explicit conversions involving type parameters
    - §10.3.10 User-defined explicit conversions
  - §10.4 Standard conversions
    - §10.4.1 General
    - §10.4.2 Standard implicit conversions
    - §10.4.3 Standard explicit conversions
  - §10.5 User-defined conversions
    - §10.5.1 General
    - §10.5.2 Permitted user-defined conversions
    - §10.5.3 Evaluation of user-defined conversions
    - §10.5.4 User-defined implicit conversions
    - §10.5.5 User-defined explicit conversions
  - §10.6 Conversions involving nullable types
    - §10.6.1 Nullable Conversions
    - §10.6.2 Lifted conversions
  - §10.7 Anonymous function conversions
    - §10.7.1 General
    - §10.7.2 Evaluation of anonymous function conversions to delegate types
    - §10.7.3 Evaluation of lambda expression conversions to expression tree types
  - §10.8 Method group conversions
- §11 Patterns and pattern matching
  - §11.1 General
  - §11.2 Pattern forms
    - §11.2.1 General
    - §11.2.2 Declaration pattern
    - §11.2.3 Constant pattern
    - §11.2.4 Var pattern

- §11.3 Pattern subsumption
- §11.4 Pattern exhaustiveness

- §12 Expressions
  - §12.1 General
  - §12.2 Expression classifications
    - §12.2.1 General
    - §12.2.2 Values of expressions
  - §12.3 Static and Dynamic Binding
    - §12.3.1 General
    - §12.3.2 Binding-time
    - §12.3.3 Dynamic binding
    - §12.3.4 Types of subexpressions
  - §12.4 Operators
    - §12.4.1 General
    - §12.4.2 Operator precedence and associativity
    - §12.4.3 Operator overloading
    - §12.4.4 Unary operator overload resolution
    - §12.4.5 Binary operator overload resolution
    - §12.4.6 Candidate user-defined operators
    - §12.4.7 Numeric promotions
      - §12.4.7.1 General
      - §12.4.7.2 Unary numeric promotions
      - §12.4.7.3 Binary numeric promotions
    - §12.4.8 Lifted operators
  - §12.5 Member lookup
    - §12.5.1 General
    - §12.5.2 Base types
  - §12.6 Function members
    - §12.6.1 General
    - §12.6.2 Argument lists
      - §12.6.2.1 General
      - §12.6.2.2 Corresponding parameters
      - §12.6.2.3 Run-time evaluation of argument lists
    - §12.6.3 Type inference
      - §12.6.3.1 General
      - §12.6.3.2 The first phase
      - §12.6.3.3 The second phase
      - §12.6.3.4 Input types
      - §12.6.3.5 Output types
      - §12.6.3.6 Dependence

- §12.6.3.7 Output type inferences
- §12.6.3.8 Explicit parameter type inferences
- §12.6.3.9 Exact inferences
- §12.6.3.10 Lower-bound inferences
- §12.6.3.11 Upper-bound inferences
- §12.6.3.12 Fixing
- §12.6.3.13 Inferred return type
- §12.6.3.14 Type inference for conversion of method groups
- §12.6.3.15 Finding the best common type of a set of expressions
- §12.6.4 Overload resolution
  - §12.6.4.1 General
  - §12.6.4.2 Applicable function member
  - §12.6.4.3 Better function member
  - §12.6.4.4 Better parameter-passing mode
  - §12.6.4.5 Better conversion from expression
  - §12.6.4.6 Exactly matching expression
  - §12.6.4.7 Better conversion target
  - §12.6.4.8 Overloading in generic classes
- §12.6.5 Compile-time checking of dynamic member invocation
- §12.6.6 Function member invocation
  - §12.6.6.1 General
  - §12.6.6.2 Invocations on boxed instances
- §12.7 Deconstruction
- §12.8 Primary expressions
  - §12.8.1 General
  - §12.8.2 Literals
  - §12.8.3 Interpolated string expressions
  - §12.8.4 Simple names
  - §12.8.5 Parenthesized expressions
  - §12.8.6 Tuple expressions
  - §12.8.7 Member access
    - §12.8.7.1 General
    - §12.8.7.2 Identical simple names and type names
  - §12.8.8 Null Conditional Member Access
  - §12.8.9 Invocation expressions
    - §12.8.9.1 General
    - §12.8.9.2 Method invocations
    - §12.8.9.3 Extension method invocations
    - §12.8.9.4 Delegate invocations
  - §12.8.10 Null Conditional Invocation Expression

- §12.8.11 Element access
  - §12.8.11.1 General
  - §12.8.11.2 Array access
  - §12.8.11.3 Indexer access
- §12.8.12 Null Conditional Element Access
- §12.8.13 This access
- §12.8.14 Base access
- §12.8.15 Postfix increment and decrement operators
- §12.8.16 The new operator
  - §12.8.16.1 General
  - §12.8.16.2 Object creation expressions
  - §12.8.16.3 Object initializers
  - §12.8.16.4 Collection initializers
  - §12.8.16.5 Array creation expressions
  - §12.8.16.6 Delegate creation expressions
  - §12.8.16.7 Anonymous object creation expressions
- §12.8.17 The typeof operator
- §12.8.18 The sizeof operator
- §12.8.19 The checked and unchecked operators
- §12.8.20 Default value expressions
- §12.8.21 Stack allocation
- §12.8.22 Nameof expressions
- §12.8.23 Anonymous method expressions

- §12.9 Unary operators
  - §12.9.1 General
  - §12.9.2 Unary plus operator
  - §12.9.3 Unary minus operator
  - §12.9.4 Logical negation operator
  - §12.9.5 Bitwise complement operator
  - §12.9.6 Prefix increment and decrement operators
  - §12.9.7 Cast expressions
  - §12.9.8 Await expressions
    - §12.9.8.1 General
    - §12.9.8.2 Awaitable expressions
    - §12.9.8.3 Classification of await expressions
    - §12.9.8.4 Run-time evaluation of await expressions
- §12.10 Arithmetic operators
  - §12.10.1 General
  - §12.10.2 Multiplication operator
  - §12.10.3 Division operator

- §12.10.4 Remainder operator
- §12.10.5 Addition operator
- §12.10.6 Subtraction operator
- §12.11 Shift operators
- §12.12 Relational and type-testing operators
  - §12.12.1 General
  - §12.12.2 Integer comparison operators
  - §12.12.3 Floating-point comparison operators
  - §12.12.4 Decimal comparison operators
  - §12.12.5 Boolean equality operators
  - §12.12.6 Enumeration comparison operators
  - §12.12.7 Reference type equality operators
  - §12.12.8 String equality operators
  - §12.12.9 Delegate equality operators
  - §12.12.10 Equality operators between nullable value types and the null literal
  - §12.12.11 Tuple equality operators
  - §12.12.12 The is operator
    - §12.12.12.1 The is-type operator
    - §12.12.12.2 The is-pattern operator
  - §12.12.13 The as operator
- §12.13 Logical operators
  - §12.13.1 General
  - §12.13.2 Integer logical operators
  - §12.13.3 Enumeration logical operators
  - §12.13.4 Boolean logical operators
  - §12.13.5 Nullable Boolean & and | operators
- §12.14 Conditional logical operators
  - §12.14.1 General
  - §12.14.2 Boolean conditional logical operators
  - §12.14.3 User-defined conditional logical operators
- §12.15 The null coalescing operator
- §12.16 The throw expression operator
- §12.17 Declaration expressions
- §12.18 Conditional operator
- §12.19 Anonymous function expressions
  - §12.19.1 General
  - §12.19.2 Anonymous function signatures
  - §12.19.3 Anonymous function bodies
  - §12.19.4 Overload resolution
  - §12.19.5 Anonymous functions and dynamic binding

        - §12.19.6 Outer variables
            - §12.19.6.1 General
            - §12.19.6.2 Captured outer variables
            - §12.19.6.3 Instantiation of local variables
        - §12.19.7 Evaluation of anonymous function expressions
        - §12.19.8 Implementation Example
    - §12.20 Query expressions
        - §12.20.1 General
        - §12.20.2 Ambiguities in query expressions
        - §12.20.3 Query expression translation
            - §12.20.3.1 General
            - §12.20.3.2 Query expressions with continuations
            - §12.20.3.3 Explicit range variable types
            - §12.20.3.4 Degenerate query expressions
            - §12.20.3.5 From, let, where, join and orderby clauses
            - §12.20.3.6 Select clauses
            - §12.20.3.7 Group clauses
            - §12.20.3.8 Transparent identifiers
        - §12.20.4 The query-expression pattern
    - §12.21 Assignment operators
        - §12.21.1 General
        - §12.21.2 Simple assignment
        - §12.21.3 Ref assignment
        - §12.21.4 Compound assignment
        - §12.21.5 Event assignment
    - §12.22 Expression
    - §12.23 Constant expressions
    - §12.24 Boolean expressions
- §13 Statements
    - §13.1 General
    - §13.2 End points and reachability
    - §13.3 Blocks
        - §13.3.1 General
        - §13.3.2 Statement lists
    - §13.4 The empty statement
    - §13.5 Labeled statements
    - §13.6 Declaration statements
        - §13.6.1 General
        - §13.6.2 Local variable declarations
        - §13.6.3 Local constant declarations

    - §13.6.4 Local function declarations
  - §13.7 Expression statements
  - §13.8 Selection statements
    - §13.8.1 General
    - §13.8.2 The if statement
    - §13.8.3 The switch statement
  - §13.9 Iteration statements
    - §13.9.1 General
    - §13.9.2 The while statement
    - §13.9.3 The do statement
    - §13.9.4 The for statement
    - §13.9.5 The foreach statement
  - §13.10 Jump statements
    - §13.10.1 General
    - §13.10.2 The break statement
    - §13.10.3 The continue statement
    - §13.10.4 The goto statement
    - §13.10.5 The return statement
    - §13.10.6 The throw statement
  - §13.11 The try statement
  - §13.12 The checked and unchecked statements
  - §13.13 The lock statement
  - §13.14 The using statement
  - §13.15 The yield statement
- §14 Namespaces
  - §14.1 General
  - §14.2 Compilation units
  - §14.3 Namespace declarations
  - §14.4 Extern alias directives
  - §14.5 Using directives
    - §14.5.1 General
    - §14.5.2 Using alias directives
    - §14.5.3 Using namespace directives
    - §14.5.4 Using static directives
  - §14.6 Namespace member declarations
  - §14.7 Type declarations
  - §14.8 Qualified alias member
    - §14.8.1 General
    - §14.8.2 Uniqueness of aliases
- §15 Classes

- §15.1 General
- §15.2 Class declarations
  - §15.2.1 General
  - §15.2.2 Class modifiers
    - §15.2.2.1 General
    - §15.2.2.2 Abstract classes
    - §15.2.2.3 Sealed classes
    - §15.2.2.4 Static classes
      - §15.2.2.4.1 General
      - §15.2.2.4.2 Referencing static class types
  - §15.2.3 Type parameters
  - §15.2.4 Class base specification
    - §15.2.4.1 General
    - §15.2.4.2 Base classes
    - §15.2.4.3 Interface implementations
  - §15.2.5 Type parameter constraints
  - §15.2.6 Class body
  - §15.2.7 Partial declarations
- §15.3 Class members
  - §15.3.1 General
  - §15.3.2 The instance type
  - §15.3.3 Members of constructed types
  - §15.3.4 Inheritance
  - §15.3.5 The new modifier
  - §15.3.6 Access modifiers
  - §15.3.7 Constituent types
  - §15.3.8 Static and instance members
  - §15.3.9 Nested types
    - §15.3.9.1 General
    - §15.3.9.2 Fully qualified name
    - §15.3.9.3 Declared accessibility
    - §15.3.9.4 Hiding
    - §15.3.9.5 this access
    - §15.3.9.6 Access to private and protected members of the containing type
    - §15.3.9.7 Nested types in generic classes
  - §15.3.10 Reserved member names
    - §15.3.10.1 General
    - §15.3.10.2 Member names reserved for properties
    - §15.3.10.3 Member names reserved for events
    - §15.3.10.4 Member names reserved for indexers

- §15.3.10.5 Member names reserved for finalizers
- §15.4 Constants
- §15.5 Fields
  - §15.5.1 General
  - §15.5.2 Static and instance fields
  - §15.5.3 Readonly fields
    - §15.5.3.1 General
    - §15.5.3.2 Using static readonly fields for constants
    - §15.5.3.3 Versioning of constants and static readonly fields
  - §15.5.4 Volatile fields
  - §15.5.5 Field initialization
  - §15.5.6 Variable initializers
    - §15.5.6.1 General
    - §15.5.6.2 Static field initialization
    - §15.5.6.3 Instance field initialization
- §15.6 Methods
  - §15.6.1 General
  - §15.6.2 Method parameters
    - §15.6.2.1 General
    - §15.6.2.2 Value parameters
    - §15.6.2.3 Input parameters
    - §15.6.2.4 Reference parameters
    - §15.6.2.5 Output parameters
    - §15.6.2.6 Parameter arrays
  - §15.6.3 Static and instance methods
  - §15.6.4 Virtual methods
  - §15.6.5 Override methods
  - §15.6.6 Sealed methods
  - §15.6.7 Abstract methods
  - §15.6.8 External methods
  - §15.6.9 Partial methods
  - §15.6.10 Extension methods
  - §15.6.11 Method body
- §15.7 Properties
  - §15.7.1 General
  - §15.7.2 Static and instance properties
  - §15.7.3 Accessors
  - §15.7.4 Automatically implemented properties
  - §15.7.5 Accessibility
  - §15.7.6 Virtual, sealed, override, and abstract accessors

- §15.8 Events
  - §15.8.1 General
  - §15.8.2 Field-like events
  - §15.8.3 Event accessors
  - §15.8.4 Static and instance events
  - §15.8.5 Virtual, sealed, override, and abstract accessors
- §15.9 Indexers
  - §15.9.1 General
  - §15.9.2 Indexer and Property Differences
- §15.10 Operators
  - §15.10.1 General
  - §15.10.2 Unary operators
  - §15.10.3 Binary operators
  - §15.10.4 Conversion operators
- §15.11 Instance constructors
  - §15.11.1 General
  - §15.11.2 Constructor initializers
  - §15.11.3 Instance variable initializers
  - §15.11.4 Constructor execution
  - §15.11.5 Default constructors
- §15.12 Static constructors
- §15.13 Finalizers
- §15.14 Iterators
  - §15.14.1 General
  - §15.14.2 Enumerator interfaces
  - §15.14.3 Enumerable interfaces
  - §15.14.4 Yield type
  - §15.14.5 Enumerator objects
    - §15.14.5.1 General
    - §15.14.5.2 The MoveNext method
    - §15.14.5.3 The Current property
    - §15.14.5.4 The Dispose method
  - §15.14.6 Enumerable objects
    - §15.14.6.1 General
    - §15.14.6.2 The GetEnumerator method
- §15.15 Async Functions
  - §15.15.1 General
  - §15.15.2 Task-type builder pattern
  - §15.15.3 Evaluation of a task-returning async function
  - §15.15.4 Evaluation of a void-returning async function

- §16 Structs
  - §16.1 General
  - §16.2 Struct declarations
    - §16.2.1 General
    - §16.2.2 Struct modifiers
    - §16.2.3 Ref modifier
    - §16.2.4 Partial modifier
    - §16.2.5 Struct interfaces
    - §16.2.6 Struct body
  - §16.3 Struct members
  - §16.4 Class and struct differences
    - §16.4.1 General
    - §16.4.2 Value semantics
    - §16.4.3 Inheritance
    - §16.4.4 Assignment
    - §16.4.5 Default values
    - §16.4.6 Boxing and unboxing
    - §16.4.7 Meaning of this
    - §16.4.8 Field initializers
    - §16.4.9 Constructors
    - §16.4.10 Static constructors
    - §16.4.11 Automatically implemented properties
    - §16.4.12 Safe context constraint
      - §16.4.12.1 General
      - §16.4.12.2 Parameter safe context
      - §16.4.12.3 Local variable safe context
      - §16.4.12.4 Field safe context
      - §16.4.12.5 Operators
      - §16.4.12.6 Method and property invocation
      - §16.4.12.7 stackalloc
      - §16.4.12.8 Constructor invocations
- §17 Arrays
  - §17.1 General
  - §17.2 Array types
    - §17.2.1 General
    - §17.2.2 The System.Array type
    - §17.2.3 Arrays and the generic collection interfaces
  - §17.3 Array creation
  - §17.4 Array element access
  - §17.5 Array members

- §17.6 Array covariance
- §17.7 Array initializers
- §18 Interfaces
  - §18.1 General
  - §18.2 Interface declarations
    - §18.2.1 General
    - §18.2.2 Interface modifiers
    - §18.2.3 Variant type parameter lists
      - §18.2.3.1 General
      - §18.2.3.2 Variance safety
      - §18.2.3.3 Variance conversion
    - §18.2.4 Base interfaces
  - §18.3 Interface body
  - §18.4 Interface members
    - §18.4.1 General
    - §18.4.2 Interface methods
    - §18.4.3 Interface properties
    - §18.4.4 Interface events
    - §18.4.5 Interface indexers
    - §18.4.6 Interface member access
  - §18.5 Qualified interface member names
  - §18.6 Interface implementations
    - §18.6.1 General
    - §18.6.2 Explicit interface member implementations
    - §18.6.3 Uniqueness of implemented interfaces
    - §18.6.4 Implementation of generic methods
    - §18.6.5 Interface mapping
    - §18.6.6 Interface implementation inheritance
    - §18.6.7 Interface re-implementation
    - §18.6.8 Abstract classes and interfaces
- §19 Enums
  - §19.1 General
  - §19.2 Enum declarations
  - §19.3 Enum modifiers
  - §19.4 Enum members
  - §19.5 The System.Enum type
  - §19.6 Enum values and operations
- §20 Delegates
  - §20.1 General
  - §20.2 Delegate declarations

- §20.3 Delegate members
- §20.4 Delegate compatibility
- §20.5 Delegate instantiation
- §20.6 Delegate invocation

- §21 Exceptions
  - §21.1 General
  - §21.2 Causes of exceptions
  - §21.3 The System.Exception class
  - §21.4 How exceptions are handled
  - §21.5 Common exception classes

- §22 Attributes
  - §22.1 General
  - §22.2 Attribute classes
    - §22.2.1 General
    - §22.2.2 Attribute usage
    - §22.2.3 Positional and named parameters
    - §22.2.4 Attribute parameter types
  - §22.3 Attribute specification
  - §22.4 Attribute instances
    - §22.4.1 General
    - §22.4.2 Compilation of an attribute
    - §22.4.3 Run-time retrieval of an attribute instance
  - §22.5 Reserved attributes
    - §22.5.1 General
    - §22.5.2 The AttributeUsage attribute
    - §22.5.3 The Conditional attribute
      - §22.5.3.1 General
      - §22.5.3.2 Conditional methods
      - §22.5.3.3 Conditional attribute classes
    - §22.5.4 The Obsolete attribute
    - §22.5.5 Caller-info attributes
      - §22.5.5.1 General
      - §22.5.5.2 The CallerLineNumber attribute
      - §22.5.5.3 The CallerFilePath attribute
      - §22.5.5.4 The CallerMemberName attribute
  - §22.6 Attributes for interoperation

- §23 Unsafe code
  - §23.1 General
  - §23.2 Unsafe contexts
  - §23.3 Pointer types

    - §23.4 Fixed and moveable variables
    - §23.5 Pointer conversions
        - §23.5.1 General
        - §23.5.2 Pointer arrays
    - §23.6 Pointers in expressions
        - §23.6.1 General
        - §23.6.2 Pointer indirection
        - §23.6.3 Pointer member access
        - §23.6.4 Pointer element access
        - §23.6.5 The address-of operator
        - §23.6.6 Pointer increment and decrement
        - §23.6.7 Pointer arithmetic
        - §23.6.8 Pointer comparison
        - §23.6.9 The sizeof operator
    - §23.7 The fixed statement
    - §23.8 Fixed-size buffers
        - §23.8.1 General
        - §23.8.2 Fixed-size buffer declarations
        - §23.8.3 Fixed-size buffers in expressions
        - §23.8.4 Definite assignment checking
    - §23.9 Stack allocation
- §A Grammar
    - §A.1 General
    - §A.2 Lexical grammar
    - §A.3 Syntactic grammar
    - §A.4 Grammar extensions for unsafe code
- §B Portability issues
    - §B.1 General
    - §B.2 Undefined behavior
    - §B.3 Implementation-defined behavior
    - §B.4 Unspecified behavior
    - §B.5 Other issues
- §C Standard library
    - §C.1 General
    - §C.2 Standard Library Types defined in ISO/IEC 23271
    - §C.3 Standard Library Types not defined in ISO/IEC 23271
    - §C.4 Format Specifications
    - §C.5 Library Type Abbreviations
- §D Documentation comments
    - §D.1 General

- §D.2 Introduction
- §D.3 Recommended tags
  - §D.3.1 General
  - §D.3.2 <c>
  - §D.3.3 <code>
  - §D.3.4 <example>
  - §D.3.5 <exception>
  - §D.3.6 <include>
  - §D.3.7 <list>
  - §D.3.8 <para>
  - §D.3.9 <param>
  - §D.3.10 <paramref>
  - §D.3.11 <permission>
  - §D.3.12 <remarks>
  - §D.3.13 <returns>
  - §D.3.14 <see>
  - §D.3.15 <seealso>
  - §D.3.16 <summary>
  - §D.3.17 <typeparam>
  - §D.3.18 <typeparamref>
  - §D.3.19 <value>
- §D.4 Processing the documentation file
  - §D.4.1 General
  - §D.4.2 ID string format
  - §D.4.3 ID string examples
- §D.5 An example
  - §D.5.1 C# source code
  - §D.5.2 Resulting XML
- §E Bibliography

Foreword

Article • 03/23/2023

This specification replaces ECMA-334:2022. Changes from the previous edition include the addition of the following:

- Binary integer literals
- Embedded digit separators in numeric literals
- Leading-digit separators in binary and hexadecimal integer literals
- `out` variables
- Discards
- Tuple types
- Pattern Matching
- `ref` locals and returns, conditional `ref` expressions, `ref` with `this` in extension methods, and reassignment of `ref` local variables
- Local Functions
- More expression-bodied members
- `throw` Expressions
- Generalized `async` return types
- `async Main` method
- `default` literal expressions
- Non-trailing named arguments
- `private protected` access modifier
- `in` parameter modifier
- `readonly` structs
- `ref` structs
- Indexing movable fixed buffer without pinning
- Initializers on `stackalloc` arrays
- Pattern-based `fixed` statements
- `System.Delegate` and `System.Enum` as *class_type* constraints.
- Additional generic constraints
- Allow expression variables in more locations
- Attach attributes to the backing field of auto-implemented properties
- Reduce ambiguity of overload resolution

Introduction

Article • 13/05/2021

C# (prononcé « C Sharp ») est un langage de programmation simple, moderne, orienté objet et de type sécurisé. C# a ses racines dans la famille de langages C et sera immédiatement familiarisé avec les programmeurs C, C++ et Java. C# est standardisé par ECMA International en tant que * norme ***ECMA-334*** _ *et par ISO/IEC comme* _ *iso/IEC 23270* standard. Le compilateur C# de Microsoft pour le .NET Framework est une implémentation conforme de ces deux normes.

C# est un langage orienté objet, mais C# inclut de plus la prise en charge de la programmation ***orientée composant***. La conception logicielle moderne s'appuie de plus en plus sur les composants logiciels sous la forme de packages de fonctionnalités autonomes et autodescriptifs. Point important, ces composants présentent un modèle de programmation avec propriétés, méthodes et événements ; ils ont des attributs qui fournissent des informations déclaratives sur le composant ; et ils intègrent leur propre documentation. C# fournit des constructions de langage pour prendre en charge directement ces concepts, en faisant de C# un langage très naturel dans lequel créer et utiliser des composants logiciels.

Plusieurs fonctionnalités C# aident à la construction d'applications robustes et durables : ***garbage collection*** _ *récupère automatiquement la mémoire occupée par les objets inutilisés ; la* _*gestion des exceptions*_ *fournit une approche structurée et extensible de la détection et de la récupération des erreurs ; la conception de* _ *type-safe* du langage rend impossible la lecture des variables non initialisées, l'indexation des tableaux au-delà de leurs limites ou l'exécution de conversions de type non vérifiées.

C# a un ***système de type unifié***. Tous les types C#, y compris les types primitifs tels que `int` et `double`, héritent d'un seul type `object` racine. Par conséquent, tous les types partagent un ensemble d'opérations communes, et des valeurs de tous types peuvent être stockées, transmises et exploitées de manière cohérente. En outre, C# prend en charge les types référence et les types valeur définis par l'utilisateur, ce qui permet l'allocation dynamique d'objets, ainsi que le stockage en ligne de structures légères.

Pour vous assurer que les bibliothèques et les programmes C# peuvent évoluer au fil du temps d'une manière compatible, l'accent est mis sur le contrôle de ***version*** dans la conception de c#. De nombreux langages de programmation n'accordent que peu d'attention à ce problème ; par conséquent, les programmes écrits dans ces langages s'arrêtent plus souvent que nécessaire lors de l'introduction de versions plus récentes des bibliothèques dépendantes. Les aspects de la conception de C# qui ont été directement influencés par les considérations relatives à la gestion des versions incluent

les `virtual` `override` modificateurs et séparés, les règles de résolution de surcharge de méthode et la prise en charge des déclarations de membres d'interface explicites.

Le reste de ce chapitre décrit les principales fonctionnalités du langage C#. Bien que les chapitres ultérieurs décrivent les règles et les exceptions selon une méthode orientée détails et parfois mathématique, ce chapitre s'efforce de clarifier et de concision au détriment de l'exhaustivité. L'objectif est de fournir au lecteur une introduction au langage qui facilitera l'écriture de programmes précoces et la lecture des chapitres ultérieurs.

Hello World

Le programme « Hello, World » est souvent utilisé pour présenter un langage de programmation. Le voici en C# :

C#

```
using System;

class Hello
{
    static void Main() {
        Console.WriteLine("Hello, World");
    }
}
```

Les fichiers sources C# ont généralement l'extension de fichier `.cs`. En supposant que le programme « Hello, World » est stocké dans le fichier `hello.cs` , le programme peut être compilé avec le compilateur Microsoft C# à l'aide de la ligne de commande.

Console

```
csc hello.cs
```

qui produit un assembly exécutable nommé `hello.exe` . La sortie produite par cette application lorsqu'elle est exécutée est

Console

```
Hello, World
```

Le programme « Hello, World » commence par une directive `using` qui fait référence à l'espace de noms `System`. Les espaces de noms représentent un moyen hiérarchique

d'organiser les bibliothèques et les programmes C#. Les espaces de noms contiennent des types et d'autres espaces de noms ; par exemple, l'espace de noms `System` contient plusieurs types, notamment la classe `Console` référencée dans le programme, et d'autres espaces de noms, tels que `IO` et `Collections`. Une directive `using` qui fait référence à un espace de noms donné permet l'utilisation non qualifiée des types membres de cet espace de noms. En raison de la directive `using`, le programme peut utiliser `Console.WriteLine` comme raccourci pour `System.Console.WriteLine`.

La classe `Hello` déclarée par le programme « Hello, World » a un membre unique, la méthode nommée `Main`. La `Main` méthode est déclarée avec le `static` modificateur. Si les méthodes d'instance peuvent faire référence à une instance d'objet englobante particulière avec le mot clé `this`, les méthodes statiques fonctionnent sans référence à un objet particulier. Par convention, une méthode statique nommée `Main` sert de point d'entrée d'un programme.

La sortie du programme est générée par la méthode `WriteLine` de la classe `Console` dans l'espace de noms `System`. Cette classe est fournie par les bibliothèques de classes .NET Framework, qui, par défaut, sont automatiquement référencées par le compilateur Microsoft C#. Notez que C# lui-même n'a pas de bibliothèque Runtime distincte. Au lieu de cela, le .NET Framework est la bibliothèque Runtime de C#.

Structure du programme

Les concepts clés de l'organisation en C# sont *__Programs__, *espaces de noms*, *types*, *membres* et *assemblys*. Les programmes C++, comme les programmes C, se composent d'un ou plusieurs fichiers sources. Les programmes déclarent des types qui contiennent des membres et peuvent être organisés en espaces de noms. Les classes et les interfaces sont des exemples de types. Les champs, méthodes, propriétés et événements sont des exemples de membres. Lorsque les programmes C# sont compilés, ils sont physiquement empaquetés dans des assemblys. Les assemblys ont généralement l'extension de fichier `.exe` ou `.dll` , selon qu'ils implémentent des *applications* ou _ *Libraries* *.

L'exemple

C#

```
using System;

namespace Acme.Collections
{
    public class Stack
```

```
    {
        Entry top;

        public void Push(object data) {
            top = new Entry(top, data);
        }

        public object Pop() {
            if (top == null) throw new InvalidOperationException();
            object result = top.data;
            top = top.next;
            return result;
        }

        class Entry
        {
            public Entry next;
            public object data;

            public Entry(Entry next, object data) {
                this.next = next;
                this.data = data;
            }
        }
    }
}
```

déclare une classe nommée `Stack` dans un espace de noms appelé `Acme.Collections` . Le nom qualifié complet de cette classe est `Acme.Collections.Stack`. La classe contient plusieurs membres : un champ nommé `top`, deux méthodes nommées `Push` et `Pop`, et une classe imbriquée nommée `Entry`. La classe `Entry` contient trois membres en plus : un champ nommé `next`, un autre nommé `data` et un constructeur. En supposant que le code source de l'exemple est stocké dans le fichier `acme.cs`, la ligne de commande

Console

```
csc /t:library acme.cs
```

compile l'exemple en tant que bibliothèque (code sans point d'entrée `Main`) et produit un assembly nommé `acme.dll`.

Les assemblys contiennent du code exécutable sous la forme d'instructions de **langage intermédiaire** _ (il) et des informations symboliques sous la forme _ *Metadata* *. Avant son exécution, le code de langage intermédiaire dans un assembly est automatiquement converti en code spécifique au processeur par le compilateur juste à temps (JIT) du Common Language Runtime .NET.

Comme un assembly est une unité de fonctionnalité autodescriptive contenant du code et des métadonnées, les directives `#include` et les fichiers d'en-tête ne sont pas nécessaires en C#. Les membres et types publics contenus dans un assembly particulier sont disponibles dans un programme C# par simple référence à cet assembly lors de la compilation du programme. Par exemple, ce programme utilise la classe `Acme.Collections.Stack` à partir de l'assembly `acme.dll` :

C#

```
using System;
using Acme.Collections;

class Test
{
    static void Main() {
        Stack s = new Stack();
        s.Push(1);
        s.Push(10);
        s.Push(100);
        Console.WriteLine(s.Pop());
        Console.WriteLine(s.Pop());
        Console.WriteLine(s.Pop());
    }
}
```

Si le programme est stocké dans le fichier `test.cs` , lorsque `test.cs` est compilé, l' `acme.dll` assembly peut être référencé à l'aide de l'option du compilateur `/r` :

Console

```
csc /r:acme.dll test.cs
```

Cela permet de créer un assembly exécutable nommé `test.exe`, qui, lors de l'exécution, produit la sortie :

Console

```
100
10
1
```

C# permet le stockage du texte source d'un programme dans plusieurs fichiers source. Lorsqu'un programme C# multifichier est compilé, tous les fichiers source sont traités ensemble et les fichiers source peuvent librement se référencer mutuellement. Sur le plan conceptuel, c'est comme si tous les fichiers source étaient concaténés en un seul

fichier volumineux avant d'être traités. Déclarations anticipées ne sont jamais nécessaires en C#, car, à de très rares exceptions près, l'ordre de déclaration n'a pas d'importance. C# ne limite pas un fichier source à la déclaration d'un seul type public et ne nécessite pas non plus que le nom du fichier source corresponde à un type déclaré dans ce fichier.

Types et variables

Il existe deux genres de types en C# : *__types valeur__ _ et _ *types référence* *. Les variables des types valeur contiennent directement leurs données alors que les variables des types référence contiennent des références à leurs données, connues sous le nom d'objets. Avec les types référence, deux variables peuvent faire référence au même objet et, par conséquent, les opérations sur une variable peuvent affecter le même objet référencé par l'autre variable. Avec les types valeur, les variables possèdent leur propre copie de données, et les opérations sur une variable ne peuvent absolument pas affecter l'autre (sauf pour les variables de paramètre `ref` et `out`).

Les types valeur de C# sont encore divisés en ***types simples*** _, *_types ENUM_, types _struct_ et _types Nullable_, et les types référence de c# sont divisés en _types classe_, _types interface_, _types tableau*_ et _ *types délégués* *.

Le tableau suivant fournit une vue d'ensemble du système de type de C#.

Catégorie		Description
Types de valeur	Types simples	Entier signé : `sbyte`, `short`, `int`, `long`
		Entier non signé : `byte`, `ushort`, `uint`, `ulong`
		Caractères Unicode : `char`
		Virgule flottante IEEE : `float`, `double`
		Décimale haute précision :`decimal`
		Booléen : `bool`
	Types d'enum	Types définis par l'utilisateur de la forme `enum E {...}`
	Types struct	Types définis par l'utilisateur de la forme `struct S {...}`
	Types Nullable	Extensions de tous les autres types de valeurs avec une valeur `null`

Catégorie		Description
Types référence	Types de classe	Classe de base fondamentale de tous les autres types : `object`
		Chaînes Unicode : `string`
		Types définis par l'utilisateur de la forme `class C {...}`
	Types interface	Types définis par l'utilisateur de la forme `interface I {...}`
	Types de tableaux	Uni et multidimensionnels, par exemple `int[]` et `int[,]`
	Types délégués	Les types définis par l'utilisateur de la forme, par exemple `delegate int D(...)`

Les types intégraux huit prennent en charge les valeurs 8 bits, 16 bits, 32 bits et 64 bits sous forme signée ou non signée.

Les deux types à virgule flottante, `float` et `double` , sont représentés à l'aide des formats IEEE 754 simple précision et double précision 64 bits 32.

Le type `decimal` est un type de données 128 bits adapté aux calculs financiers et monétaires.

Le type de C# `bool` est utilisé pour représenter des valeurs booléennes, qui sont des valeurs `true` ou `false` .

Le traitement des caractères et chaînes dans le langage C# utilise l'encodage Unicode. Le type `char` représente une unité de code UTF-16, et le type `string` représente une séquence d'unités de code UTF-16.

Le tableau suivant récapitule les types numériques de C#.

Catégorie	Bits	Type	Plage/précision
Entier signé	8	`sbyte`	-128... 127
	16	`short`	-32768... 32, 767
	32	`int`	-2147483648... 2, 147, 483, 647
	64	`long`	-9223372036854775808... 9, 223, 372, 036, 854, 775, 807
Entier non signé	8	`byte`	0... 255

Catégorie	Bits	Type	Plage/précision
	16	`ushort`	0... 65, 535
	32	`uint`	0... 4, 294, 967, 295
	64	`ulong`	0... 18, 446, 744, 073, 709, 551, 615
Virgule flottante	32	`float`	1,5 × 10 ^ − 45 à 3,4 × 10 ^ 38, précision à 7 chiffres
	64	`double`	5,0 × 10 ^ − 324 à 1,7 × 10 ^ 308, précision de 15 chiffres
Decimal	128	`decimal`	1,0 × 10 ^ − 28 à 7,9 × 10 ^ 28, précision de 28 chiffres

Les programmes C# utilisent les ***déclarations de type*** pour créer de nouveaux types. Une déclaration de type spécifie le nom et les membres du nouveau type. Cinq des catégories de types C# sont définissables par l'utilisateur : types classe, types struct, types interface, types ENUM et types délégués.

Un type de classe définit une structure de données qui contient des données membres (champs) et des fonctions membres (méthodes, propriétés et autres). Les types de classes prennent en charge l'héritage unique et le polymorphisme, des mécanismes par lesquels les classes dérivées peuvent étendre et spécialiser les classes de base.

Un type struct est semblable à un type de classe, car il représente une structure avec des membres de données et des fonctions membres. Toutefois, contrairement aux classes, les structs sont des types valeur et ne nécessitent pas d'allocation de tas. Les types struct ne prennent pas en charge l'héritage spécifié par l'utilisateur, et tous les types struct héritent implicitement du type `object`.

Un type interface définit un contrat comme un jeu nommé de membres de fonction publics. Une classe ou un struct qui implémente une interface doit fournir des implémentations des membres de la fonction de l'interface. Une interface peut hériter de plusieurs interfaces de base, et une classe ou un struct peut implémenter plusieurs interfaces.

Un type délégué représente des références aux méthodes avec une liste de paramètres et un type de retour particuliers. Les délégués permettent de traiter les méthodes en tant qu'entités qui peuvent être affectées à des variables et passées comme paramètres. Les délégués sont similaires au concept de pointeurs de fonction dans d'autres langages, mais contrairement aux pointeurs de fonction, les délégués sont orientés objet et de type sécurisé.

Les types de classe, de struct, d'interface et de délégué prennent tous en charge les génériques, dans lesquels ils peuvent être paramétrés avec d'autres types.

Un type enum est un type distinct avec des constantes nommées. Chaque type enum a un type sous-jacent, qui doit être l'un des huit types intégraux. L'ensemble de valeurs d'un type enum est le même que l'ensemble de valeurs du type sous-jacent.

C# prend en charge les tableaux uni et multidimensionnels de tout type. Contrairement aux types mentionnés ci-dessus, les types de tableaux n'ont pas à être déclarés avant de pouvoir être utilisés. Au lieu de cela, les types de tableaux sont construits en ajoutant des crochets à un nom de type. Par exemple, `int[]` est un tableau unidimensionnel de `int` , `int[,]` est un tableau à deux dimensions de `int` , et `int[][]` est un tableau unidimensionnel de tableaux unidimensionnels de `int` .

Les types Nullable n'ont pas besoin d'être déclarés avant de pouvoir être utilisés. Pour chaque type valeur n'acceptant pas les valeurs null `T` , il existe un type Nullable correspondant `T?` qui peut contenir une valeur supplémentaire `null` . Par exemple, `int?` est un type qui peut contenir n'importe quel entier 32 bits ou la valeur `null` .

Le système de type C# est unifié de telle façon qu'une valeur de n'importe quel type peut être traitée en tant qu'objet. Chaque type dans C# dérive directement ou indirectement du type `object`, et `object` est la classe de base fondamentale de tous les types. Les valeurs des types référence sont considérées comme des objets simplement en affichant les valeurs en tant que type `object`. Les valeurs des types valeur sont traitées comme des objets en effectuant des opérations *__boxing__ _ et _ *_unboxing_**. Dans l'exemple suivant, une valeur `int` est convertie en `object` et à nouveau en `int`.

C#

```
using System;

class Test
{
    static void Main() {
        int i = 123;
        object o = i;          // Boxing
        int j = (int)o;        // Unboxing
    }
}
```

Quand une valeur d'un type valeur est convertie en type `object` , une instance d'objet, également appelée « Box », est allouée pour contenir la valeur, et la valeur est copiée dans cette zone. À l'inverse, lorsqu'une `object` référence est castée en un type valeur, une vérification est effectuée que l'objet référencé est une zone du type valeur correct et, si la vérification est réussie, la valeur dans la zone est copiée.

Le système de type unifié de C# signifie que les types valeur peuvent devenir des objets « à la demande ». En raison de l'unification, les bibliothèques à usage général qui utilisent le type `object` peuvent être utilisées avec les types référence et les types valeur.

Il existe plusieurs types de ***variables*** en C#, y compris les champs, les éléments de tableau, les variables locales et les paramètres. Les variables représentent des emplacements de stockage, et chaque variable a un type qui détermine les valeurs qui peuvent être stockées dans la variable, comme indiqué dans le tableau suivant.

Type de variable	Contenu possible
Type de valeur n'acceptant pas Null	Une valeur de ce type exact
Types valeur Nullable	Une valeur null ou une valeur de ce type exact
`object`	Une référence null, une référence à un objet de n'importe quel type référence ou une référence à une valeur boxed de n'importe quel type valeur
Type de classe	Une référence null, une référence à une instance de ce type de classe ou une référence à une instance d'une classe dérivée de ce type de classe
Type d'interface	Une référence null, une référence à une instance d'un type de classe qui implémente ce type d'interface, ou une référence à une valeur boxed d'un type valeur qui implémente ce type d'interface
Type tableau	Une référence null, une référence à une instance de ce type de tableau ou une référence à une instance d'un type de tableau compatible
Type délégué	Une référence null ou une référence à une instance de ce type délégué

Expressions

*Les **expressions** _ sont construites à partir d' _opérandes_ et d' _opérateurs*. and *operators*. Les opérateurs d'une expression indiquent les opérations à appliquer aux opérandes. Parmi les exemples d'opérateurs figurent `+`, `-`, `_`, `/` et `new`. Les littéraux, les champs, les variables locales et les expressions sont des exemples d'opérandes.

Lorsqu'une expression contient plusieurs opérateurs, le ***précédence** _ des opérateurs contrôle l'ordre dans lequel les opérateurs individuels sont évalués. Par exemple, l'expression `x + y _ z` est évaluée comme `x + (y * z)`, car l'opérateur `*` a une priorité plus élevée que `+`.

La plupart des opérateurs peuvent être ***surchargés***. La surcharge d'opérateur autorise la spécification d'implémentations d'opérateurs définies par l'utilisateur pour les opérations où l'un des deux opérandes ou les deux sont d'un type classe ou struct défini par l'utilisateur.

Le tableau suivant récapitule les opérateurs de C#, en répertoriant les catégories d'opérateurs par ordre de priorité, de la plus élevée à la plus faible. Les opérateurs d'une même catégorie ont une priorité identique.

Catégorie	Expression	Description
Principal	`x.m`	Accès au membre
	`x(...)`	Méthode et appel de délégué
	`x[...]`	Tableau et accès d'indexeur
	`x++`	Post-incrémentation
	`x--`	Post-décrémentation
	`new T(...)`	Création d'objet et de délégué
	`new T(...) {...}`	Création d'objet avec initialiseur
	`new {...}`	Initialiseur d'objet anonyme
	`new T[...]`	Création de tableau
	`typeof(T)`	Obtenir l'objet `System.Type` de `T`
	`checked(x)`	Évaluer l'expression dans le contexte vérifié (checked)
	`unchecked(x)`	Évaluer l'expression dans le contexte non vérifié (unchecked)
	`default(T)`	Obtenir la valeur par défaut du type `T`
	`delegate {...}`	Fonction anonyme (méthode anonyme)
Unaire	`+x`	Identité
	`-x`	Négation
	`!x`	Négation logique
	`~x`	Négation d'opération de bits

Catégorie	Expression	Description
	`++x`	Pré-incrémentation
	`--x`	Pré-décrémentation
	`(T)x`	Convertir explicitement `x` en type `T`
	`await x`	Attendre de façon asynchrone la fin de `x`
Multiplicatif	`x * y`	Multiplication
	`x / y`	Division
	`x % y`	Reste
Additive	`x + y`	Addition, concaténation de chaînes, combinaison de délégués
	`x - y`	Soustraction, suppression de délégué
Shift	`x << y`	Décalage à gauche
	`x >> y`	Décalage à droite
Relations et test de type	`x < y`	Inférieur à
	`x > y`	Supérieur à
	`x <= y`	Inférieur ou égal à
	`x >= y`	Supérieur ou égal à
	`x is T`	Retourne `true` si `x` est un `T`, `false` sinon
	`x as T`	Retourne `x` de type `T`, ou `null` si `x` n'est pas un `T`
Égalité	`x == y`	Égal à
	`x != y`	Différent de
ET logique	`x & y`	AND d'entiers au niveau du bit, AND logique booléen
XOR logique	`x ^ y`	Opération de bits entière XOR, Boolean logique XOR
OU logique	`x | y`	Opération de bits entière OR, Boolean logique OR
AND conditionnel	`x && y`	Évalue `y` uniquement si `x` est `true`
OR conditionnel	`x || y`	Évalue `y` uniquement si `x` est `false`

Catégorie	Expression	Description
Fusion de Null	`x ?? y`	Prend la valeur `y` si `x` est `null` , `x` sinon
Logique conditionnelle	`x ? y : z`	Prend la valeur `y` si `x` est `true`, `z` si `x` est `false`
Attribution ou fonction anonyme	`x = y`	Affectation
	`x op= y`	Assignation composée ; les opérateurs pris en charge `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` sont `&=` `^=` `|=`
	`(T x) => y`	Fonction anonyme (expression lambda)

Instructions

Les actions d'un programme sont exprimées à l'aide ***d'instructions***. C# prend en charge plusieurs types d'instructions, dont plusieurs sont définies en termes d'instructions incorporées.

Un ***bloc*** autorise l'écriture de plusieurs instructions dans les contextes où une seule instruction est autorisée. Un bloc se compose d'une liste d'instructions écrites entre les délimiteurs `{` et `}`.

Les ***instructions de déclaration*** sont utilisées pour déclarer des variables locales et des constantes.

Les ***instructions d'expression*** sont utilisées pour évaluer des expressions. Les expressions qui peuvent être utilisées comme instructions incluent les appels de méthode, les allocations d'objets à l'aide de l' `new` opérateur, les assignations utilisant `=` et les opérateurs d'assignation composée, les opérations d'incrémentation et de décrémentation à l'aide des `++` `--` opérateurs et et des expressions await.

Les ***instructions de sélection*** sont utilisées pour sélectionner un nombre d'instructions possibles pour l'exécution en fonction de la valeur d'une expression. Dans ce groupe, ce sont les instructions `if` et `switch`.

Les ***instructions d'itération*** sont utilisées pour exécuter une instruction incorporée de façon répétée. Dans ce groupe, ce sont les instructions `while`, `do`, `for` et `foreach`.

Les ***instructions de saut*** sont utilisées pour transférer le contrôle. Dans ce groupe, ce sont les instructions `break`, `continue`, `goto`, `throw`, `return` et `yield`.

L'instruction `try`...`catch` est utilisée pour intercepter les exceptions qui se produisent pendant l'exécution d'un bloc, et l'instruction `try`...`finally` permet de spécifier que le code de finalisation est toujours exécuté, qu'une exception se soit produite ou non.

Les `checked` `unchecked` instructions et sont utilisées pour contrôler le contexte de contrôle de dépassement de capacité pour les opérations arithmétiques de type intégral et les conversions.

L'instruction `lock` est utilisée pour obtenir le verrou d'exclusion mutuelle pour un objet donné, exécuter une instruction, puis libérer le verrou.

L'instruction `using` est utilisée pour obtenir une ressource, exécuter une instruction puis supprimer cette ressource.

Voici des exemples de chaque type d'instruction

Déclarations de variables locales

C#

```
static void Main() {
   int a;
   int b = 2, c = 3;
   a = 1;
   Console.WriteLine(a + b + c);
}
```

Déclaration de constante locale

C#

```
static void Main() {
    const float pi = 3.1415927f;
    const int r = 25;
    Console.WriteLine(pi * r * r);
}
```

Instruction d'expression

C#

```
static void Main() {
    int i;
    i = 123;                // Expression statement
    Console.WriteLine(i);   // Expression statement
    i++;                    // Expression statement
```

```
    Console.WriteLine(i);   // Expression statement
}
```

`if` gestion

C#

```
static void Main(string[] args) {
    if (args.Length == 0) {
        Console.WriteLine("No arguments");
    }
    else {
        Console.WriteLine("One or more arguments");
    }
}
```

`switch` gestion

C#

```
static void Main(string[] args) {
    int n = args.Length;
    switch (n) {
        case 0:
            Console.WriteLine("No arguments");
            break;
        case 1:
            Console.WriteLine("One argument");
            break;
        default:
            Console.WriteLine("{0} arguments", n);
            break;
    }
}
```

`while` gestion

C#

```
static void Main(string[] args) {
    int i = 0;
    while (i < args.Length) {
        Console.WriteLine(args[i]);
        i++;
    }
}
```

`do` gestion

C#

```
static void Main() {
    string s;
    do {
        s = Console.ReadLine();
        if (s != null) Console.WriteLine(s);
    } while (s != null);
}
```

`for` gestion

C#

```
static void Main(string[] args) {
    for (int i = 0; i < args.Length; i++) {
        Console.WriteLine(args[i]);
    }
}
```

`foreach` gestion

C#

```
static void Main(string[] args) {
    foreach (string s in args) {
        Console.WriteLine(s);
    }
}
```

`break` gestion

C#

```
static void Main() {
    while (true) {
        string s = Console.ReadLine();
        if (s == null) break;
        Console.WriteLine(s);
    }
}
```

`continue` gestion

C#

```
static void Main(string[] args) {
    for (int i = 0; i < args.Length; i++) {
```

```
        if (args[i].StartsWith("/")) continue;
        Console.WriteLine(args[i]);
    }
}
```

`goto` gestion

C#

```
static void Main(string[] args) {
    int i = 0;
    goto check;
    loop:
    Console.WriteLine(args[i++]);
    check:
    if (i < args.Length) goto loop;
}
```

`return` gestion

C#

```
static int Add(int a, int b) {
    return a + b;
}

static void Main() {
    Console.WriteLine(Add(1, 2));
    return;
}
```

`yield` gestion

C#

```
static IEnumerable<int> Range(int from, int to) {
    for (int i = from; i < to; i++) {
        yield return i;
    }
    yield break;
}

static void Main() {
    foreach (int x in Range(-10,10)) {
        Console.WriteLine(x);
    }
}
```

`throw` `try` instructions et

C#

```
static double Divide(double x, double y) {
    if (y == 0) throw new DivideByZeroException();
    return x / y;
}

static void Main(string[] args) {
    try {
        if (args.Length != 2) {
            throw new Exception("Two numbers required");
        }
        double x = double.Parse(args[0]);
        double y = double.Parse(args[1]);
        Console.WriteLine(Divide(x, y));
    }
    catch (Exception e) {
        Console.WriteLine(e.Message);
    }
    finally {
        Console.WriteLine("Good bye!");
    }
}
```

`checked` `unchecked` instructions et

C#

```
static void Main() {
    int i = int.MaxValue;
    checked {
        Console.WriteLine(i + 1);        // Exception
    }
    unchecked {
        Console.WriteLine(i + 1);        // Overflow
    }
}
```

`lock` gestion

C#

```
class Account
{
    decimal balance;
    public void Withdraw(decimal amount) {
        lock (this) {
            if (amount > balance) {
                throw new Exception("Insufficient funds");
            }
            balance -= amount;
```

```
        }
    }
}
```

`using` gestion

C#

```
static void Main() {
    using (TextWriter w = File.CreateText("test.txt")) {
        w.WriteLine("Line one");
        w.WriteLine("Line two");
        w.WriteLine("Line three");
    }
}
```

Classes et objets

*Les **classes** _ sont les types de C# les plus fondamentaux. Une classe est une structure de données qui combine l'état (champs) et les actions (méthodes et autres fonctions membres) en une seule unité. Une classe fournit une définition pour les *instances* créées dynamiquement de la classe, également appelées *objets*. Les classes prennent en charge *l'héritage* et le *polymorphisme*, des mécanismes par lesquels les *classes dérivées* peuvent étendre et spécialiser _ *classes de base* *.

Les nouvelles classes sont créées à l'aide des déclarations de classe. Une déclaration de classe commence par un en-tête qui spécifie les attributs et modificateurs de la classe, le nom de la classe, la classe de base (si fournie) et les interfaces implémentées par la classe. L'en-tête est suivi par le corps de la classe, qui se compose d'une liste de déclarations de membres écrites entre les délimiteurs `{` et `}`.

Voici une déclaration d'une classe simple nommée `Point` :

C#

```
public class Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Les instances de classes sont créées à l'aide de l'opérateur `new`, qui alloue la mémoire pour une nouvelle instance, appelle un constructeur pour initialiser l'instance et retourne une référence à l'instance. Les instructions suivantes créent deux `Point` objets et stockent les références à ces objets dans deux variables :

```
C#

Point p1 = new Point(0, 0);
Point p2 = new Point(10, 20);
```

La mémoire occupée par un objet est automatiquement récupérée lorsque l'objet n'est plus utilisé. Il n'est ni possible ni nécessaire de libérer explicitement des objets dans C#.

Membres

Les membres d'une classe sont *__membres statiques__ _ ou _ *_membres_ de l'instance * *. Les membres statiques appartiennent à des classes, et les membres d'instance appartiennent à des objets (instances de classes).

Le tableau suivant fournit une vue d'ensemble des types de membres qu'une classe peut contenir.

Membre	Description
Constantes	Valeurs constantes associées à la classe
Champs	Variables de la classe
Méthodes	Calculs et les actions qui peuvent être effectués par la classe
Propriétés	Actions associées à la lecture et l'écriture des propriétés nommées de la classe
Indexeurs	Actions liées à l'indexation des instances de la classe comme un tableau
Événements	Les notifications qui peuvent être générées par la classe
Opérateurs	Les opérateurs de conversion et d'expression pris en charge par la classe
Constructeurs	Les actions requises pour initialiser les instances de la classe ou la classe elle-même
Destructeurs	Actions à effectuer avant que les instances de la classe soient abandonnées de façon définitive
Types	Types imbriqués déclarés par la classe

Accessibilité

Chaque membre d'une classe a une accessibilité associée, qui contrôle les régions du texte du programme qui sont en mesure d'accéder au membre. Il existe cinq formes possibles d'accessibilité. Celles-ci sont récapitulées dans le tableau ci-dessous.

Accessibilité	Signification
`public`	Accès non limité
`protected`	Accès limité à cette classe ou aux classes dérivées de cette classe
`internal`	Accès limité à ce programme
`protected internal`	Accès limité à ce programme ou aux classes dérivées de cette classe
`private`	Accès limité à cette classe

Paramètres de type

Une définition de classe peut spécifier un jeu de paramètres de type en faisant suivre le nom de classe par une liste de noms de paramètre de type entre crochets. Les paramètres de type peuvent ensuite être utilisés dans le corps des déclarations de classe pour définir les membres de la classe. Dans l'exemple suivant, les paramètres de type de `Pair` sont `TFirst` et `TSecond` :

C#

```
public class Pair<TFirst,TSecond>
{
    public TFirst First;
    public TSecond Second;
}
```

Un type de classe déclaré pour prendre des paramètres de type est appelé un type de classe générique. Les types struct, interface et délégué peuvent également être génériques.

Lorsque la classe générique est utilisée, des arguments de type doivent être fournis pour chacun des paramètres de type :

C#

```
Pair<int,string> pair = new Pair<int,string> { First = 1, Second = "two" };
int i = pair.First;     // TFirst is int
string s = pair.Second; // TSecond is string
```

Un type générique avec des arguments de type fournis, comme `Pair<int,string>` ci-dessus, est appelé un type construit.

Classes de base

Une déclaration de classe peut spécifier une classe de base en faisant suivre les paramètres de nom et de type de classe par un signe deux-points et le nom de la classe de base. L'omission d'une spécification de classe de base revient à dériver du type `object`. Dans l'exemple suivant, la classe de base de `Point3D` est `Point`, et la classe de base de `Point` est `object` :

C#

```
public class Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

public class Point3D: Point
{
    public int z;

    public Point3D(int x, int y, int z): base(x, y) {
        this.z = z;
    }
}
```

Une classe hérite des membres de sa classe de base. L'héritage signifie qu'une classe contient implicitement tous les membres de sa classe de base, à l'exception de l'instance et des constructeurs statiques, et des destructeurs de la classe de base. Une classe dérivée peut ajouter des membres hérités, mais ne peut pas supprimer la définition d'un membre hérité. Dans l'exemple précédent, `Point3D` hérite des champs `x` et `y` de `Point`, et chaque instance de `Point3D` contient trois champs, `x`, `y` et `z`.

Il existe une conversion implicite d'un type de classe vers un de ses types de classe de base. Par conséquent, une variable d'un type de classe peut référencer une instance de cette classe ou une instance d'une classe dérivée. Par exemple, pour les déclarations de classe précédentes, une variable de type `Point` peut faire référence à un objet `Point` ou `Point3D` :

C#

```
Point a = new Point(10, 20);
Point b = new Point3D(10, 20, 30);
```

Champs

Un champ est une variable qui est associée à une classe ou une instance d'une classe.

Un champ déclaré avec le `static` modificateur définit un ***champ statique***. Un champ statique identifie exactement un seul emplacement de stockage. Quel que soit le nombre d'instances d'une classe qui sont créées, il n'existe qu'une seule copie d'un champ statique.

Un champ déclaré sans le `static` modificateur définit un ***champ d'instance***. Chaque instance d'une classe contient une copie distincte de tous les champs d'instance de cette classe.

Dans l'exemple suivant, chaque instance de la classe `Color` possède une copie distincte des champs d'instance `r`, `g` et `b`, mais il existe une seule copie des champs statiques `Black`, `White`, `Red`, `Green` et `Blue` :

C#

```
public class Color
{
    public static readonly Color Black = new Color(0, 0, 0);
    public static readonly Color White = new Color(255, 255, 255);
    public static readonly Color Red = new Color(255, 0, 0);
    public static readonly Color Green = new Color(0, 255, 0);
    public static readonly Color Blue = new Color(0, 0, 255);
    private byte r, g, b;

    public Color(byte r, byte g, byte b) {
        this.r = r;
        this.g = g;
        this.b = b;
    }
}
```

Comme indiqué dans l'exemple précédent, les ***champs en lecture seule*** peuvent être déclarés avec un modificateur `readonly`. L'assignation à un `readonly` champ ne peut se produire que dans le cadre de la déclaration du champ ou dans un constructeur de la même classe.

Méthodes

Une *__méthode__ _ est un membre qui implémente un calcul ou une action qui peut être effectuée par un objet ou une classe. Les *méthodes statiques* sont accessibles par le biais de la classe. _ Les *méthodes d'instance** sont accessibles via des instances de la classe.

Les méthodes ont une liste (éventuellement vide) de *__Parameters__, qui représentent des valeurs ou des références variables passées à la méthode, et un _ *type de retour* *, qui spécifie le type de la valeur calculée et retournée par la méthode. Le type de retour d'une méthode est `void` s'il ne retourne pas de valeur.

Comme les types, les méthodes peuvent également être un jeu de paramètres de type pour lesquels les arguments de type doivent être spécifiés lorsque la méthode est appelée. Contrairement aux types, les arguments de type peuvent souvent être déduits à partir des arguments d'un appel de méthode et n'ont pas à être fournis explicitement.

La ***signature*** d'une méthode doit être unique dans la classe dans laquelle la méthode est déclarée. La signature d'une méthode se compose du nom de la méthode, du nombre de paramètres de type et du nombre, des modificateurs et des types de ses paramètres. La signature d'une méthode n'inclut pas le type de retour.

Paramètres

Les paramètres sont utilisés pour passer des valeurs ou des références variables aux méthodes. Les paramètres d'une méthode obtiennent leurs valeurs réelles à partir des ***arguments*** qui sont spécifiés lorsque la méthode est appelée. Il existe quatre types de paramètres : les paramètres de valeur, les paramètres de référence, les paramètres de sortie et les tableaux de paramètres.

Un ***paramètre de valeur*** est utilisé pour le passage de paramètres d'entrée. Un paramètre de valeur correspond à une variable locale qui obtient sa valeur initiale de l'argument qui a été transmis pour le paramètre. Les modifications apportées à un paramètre de valeur n'affectent pas l'argument qui a été transmis pour le paramètre.

Les paramètres de valeur peuvent être facultatifs, en spécifiant une valeur par défaut afin que les arguments correspondants puissent être omis.

Un ***paramètre de référence*** est utilisé pour la transmission de paramètres en entrée et en sortie. L'argument passé pour un paramètre de référence doit être une variable, et pendant l'exécution de la méthode, le paramètre de référence représente le même emplacement de stockage que la variable d'argument. Un paramètre de référence est déclaré avec le modificateur `ref`. L'exemple suivant illustre l'utilisation des paramètres `ref`.

C#

```
using System;

class Test
{
    static void Swap(ref int x, ref int y) {
        int temp = x;
        x = y;
        y = temp;
    }

    static void Main() {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        Console.WriteLine("{0} {1}", i, j);            // Outputs "2 1"
    }
}
```

Un ***paramètre de sortie*** est utilisé pour le passage de paramètres de sortie. Un paramètre de sortie est similaire à un paramètre de référence, sauf que la valeur initiale de l'argument fourni par l'appelant n'a pas d'importance. Un paramètre de sortie est déclaré avec le modificateur `out`. L'exemple suivant illustre l'utilisation des paramètres `out`.

C#

```
using System;

class Test
{
    static void Divide(int x, int y, out int result, out int remainder) {
        result = x / y;
        remainder = x % y;
    }

    static void Main() {
        int res, rem;
        Divide(10, 3, out res, out rem);
        Console.WriteLine("{0} {1}", res, rem);    // Outputs "3 1"
    }
}
```

Un ***tableau de paramètres*** autorise le passage d'un nombre variable d'arguments à une méthode. Un tableau de paramètres est déclaré avec le modificateur `params`. Seul le dernier paramètre d'une méthode peut être un tableau de paramètres, et le type d'un tableau de paramètres doit être un type tableau unidimensionnel. Les `Write` `WriteLine`

méthodes et de la `System.Console` classe sont de bons exemples d'utilisation des tableaux de paramètres. Vous les déclarez de la façon suivante.

C#

```
public class Console
{
    public static void Write(string fmt, params object[] args) {...}
    public static void WriteLine(string fmt, params object[] args) {...}
    ...
}
```

Dans une méthode qui utilise un tableau de paramètres, le tableau de paramètres se comporte exactement comme un paramètre ordinaire de type tableau. Toutefois, dans un appel à une méthode avec un tableau de paramètres, il est possible de passer un argument unique de type tableau de paramètres ou n'importe quel nombre d'arguments du type d'élément du tableau de paramètres. Dans ce cas, une instance de tableau est automatiquement créée et initialisée avec les arguments donnés. Cet exemple

C#

```
Console.WriteLine("x={0} y={1} z={2}", x, y, z);
```

revient à écrire ce qui suit.

C#

```
string s = "x={0} y={1} z={2}";
object[] args = new object[3];
args[0] = x;
args[1] = y;
args[2] = z;
Console.WriteLine(s, args);
```

Corps de la méthode et variables locales

Le corps d'une méthode spécifie les instructions à exécuter lorsque la méthode est appelée.

Un corps de méthode peut déclarer des variables qui sont spécifiques à l'appel de la méthode. Ces variables sont appelées ***variables locales***. Une déclaration de variable locale spécifie un nom de type, un nom de variable et éventuellement une valeur initiale.

L'exemple suivant déclare une variable locale `i` avec une valeur initiale de zéro et une variable locale `j` sans valeur initiale.

C#

```
using System;

class Squares
{
    static void Main() {
        int i = 0;
        int j;
        while (i < 10) {
            j = i * i;
            Console.WriteLine("{0} x {0} = {1}", i, j);
            i = i + 1;
        }
    }
}
```

C# requiert qu'une variable locale soit ***assignée de manière définitive*** avant de pouvoir obtenir sa valeur. Par exemple, si la déclaration du `i` précédent n'inclut pas de valeur initiale, le compilateur signale une erreur pour les utilisations ultérieures de `i`, car `i` ne serait pas assigné de manière définitive à ces points dans le programme.

Une méthode peut utiliser les instructions `return` pour retourner le contrôle à son appelant. Dans une méthode retournant `void`, les instructions `return` ne peuvent pas spécifier une expression. Dans une méthode qui retourne des `void` instructions non-, `return` doit inclure une expression qui calcule la valeur de retour.

Méthodes statiques et d'instance

Une méthode déclarée avec un `static` modificateur est une ***méthode statique***. Une méthode statique n'opère pas sur une instance spécifique et permet uniquement d'accéder directement à des membres statiques.

Une méthode déclarée sans `static` modificateur est une ***méthode d'instance***. Une méthode d'instance opère sur une instance spécifique et peut accéder aux membres statiques et d'instance. L'instance sur laquelle une méthode d'instance a été appelée est explicitement accessible en tant que `this`. Une erreur consiste à faire référence à `this` dans une méthode statique.

La classe `Entity` suivante a à la fois des statiques et des membres d'instance.

C#

```
class Entity
{
    static int nextSerialNo;
    int serialNo;

    public Entity() {
        serialNo = nextSerialNo++;
    }

    public int GetSerialNo() {
        return serialNo;
    }

    public static int GetNextSerialNo() {
        return nextSerialNo;
    }

    public static void SetNextSerialNo(int value) {
        nextSerialNo = value;
    }
}
```

Chaque instance `Entity` contient un numéro de série (et probablement d'autres informations qui ne sont pas indiquées ici). Le constructeur `Entity` (qui est similaire à une méthode d'instance) initialise la nouvelle instance avec le numéro de série suivant. Étant donné que le constructeur est un membre d'instance, il est autorisé à accéder à la fois au champ d'instance `serialNo` et au champ statique `nextSerialNo`.

Les méthodes statiques `GetNextSerialNo` et `SetNextSerialNo` peuvent accéder au champ statique `nextSerialNo`, mais ce serait une erreur pour eux d'accéder directement au champ d'instance `serialNo`.

L'exemple suivant illustre l'utilisation de la `Entity` classe.

C#

```
using System;

class Test
{
    static void Main() {
        Entity.SetNextSerialNo(1000);
        Entity e1 = new Entity();
        Entity e2 = new Entity();
        Console.WriteLine(e1.GetSerialNo());           // Outputs "1000"
        Console.WriteLine(e2.GetSerialNo());           // Outputs "1001"
        Console.WriteLine(Entity.GetNextSerialNo());   // Outputs "1002"
    }
}
```

Notez que les méthodes statiques `SetNextSerialNo` et `GetNextSerialNo` sont appelées sur la classe alors que la méthode d'instance `GetSerialNo` est appelée sur les instances de la classe.

Méthodes virtuelles, de substitution et abstraites

Lorsqu'une déclaration de méthode d'instance comprend un `virtual` modificateur, la méthode est dite « ***méthode virtuelle** _ ». Si aucun `virtual` modificateur n'est présent, la méthode est dite *« _ » non virtuelle** *.

Quand une méthode virtuelle est appelée, le ***type au moment** de l'exécution _ de l'instance pour laquelle cet appel se produit détermine l'implémentation de méthode réelle à appeler. Dans un appel de méthode non virtuel, le _ *type au moment* de la compilation * de l'instance est le facteur déterminant.

Une méthode virtuelle peut être ***substituée*** dans une classe dérivée. Lorsqu'une déclaration de méthode d'instance comprend un `override` modificateur, la méthode substitue une méthode virtuelle héritée avec la même signature. Là où une déclaration de méthode virtuelle présente une nouvelle méthode, une déclaration de méthode de substitution spécialise une méthode virtuelle héritée existante en fournissant une nouvelle implémentation de cette méthode.

Une méthode ***abstraite*** est une méthode virtuelle sans implémentation. Une méthode abstraite est déclarée avec le `abstract` modificateur et est autorisée uniquement dans une classe qui est également déclarée `abstract` . Une méthode abstraite doit être remplacée dans chaque classe dérivée non abstraite.

L'exemple suivant déclare une classe abstraite, `Expression`, qui représente un nœud d'arborescence de l'expression et trois classes dérivées, `Constant`, `VariableReference` et `Operation`, qui implémentent des nœuds d'arborescence de l'expression pour les références variables, les constantes et les opérations arithmétiques. (Cela est similaire à, mais à ne pas confondre avec les types d'arborescence d'expression introduits dans les types d'arborescence d'expression).

C#

```
using System;
using System.Collections;

public abstract class Expression
{
    public abstract double Evaluate(Hashtable vars);
```

```
}

public class Constant: Expression
{
    double value;

    public Constant(double value) {
        this.value = value;
    }

    public override double Evaluate(Hashtable vars) {
        return value;
    }
}

public class VariableReference: Expression
{
    string name;

    public VariableReference(string name) {
        this.name = name;
    }

    public override double Evaluate(Hashtable vars) {
        object value = vars[name];
        if (value == null) {
            throw new Exception("Unknown variable: " + name);
        }
        return Convert.ToDouble(value);
    }
}

public class Operation: Expression
{
    Expression left;
    char op;
    Expression right;

    public Operation(Expression left, char op, Expression right) {
        this.left = left;
        this.op = op;
        this.right = right;
    }

    public override double Evaluate(Hashtable vars) {
        double x = left.Evaluate(vars);
        double y = right.Evaluate(vars);
        switch (op) {
            case '+': return x + y;
            case '-': return x - y;
            case '*': return x * y;
            case '/': return x / y;
        }
        throw new Exception("Unknown operator");
```

```
    }
}
```

Les quatre classes précédentes permet de modéliser des expressions arithmétiques. Par exemple, en utilisant des instances de ces classes, l'expression `x + 3` peut être représentée comme suit.

C#

```
Expression e = new Operation(
    new VariableReference("x"),
    '+',
    new Constant(3));
```

La méthode `Evaluate` d'une instance `Expression` est appelée pour évaluer l'expression donnée et produire une valeur `double`. La méthode prend comme argument un `Hashtable` qui contient des noms de variables (comme clés des entrées) et des valeurs (comme valeurs des entrées). La `Evaluate` méthode est une méthode abstraite virtuelle, ce qui signifie que les classes dérivées non abstraites doivent être substituées pour fournir une implémentation réelle.

Une implémentation de `Constant` de `Evaluate` renvoie simplement la constante stockée. L' `VariableReference` implémentation de A recherche le nom de variable dans la Hashtable et retourne la valeur résultante. Une implémentation de `Operation` évalue d'abord les opérandes de gauche et de droite (en appelant de manière récursive leurs méthodes `Evaluate`), puis effectue l'opération arithmétique donnée.

Le programme suivant utilise les classes `Expression` pour évaluer l'expression `x * (y + 2)` pour différentes valeurs de `x` et `y`.

C#

```
using System;
using System.Collections;

class Test
{
    static void Main() {
        Expression e = new Operation(
            new VariableReference("x"),
            '*',
            new Operation(
                new VariableReference("y"),
                '+',
                new Constant(2)
            )
```

```
        );
        Hashtable vars = new Hashtable();
        vars["x"] = 3;
        vars["y"] = 5;
        Console.WriteLine(e.Evaluate(vars));        // Outputs "21"
        vars["x"] = 1.5;
        vars["y"] = 9;
        Console.WriteLine(e.Evaluate(vars));        // Outputs "16.5"
    }
}
```

Surcharge de méthode

La méthode *__surcharge__ _ permet à plusieurs méthodes dans la même classe d'avoir le même nom, à condition qu'elles aient des signatures uniques. Lors de la compilation d'un appel d'une méthode surchargée, le compilateur utilise _ *Overload Resolution** pour déterminer la méthode spécifique à appeler. La résolution de surcharge trouve la méthode qui correspond le mieux aux arguments ou signale une erreur si aucune meilleure correspondance ne peut être trouvée. L'exemple suivant montre la résolution de surcharge en action. Le commentaire pour chaque appel de la méthode `Main` affiche une méthode qui est réellement appelée.

C#

```
class Test
{
    static void F() {
        Console.WriteLine("F()");
    }

    static void F(object x) {
        Console.WriteLine("F(object)");
    }

    static void F(int x) {
        Console.WriteLine("F(int)");
    }

    static void F(double x) {
        Console.WriteLine("F(double)");
    }

    static void F<T>(T x) {
        Console.WriteLine("F<T>(T)");
    }

    static void F(double x, double y) {
        Console.WriteLine("F(double, double)");
    }
```

```
    static void Main() {
        F();                // Invokes F()
        F(1);               // Invokes F(int)
        F(1.0);             // Invokes F(double)
        F("abc");           // Invokes F(object)
        F((double)1);       // Invokes F(double)
        F((object)1);       // Invokes F(object)
        F<int>(1);          // Invokes F<T>(T)
        F(1, 1);            // Invokes F(double, double)
    }
}
```

Comme le montre l'exemple, une méthode particulière peut toujours être sélectionnée en effectuant un cast explicite des arguments aux types de paramètres exacts et/ou en fournissant explicitement les arguments de type.

Autres fonctions membres

Les membres qui contiennent du code exécutable sont collectivement regroupés sous les ***membres de fonction*** d'une classe. La section précédente décrit les méthodes qui sont du type principal des fonctions membres. Cette section décrit les autres genres de fonctions membres pris en charge par C# : les constructeurs, les propriétés, les indexeurs, les événements, les opérateurs et les destructeurs.

Le code suivant illustre une classe générique appelée `List<T>` , qui implémente une liste d'objets pouvant être augmentée. La classe contient plusieurs exemples des types les plus courants de membres de fonction.

C#

```
public class List<T> {
    // Constant...
    const int defaultCapacity = 4;

    // Fields...
    T[] items;
    int count;

    // Constructors...
    public List(int capacity = defaultCapacity) {
        items = new T[capacity];
    }

    // Properties...
    public int Count {
        get { return count; }
    }
    public int Capacity {
```

```
        get {
            return items.Length;
        }
        set {
            if (value < count) value = count;
            if (value != items.Length) {
                T[] newItems = new T[value];
                Array.Copy(items, 0, newItems, 0, count);
                items = newItems;
            }
        }
    }

    // Indexer...
    public T this[int index] {
        get {
            return items[index];
        }
        set {
            items[index] = value;
            OnChanged();
        }
    }

    // Methods...
    public void Add(T item) {
        if (count == Capacity) Capacity = count * 2;
        items[count] = item;
        count++;
        OnChanged();
    }
    protected virtual void OnChanged() {
        if (Changed != null) Changed(this, EventArgs.Empty);
    }
    public override bool Equals(object other) {
        return Equals(this, other as List<T>);
    }
    static bool Equals(List<T> a, List<T> b) {
        if (a == null) return b == null;
        if (b == null || a.count != b.count) return false;
        for (int i = 0; i < a.count; i++) {
            if (!object.Equals(a.items[i], b.items[i])) {
                return false;
            }
        }
        return true;
    }

    // Event...
    public event EventHandler Changed;

    // Operators...
    public static bool operator ==(List<T> a, List<T> b) {
        return Equals(a, b);
    }
```

```
    public static bool operator !=(List<T> a, List<T> b) {
        return !Equals(a, b);
    }
}
```

Constructeurs

C# prend en charge les constructeurs statiques et d'instance. Un *__constructeur d'instance__ _ est un membre qui implémente les actions requises pour initialiser une instance d'une classe. Un *constructeur _ static** est un membre qui implémente les actions requises pour initialiser une classe lorsqu'elle est chargée pour la première fois.

Un constructeur est déclaré comme une méthode sans aucun type de retour et le même nom que la classe contenante. Si une déclaration de constructeur comprend un `static` modificateur, elle déclare un constructeur statique. Dans le cas contraire, elle déclare un constructeur d'instance.

Les constructeurs d'instance peuvent être surchargés. Par exemple, la classe `List<T>` déclare deux constructeurs d'instance, un sans paramètres et une fonction qui accepte un paramètre `int`. Les constructeurs d'instance sont appelés en utilisant l'opérateur `new`. Les instructions suivantes allouent deux `List<string>` instances à l'aide de chacun des constructeurs de la `List` classe.

```
C#

List<string> list1 = new List<string>();
List<string> list2 = new List<string>(10);
```

Contrairement aux autres membres, les constructeurs d'instance ne sont pas hérités, et une classe n'a aucun constructeur d'instance autre que ceux réellement déclarés dans la classe. Si aucun constructeur d'instance n'est fourni pour une classe, un constructeur vide sans paramètre est fourni automatiquement.

Propriétés

*__Properties__ _ sont une extension naturelle des champs. Les deux sont des membres nommés avec des types associés, et la syntaxe pour accéder aux champs et propriétés est la même. Toutefois, contrairement aux champs, les propriétés ne désignent pas des emplacements de stockage. Au lieu de cela, les propriétés ont des _ *accesseurs** qui spécifient les instructions à exécuter lorsque leurs valeurs sont lues ou écrites.

Une propriété est déclarée comme un champ, sauf que la déclaration se termine par un `get` accesseur et/ou un `set` accesseur écrit entre les délimiteurs `{` et `}` au lieu de se terminer par un point-virgule. Une propriété qui possède à la fois un `get` accesseur et un `set` accesseur est une propriété **en lecture-écriture** , *une propriété qui a uniquement un `get` accesseur est une propriété en lecture seule, et une propriété qui a uniquement un `set` accesseur est une propriété _ * en écriture seule * *.*

Un `get` accesseur correspond à une méthode sans paramètre avec une valeur de retour du type de propriété. À l'exception de la cible d'une assignation, lorsqu'une propriété est référencée dans une expression, l' `get` accesseur de la propriété est appelé pour calculer la valeur de la propriété.

Un `set` accesseur correspond à une méthode avec un paramètre unique nommé `value` et aucun type de retour. Lorsqu'une propriété est référencée comme cible d'une assignation ou comme opérande de `++` ou `--` , l' `set` accesseur est appelé avec un argument qui fournit la nouvelle valeur.

La classe `List<T>` déclare deux propriétés, `Count` et `Capacity`, qui sont respectivement en lecture seule et en lecture-écriture. Voici un exemple d'utilisation de ces propriétés.

C#

```
List<string> names = new List<string>();
names.Capacity = 100;           // Invokes set accessor
int i = names.Count;            // Invokes get accessor
int j = names.Capacity;         // Invokes get accessor
```

C# prend en charge les propriétés d'instance et les propriétés statiques, similaires aux champs et aux méthodes. Les propriétés statiques sont déclarées avec le `static` modificateur, et les propriétés d'instance sont déclarées sans lui.

Le ou les accesseurs d'une propriété peuvent être virtuels. Lorsqu'une déclaration de propriété inclut un modificateur `virtual`, `abstract` ou `override`, elle s'applique aux accesseurs de la propriété.

Indexeurs

Un ***indexeur*** est un membre qui permet l'indexation des objets de la même façon en tant que tableau. Un indexeur est déclaré comme une propriété, sauf que le nom du membre est `this`, suivi d'une liste de paramètres écrits entre les délimiteurs `[` et `]`. Les paramètres sont disponibles dans le ou les accesseurs de l'indexeur. Similaires aux

propriétés, les indexeurs peuvent être en lecture-écriture, en lecture seule et en écriture seule, et les accesseurs d'un indexeur peuvent être virtuels.

La classe `List` déclare un indexeur en lecture-écriture unique qui prend un paramètre `int`. L'indexeur rend possible l'indexation des instances `List` avec des valeurs `int`. Par exemple

```
C#

List<string> names = new List<string>();
names.Add("Liz");
names.Add("Martha");
names.Add("Beth");
for (int i = 0; i < names.Count; i++) {
    string s = names[i];
    names[i] = s.ToUpper();
}
```

Les indexeurs peuvent être surchargés, ce qui signifie qu'une classe peut déclarer plusieurs indexeurs tant que le nombre ou les types de leurs paramètres diffèrent.

Événements

Un ***événement*** est un membre qui permet à une classe ou un objet de fournir des notifications. Un événement est déclaré comme un champ, sauf que la déclaration comprend un `event` mot clé et que le type doit être un type délégué.

Dans une classe qui déclare un membre d'événement, l'événement se comporte comme un champ d'un type délégué (à condition que l'événement n'est pas abstrait et ne déclare pas d'accesseurs). Le champ stocke une référence à un délégué qui représente les gestionnaires d'événements qui ont été ajoutés à l'événement. Si aucun descripteur d'événement n'est présent, le champ est `null` .

La classe `List<T>` déclare un membre d'événement unique appelé `Changed`, qui indique qu'un nouvel élément a été ajouté à la liste. L' `Changed` événement est déclenché par la `OnChanged` méthode virtuelle, qui vérifie d'abord si l'événement est `null` (ce qui signifie qu'aucun gestionnaire n'est présent). La notion de déclenchement d'un événement est équivalente à l'appel de délégué représenté par l'événement. Par conséquent, il n'existe aucune construction de langage particulière pour déclencher des événements.

Les clients réagissent aux événements via les ***gestionnaires d'événements***. Les gestionnaires d'événements sont joints à l'aide de l'opérateur `+=` et supprimés à l'aide

de l'opérateur `-=`. L'exemple suivant joint un gestionnaire d'événements à l'événement `Changed` d'un `List<string>`.

C#

```
using System;

class Test
{
    static int changeCount;

    static void ListChanged(object sender, EventArgs e) {
        changeCount++;
    }

    static void Main() {
        List<string> names = new List<string>();
        names.Changed += new EventHandler(ListChanged);
        names.Add("Liz");
        names.Add("Martha");
        names.Add("Beth");
        Console.WriteLine(changeCount);        // Outputs "3"
    }
}
```

Pour les scénarios avancés où le contrôle du stockage sous-jacent d'un événement est souhaité, une déclaration d'événement peut fournir explicitement des accesseurs `add` et `remove`, qui sont plutôt similaires à l'accesseur `set` d'une propriété.

Opérateurs

Un ***opérateur*** est un membre qui définit la signification de l'application d'un opérateur d'expression particulière aux instances d'une classe. Trois types d'opérateurs peuvent être définis : les opérateurs unaires, les opérateurs binaires et les opérateurs de conversion. Tous les opérateurs doivent être déclarés comme `public` et `static`.

La classe `List<T>` déclare deux opérateurs, `operator==` et `operator!=`, et donne donc une nouvelle signification aux expressions qui appliquent ces opérateurs aux instances `List`. Plus précisément, les opérateurs définissent l'égalité de deux `List<T>` instances en comparant chacun des objets contenus à l'aide de leurs `Equals` méthodes. L'exemple suivant utilise l'opérateur `==` pour comparer deux instances de `List<int>`.

C#

```
using System;
```

```
class Test
{
    static void Main() {
        List<int> a = new List<int>();
        a.Add(1);
        a.Add(2);
        List<int> b = new List<int>();
        b.Add(1);
        b.Add(2);
        Console.WriteLine(a == b);        // Outputs "True"
        b.Add(3);
        Console.WriteLine(a == b);        // Outputs "False"
    }
}
```

La première `Console.WriteLine` génère `True`, car les deux listes contiennent le même nombre d'objets avec les mêmes valeurs dans le même ordre. Si `List<T>` n'avait pas défini `operator==`, la première `Console.WriteLine` aurait affiché `False`, car `a` et `b` référencent des instances `List<int>` différentes.

Destructeurs

Un ***destructeur*** est un membre qui implémente les actions requises pour détruire une instance d'une classe. Les destructeurs ne peuvent pas avoir de paramètres, ils ne peuvent pas avoir de modificateurs d'accessibilité et ils ne peuvent pas être appelés explicitement. Le destructeur d'une instance est appelé automatiquement pendant garbage collection.

Le garbage collector est autorisé à atteindre une grande latitude en déterminant quand collecter des objets et exécuter des destructeurs. Plus précisément, le minutage des appels du destructeur n'est pas déterministe et les destructeurs peuvent être exécutés sur n'importe quel thread. Pour ces raisons, les classes doivent implémenter des destructeurs uniquement quand aucune autre solution n'est possible.

L'instruction `using` fournit une meilleure approche pour la destruction d'objets.

Structures

Comme les classes, les ***structs*** sont des structures de données qui peuvent contenir des membres de données et des fonctions membres. Mais contrairement aux classes, les structures sont des types valeur et ne nécessitent pas d'allocation de tas. Une variable de type struct stocke directement les données de la structure, alors qu'une variable de type class stocke une référence à un objet alloué dynamiquement. Les types struct ne

prennent pas en charge l'héritage spécifié par l'utilisateur, et tous les types struct héritent implicitement du type `object`.

Les structures sont particulièrement utiles pour les petites structures de données qui ont une sémantique par rapport à leurs valeurs. Les nombres complexes, les points dans un système de coordonnées ou les paires clé-valeur dans un dictionnaire sont de bons exemples de structures. L'utilisation de structures plutôt que de classes pour les petites structures de données peut faire une grande différence dans le nombre d'allocations de mémoire effectuées par une application. Par exemple, le programme suivant crée et initialise un tableau de 100 points. Avec `Point` implémenté en tant que classe, 101 objets séparés sont instanciés : un pour le tableau et un pour chacun des 100 éléments.

C#

```
class Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

class Test
{
    static void Main() {
        Point[] points = new Point[100];
        for (int i = 0; i < 100; i++) points[i] = new Point(i, i);
    }
}
```

Une alternative consiste à créer `Point` un struct.

C#

```
struct Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Maintenant, un seul objet est instancié, celui du tableau, et les instances `Point` sont stockées à la suite dans le tableau.

Vous appelez les constructeurs de struct avec l'opérateur `new`, mais cela n'implique pas que de la mémoire soit allouée. Au lieu d'allouer dynamiquement un objet et de renvoyer une référence à cet objet, un constructeur de struct retourne simplement la valeur du struct (généralement dans un emplacement temporaire sur la pile), et cette valeur est ensuite copiée si nécessaire.

Avec les classes, deux variables peuvent faire référence au même objet et, par conséquent, les opérations sur une variable peuvent affecter le même objet référencé par l'autre variable. Avec les structs, les variables disposent chacune de leur propre copie des données, et il n'est pas possible pour les opérations sur une d'affecter les autres. Par exemple, la sortie produite par le fragment de code suivant varie selon qu'il `Point` s'agit d'une classe ou d'un struct.

```
C#

Point a = new Point(10, 10);
Point b = a;
a.x = 20;
Console.WriteLine(b.x);
```

Si `Point` est une classe, la sortie est `20` due à `a` et `b` référence le même objet. Si `Point` est un struct, la sortie est `10` due au fait que l'assignation de `a` à `b` crée une copie de la valeur et que cette copie n'est pas affectée par l'assignation suivante à `a.x` .

L'exemple précédent illustre deux des limitations des structs. Tout d'abord, la copie d'un struct entier est généralement moins efficace que la copie d'une référence d'objet, aussi le passage de paramètres d'affectation et de valeur peut être plus coûteux avec les structures qu'avec les types référence. Ensuite, à l'exception des paramètres `ref` et `out`, il n'est pas possible de créer des références aux structures, ce qui rend leur utilisation impossible dans un certain nombre de situations.

Tableaux

Un *__tableau__ _ est une structure de données qui contient un certain nombre de variables accessibles par le biais d'index calculés. Les variables contenues dans un tableau, également appelées *_éléments_* du tableau, sont toutes du même type, et ce type est appelé « type d'*élément*» * du tableau.elements*_ of the array, are all of the same type, and this type is called the _ *element type* of the array.

Les types tableau sont des types référence, et la déclaration d'une variable tableau réserve simplement un espace pour une référence à une instance de tableau. Les instances de tableau réelles sont créées dynamiquement au moment de l'exécution à

l'aide de l' `new` opérateur. L' `new` opération spécifie la ***longueur*** de la nouvelle instance de tableau, qui est ensuite résolue pour la durée de vie de l'instance. Les indices des éléments d'un tableau vont de `0` à `Length - 1`. L'opérateur `new` initialise automatiquement les éléments d'un tableau à leur valeur par défaut, c'est-à-dire, par exemple, zéro pour tous les types numériques et `null` pour tous les types référence.

L'exemple suivant crée un tableau de `int` éléments, initialise le tableau et imprime le contenu du tableau.

C#

```
using System;

class Test
{
    static void Main() {
        int[] a = new int[10];
        for (int i = 0; i < a.Length; i++) {
            a[i] = i * i;
        }
        for (int i = 0; i < a.Length; i++) {
            Console.WriteLine("a[{0}] = {1}", i, a[i]);
        }
    }
}
```

Cet exemple crée et opère sur un ***tableau unidimensionnel** _. C# prend également en charge *les tableaux multidimensionnels*. Le nombre de dimensions d'un type tableau, également appelé _ *Rank** du type tableau, est un plus le nombre de virgules écrites entre les crochets du type tableau. L'exemple suivant alloue un tableau unidimensionnel, à deux dimensions et à trois dimensions.

C#

```
int[] a1 = new int[10];
int[,] a2 = new int[10, 5];
int[,,] a3 = new int[10, 5, 2];
```

Le tableau `a1` contient 10 éléments, le tableau `a2` en contient 50 (10 x 5) et le tableau `a3` en contient 100 (10 × 5 × 2).

Le type d'élément d'un tableau peut être de n'importe quel type, y compris un type tableau. Un tableau avec des éléments d'un type tableau est parfois appelé un ***tableau en escalier***, car les longueurs des tableaux d'éléments ne sont pas nécessairement pas les mêmes. L'exemple suivant alloue un tableau de tableaux de `int` :

C#

```
int[][] a = new int[3][];
a[0] = new int[10];
a[1] = new int[5];
a[2] = new int[20];
```

La première ligne crée un tableau avec trois éléments, chacun de type `int[]` et chacun avec une valeur initiale de `null`. Les lignes suivantes initialisent ensuite les trois éléments avec des références aux instances individuelles de tableau de longueur variable.

L' `new` opérateur permet de spécifier les valeurs initiales des éléments du tableau à l'aide d'un ***initialiseur de tableau***, qui est une liste d'expressions écrites entre les délimiteurs `{` et `}` . L'exemple suivant alloue et initialise un `int[]` avec trois éléments.

C#

```
int[] a = new int[] {1, 2, 3};
```

Notez que la longueur du tableau est déduite du nombre d'expressions entre `{` et `}` . Les déclarations locales des champs et variables peuvent être abrégées davantage afin de ne pas avoir à redéclarer le type tableau.

C#

```
int[] a = {1, 2, 3};
```

Les deux exemples précédents sont équivalents à ce qui suit :

C#

```
int[] t = new int[3];
t[0] = 1;
t[1] = 2;
t[2] = 3;
int[] a = t;
```

Interfaces

Une ***interface*** définit un contrat qui peut être implémenté par des classes et structures. Une interface peut contenir des méthodes, des propriétés, des événements et des indexeurs. Une interface ne fournit pas les implémentations des membres qu'elle définit,

elle indique simplement les membres qui doivent être fournis par les classes ou les structs qui implémentent l'interface.

Les interfaces peuvent utiliser ***l'héritage multiple***. Dans l'exemple suivant, l'interface `IComboBox` hérite à la fois de `ITextBox` et `IListBox`.

C#

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

interface IListBox: IControl
{
    void SetItems(string[] items);
}

interface IComboBox: ITextBox, IListBox {}
```

Les classes et les structs peuvent implémenter plusieurs interfaces. Dans l'exemple suivant, la classe `EditBox` implémente à la fois `IControl` et `IDataBound`.

C#

```
interface IDataBound
{
    void Bind(Binder b);
}

public class EditBox: IControl, IDataBound
{
    public void Paint() {...}
    public void Bind(Binder b) {...}
}
```

Lorsqu'une classe ou un struct implémente une interface spécifique, les instances de cette classe ou struct peuvent être converties implicitement en ce type d'interface. Par exemple

C#

```
EditBox editBox = new EditBox();
IControl control = editBox;
```

```
IDataBound dataBound = editBox;
```

Si une instance n'est pas connue pour implémenter une interface donnée de façon statique, des casts de type dynamiques peuvent être utilisés. Par exemple, les instructions suivantes utilisent des casts de type dynamique pour obtenir `IControl` les implémentations d'interface et d'objet `IDataBound` . Étant donné que le type réel de l'objet est `EditBox` , les casts sont correctement exécutés.

C#

```
object obj = new EditBox();
IControl control = (IControl)obj;
IDataBound dataBound = (IDataBound)obj;
```

Dans la `EditBox` classe précédente, la `Paint` méthode de l' `IControl` interface et la `Bind` méthode de l' `IDataBound` interface sont implémentées à l'aide de `public` membres. C# prend également en charge les ***implémentations de membres d'interface explicites***, à l'aide desquelles la classe ou le struct peuvent éviter de créer les membres `public` . Une implémentation de membre d'interface explicite est écrite à l'aide du nom de membre d'interface qualifié complet. Par exemple, la classe `EditBox` peut implémenter les méthodes `IControl.Paint` et `IDataBound.Bind` à l'aide des implémentations de membres d'interface explicites, comme suit.

C#

```
public class EditBox: IControl, IDataBound
{
    void IControl.Paint() {...}
    void IDataBound.Bind(Binder b) {...}
}
```

Les membres d'interface explicites sont accessibles uniquement via le type interface. Par exemple, l'implémentation de `IControl.Paint` fournie par la `EditBox` classe précédente ne peut être appelée qu'en convertissant d'abord la `EditBox` référence au `IControl` type d'interface.

C#

```
EditBox editBox = new EditBox();
editBox.Paint();                        // Error, no such method
IControl control = editBox;
control.Paint();                        // Ok
```

Énumérations

Un ***type enum*** est un type valeur distinct avec un ensemble de constantes nommées. L'exemple suivant déclare et utilise un type Enum nommé `Color` avec trois valeurs constantes, `Red` , `Green` et `Blue` .

C#

```
using System;

enum Color
{
    Red,
    Green,
    Blue
}

class Test
{
    static void PrintColor(Color color) {
        switch (color) {
            case Color.Red:
                Console.WriteLine("Red");
                break;
            case Color.Green:
                Console.WriteLine("Green");
                break;
            case Color.Blue:
                Console.WriteLine("Blue");
                break;
            default:
                Console.WriteLine("Unknown color");
                break;
        }
    }

    static void Main() {
        Color c = Color.Red;
        PrintColor(c);
        PrintColor(Color.Blue);
    }
}
```

Chaque type enum a un type intégral correspondant appelé le ***type sous-jacent*** du type enum. Un type enum qui ne déclare pas explicitement un type sous-jacent a un type sous-jacent de `int` . Le format de stockage d'un type enum et la plage de valeurs possibles sont déterminés par son type sous-jacent. L'ensemble de valeurs qu'un type enum peut utiliser n'est pas limité par ses membres enum. En particulier, toute valeur du

type sous-jacent d'une énumération peut être convertie en type enum et est une valeur valide distincte de ce type enum.

L'exemple suivant déclare un type Enum nommé `Alignment` avec un type sous-jacent de `sbyte` .

C#

```
enum Alignment: sbyte
{
    Left = -1,
    Center = 0,
    Right = 1
}
```

Comme indiqué dans l'exemple précédent, une déclaration de membre enum peut inclure une expression constante qui spécifie la valeur du membre. La valeur de constante pour chaque membre enum doit être comprise dans la plage du type sous-jacent de l'enum. Lorsqu'une déclaration de membre enum ne spécifie pas explicitement de valeur, le membre reçoit la valeur zéro (s'il s'agit du premier membre du type enum) ou la valeur du membre enum qui précède textuel plus un.

Les valeurs enum peuvent être converties en valeurs intégrales et vice versa à l'aide de casts de type. Par exemple

C#

```
int i = (int)Color.Blue;        // int i = 2;
Color c = (Color)2;             // Color c = Color.Blue;
```

La valeur par défaut d'un type enum est la valeur intégrale zéro convertie en type enum. Dans les cas où les variables sont automatiquement initialisées à une valeur par défaut, il s'agit de la valeur donnée aux variables de types ENUM. Pour que la valeur par défaut d'un type enum soit facilement disponible, le littéral `0` convertit implicitement en un type enum. Ce qui suit est donc autorisé.

C#

```
Color c = 0;
```

Délégués

Un ***type délégué*** représente des références aux méthodes avec une liste de paramètres et un type de retour particuliers. Les délégués permettent de traiter les méthodes en tant qu'entités qui peuvent être affectées à des variables et passées comme paramètres. Les délégués sont similaires au concept de pointeurs de fonction dans d'autres langages, mais contrairement aux pointeurs de fonction, les délégués sont orientés objet et de type sécurisé.

L'exemple suivant déclare et utilise un type délégué nommé `Function`.

C#

```
using System;

delegate double Function(double x);

class Multiplier
{
    double factor;

    public Multiplier(double factor) {
        this.factor = factor;
    }

    public double Multiply(double x) {
        return x * factor;
    }
}

class Test
{
    static double Square(double x) {
        return x * x;
    }

    static double[] Apply(double[] a, Function f) {
        double[] result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }

    static void Main() {
        double[] a = {0.0, 0.5, 1.0};
        double[] squares = Apply(a, Square);
        double[] sines = Apply(a, Math.Sin);
        Multiplier m = new Multiplier(2.0);
        double[] doubles =  Apply(a, m.Multiply);
    }
}
```

Une instance du type délégué `Function` peut faire référence à toute méthode qui accepte un argument `double` et retourne une valeur `double`. La `Apply` méthode applique un donné `Function` aux éléments d'un `double[]` , en retournant un `double[]` avec les résultats. Dans la méthode `Main`, `Apply` sert à appliquer trois fonctions différentes à un `double[]`.

Un délégué peut faire référence à une méthode statique (comme `Square` ou `Math.Sin` dans l'exemple précédent) ou à une méthode d'instance (comme `m.Multiply` dans l'exemple précédent). Un délégué qui référence une méthode d'instance référence aussi un objet particulier, et quand la méthode d'instance est appelée via le délégué, cet objet devient `this` dans l'appel.

Les délégués peuvent également être créés à l'aide de fonctions anonymes, qui sont des « méthodes inline » créées à la volée. Les fonctions anonymes peuvent voir les variables locales des méthodes qui l'entourent. Ainsi, l'exemple de multiplicateur ci-dessus peut être écrit plus facilement sans utiliser une `Multiplier` classe :

```
C#

double[] doubles =  Apply(a, (double x) => x * 2.0);
```

Une propriété intéressante et utile d'un délégué est qu'il ne connaît pas la classe de la méthode qu'il référence, et cela lui est égal. Tout ce qui importe est que la méthode référencée ait les mêmes paramètres et type de retour que le délégué.

Attributs

Les types, membres et autres entités d'un programme C# prennent en charge les modificateurs qui contrôlent certains aspects de leur comportement. Par exemple, l'accessibilité d'une méthode est contrôlée à l'aide des modificateurs `public`, `protected`, `internal` et `private`. C# généralise cette fonctionnalité pour que les types d'informations déclaratives définis par l'utilisateur puissent être associés aux entités de programme et récupérés au moment de l'exécution. Les programmes spécifient ces informations déclaratives supplémentaires en définissant et en utilisant les ***attributs***.

L'exemple suivant déclare un attribut `HelpAttribute` qui peut être placé sur des entités de programme pour fournir des liens vers leur documentation associée.

```
C#

using System;
```

```
public class HelpAttribute: Attribute
{
    string url;
    string topic;

    public HelpAttribute(string url) {
        this.url = url;
    }

    public string Url {
        get { return url; }
    }

    public string Topic {
        get { return topic; }
        set { topic = value; }
    }
}
```

Toutes les classes d'attributs dérivent de la `System.Attribute` classe de base fournie par l' .NET Framework. Les attributs peuvent être appliqués en donnant leur nom, ainsi que les éventuels arguments, à l'intérieur de crochets juste avant la déclaration associée. Si le nom d'un attribut se termine par `Attribute` , cette partie du nom peut être omise lorsque l'attribut est référencé. Par exemple, l'attribut `HelpAttribute` peut être utilisé comme suit.

C#

```
[Help("http://msdn.microsoft.com/.../MyClass.htm")]
public class Widget
{
    [Help("http://msdn.microsoft.com/.../MyClass.htm", Topic = "Display")]
    public void Display(string text) {}
}
```

Cet exemple attache un `HelpAttribute` à la `Widget` classe et un autre `HelpAttribute` à la `Display` méthode dans la classe. Les constructeurs publics d'une classe d'attributs contrôlent les informations qui doivent être fournies lorsque l'attribut est joint à une entité de programme. Des informations supplémentaires peuvent être fournies en référençant les propriétés en lecture-écriture publiques de la classe d'attributs (comme la référence à la propriété `Topic`, précédemment).

L'exemple suivant montre comment récupérer des informations d'attribut pour une entité de programme donnée au moment de l'exécution à l'aide de la réflexion.

C#

```
using System;
using System.Reflection;

class Test
{
    static void ShowHelp(MemberInfo member) {
        HelpAttribute a = Attribute.GetCustomAttribute(member,
            typeof(HelpAttribute)) as HelpAttribute;
        if (a == null) {
            Console.WriteLine("No help for {0}", member);
        }
        else {
            Console.WriteLine("Help for {0}:", member);
            Console.WriteLine("  Url={0}, Topic={1}", a.Url, a.Topic);
        }
    }

    static void Main() {
        ShowHelp(typeof(Widget));
        ShowHelp(typeof(Widget).GetMethod("Display"));
    }
}
```

Lorsqu'un attribut particulier est demandé par réflexion, le constructeur de la classe d'attributs est appelé avec les informations fournies dans la source du programme, et l'instance d'attribut qui en résulte est retournée. Si des informations supplémentaires ont été fournies via les propriétés, ces propriétés sont définies sur les valeurs données avant que le renvoi de l'instance de l'attribut.

1 Scope

Article • 01/13/2022

This specification describes the form and establishes the interpretation of programs written in the C# programming language. It describes

- The representation of C# programs;
- The syntax and constraints of the C# language;
- The semantic rules for interpreting C# programs;
- The restrictions and limits imposed by a conforming implementation of C#.

This specification does not describe

- The mechanism by which C# programs are transformed for use by a data-processing system;
- The mechanism by which C# applications are invoked for use by a data-processing system;
- The mechanism by which input data are transformed for use by a C# application;
- The mechanism by which output data are transformed after being produced by a C# application;
- The size or complexity of a program and its data that will exceed the capacity of any specific data-processing system or the capacity of a particular processor;
- All minimal requirements of a data-processing system that is capable of supporting a conforming implementation.

2 Normative references

Article • 03/18/2022

The following normative documents contain provisions, which, through reference in this text, constitute provisions of this specification. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this specification are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies. Members of ISO and IEC maintain registers of currently valid specifications.

ISO/IEC 23271:2012, *Common Language Infrastructure (CLI), Partition IV: Base Class Library (BCL), Extended Numerics Library, and Extended Array Library.*

ISO 80000-2, *Quantities and units — Part 2: Mathematical signs and symbols to be used in the natural sciences and technology*.

ISO/IEC 2382, *Information technology — Vocabulary*.

ISO/IEC 60559:2020, *Information technology — Microprocessor Systems — Floating-Point arithmetic*

The Unicode Consortium. The Unicode Standard, https://www.unicode.org/standard/standard.html

3 Terms and definitions

Article • 06/14/2022

For the purposes of this specification, the following definitions apply. Other terms are defined where they appear in ***italic*** type or on the left side of a syntax rule. Terms explicitly defined in this specification are not to be presumed to refer implicitly to similar terms defined elsewhere. Terms not defined in this specification are to be interpreted according to ISO/IEC 2382.1. Mathematical symbols not defined in this specification are to be interpreted according to ISO 80000-2.

- **application**
  - assembly with an entry point
- **application domain**
  - entity that enables application isolation by acting as a container for application state
- **argument**
  - expression in the comma-separated list bounded by the parentheses in a method or instance constructor call expression or bounded by the square brackets in an element access expression
- **assembly**
  - one or more files output by the compiler as a result of program compilation
- **behavior**
  - external appearance or action
- **behavior, implementation-defined**
  - unspecified behavior where each implementation documents how the choice is made
- **behavior, undefined**
  - behavior, upon use of a non-portable or erroneous construct or of erroneous data, for which this specification imposes no requirements
- **behavior, unspecified**
  - behavior where this specification provides two or more possibilities and imposes no further requirements on which is chosen in any instance
- **character** (when used without a qualifier)
  - In the context of a non-Unicode encoding, the meaning of character in that encoding; or
  - In the context of a character literal or a value of type char, a Unicode code point in the range U+0000 to U+FFFF (including surrogate code points), that is a UTF-16 code unit; or
  - Otherwise, a Unicode code point
- **class library**

    - assembly that can be used by other assemblies
- **compilation unit**
    - ordered sequence of Unicode characters that is input to a compiler
- **diagnostic message**
    - message belonging to an implementation-defined subset of the implementation's output messages
- **error, compile-time**
    - error reported during program translation
- **exception**
    - exceptional condition reported during program execution
- **implementation**
    - particular set of software (running in a particular translation environment under particular control options) that performs translation of programs for, and supports execution of methods in, a particular execution environment
- **module**
    - the contents of an assembly produced by a compiler. Some implementations may have facilities to produce assemblies that contain more than one module. The behavior in such situations is outside the scope of this specification
- **namespace**
    - logical organizational system grouping related program elements
- **parameter**
    - variable declared as part of a method, instance constructor, operator, or indexer definition, which acquires a value on entry to that function member
- **program**
    - one or more compilation units that are presented to the compiler and are run or executed by an execution environment
- **unsafe code**
    - code that is permitted to perform such lower-level operations as declaring and operating on pointers, performing conversions between pointers and integral types, and taking the address of variables
- **warning, compile-time**
    - informational message reported during program translation, which is intended to identify a potentially questionable usage of a program element

4 General description

Article • 04/08/2022

This text is informative.

This specification is intended to be used by implementers, academics, and application programmers. As such, it contains a considerable amount of explanatory material that, strictly speaking, is not necessary in a formal language specification.

This specification is divided into the following subdivisions: front matter; language syntax, constraints, and semantics; and annexes.

Examples are provided to illustrate possible forms of the constructions described. References are used to refer to related clauses. Notes are provided to give advice or guidance to implementers or programmers. Annexes provide additional information and summarize the information contained in this specification.

End of informative text.

Informative text is indicated in the following ways:

1. Whole or partial clauses or annexes delimited by "**This clause/text is informative**" and "**End of informative text**".
2. *Example*: The following example … code fragment, possibly with some narrative … *end example* The *Example:* and *end example* markers are in the same paragraph for single paragraph examples. If an example spans multiple paragraphs, the end example marker should be its own paragraph.
3. *Note*: narrative … *end note* The *Note*: and *end note* markers are in the same paragraph for single paragraph notes. If a note spans multiple paragraphs, the *end note* marker should be its own paragraph.

All text not marked as being informative is normative.

5 Conformance

Article • 03/18/2022

Conformance is of interest to the following audiences:

- Those designing, implementing, or maintaining C# implementations.
- Governmental or commercial entities wishing to procure C# implementations.
- Testing organizations wishing to provide a C# conformance test suite.
- Programmers wishing to port code from one C# implementation to another.
- Educators wishing to teach Standard C#.
- Authors wanting to write about Standard C#.

As such, conformance is most important, and the bulk of this specification is aimed at specifying the characteristics that make C# implementations and C# programs conforming ones.

The text in this specification that specifies requirements is considered ***normative***. All other text in this specification is ***informative***; that is, for information purposes only. Unless stated otherwise, all text is normative. Normative text is further broken into ***required*** and ***conditional*** categories. ***Conditionally normative*** text specifies a feature and its requirements where the feature is optional. However, if that feature is provided, its syntax and semantics shall be exactly as specified.

Undefined behavior is indicated in this specification only by the words 'undefined behavior.'

A ***strictly conforming program*** shall use only those features of the language specified in this specification as being required. (This means that a strictly conforming program cannot use any conditionally normative feature.) It shall not produce output dependent on any unspecified, undefined, or implementation-defined behavior.

A ***conforming implementation*** of C# shall accept any strictly conforming program.

A conforming implementation of C# shall provide and support all the types, values, objects, properties, methods, and program syntax and semantics described in the normative (but not the conditionally normative) parts in this specification.

A conforming implementation of C# shall interpret characters in conformance with the Unicode Standard. Conforming implementations shall accept compilation units encoded with the UTF-8 encoding form.

A conforming implementation of C# shall not successfully translate source containing a #error preprocessing directive unless it is part of a group skipped by conditional

compilation.

A conforming implementation of C# shall produce at least one diagnostic message if the source program violates any rule of syntax, or any negative requirement (defined as a “shall” or “shall not” or “error” or “warning” requirement), unless that requirement is marked with the words “no diagnostic is required”.

A conforming implementation of C# is permitted to provide additional types, values, objects, properties, and methods beyond those described in this specification, provided they do not alter the behavior of any strictly conforming program. Conforming implementations are required to diagnose programs that use extensions that are ill formed according to this specification. Having done so, however, they can compile and execute such programs. (The ability to have extensions implies that a conforming implementation reserves no identifiers other than those explicitly reserved in this specification.)

A conforming implementation of C# shall be accompanied by a document that defines all implementation-defined characteristics, and all extensions.

A conforming implementation of C# shall support the class library documented in Annex C. This library is included by reference in this specification.

A ***conforming program*** is one that is acceptable to a conforming implementation. (Such a program is permitted to contain extensions or conditionally normative features.)

Structure lexicale

Article • 13/05/2021

Programmes

Un **programme** C# * _ est constitué d'un ou de plusieurs *fichiers sources*, connus sous le nom d'*unités de compilation*(unités de compilation). Un fichier source est une séquence ordonnée de caractères Unicode. Les fichiers sources ont généralement une correspondance un-à-un avec les fichiers dans un système de fichiers, mais cette correspondance n'est pas nécessaire. Pour une portabilité maximale, il est recommandé d'encoder les fichiers d'un système de fichiers avec l'encodage UTF-8.

Conceptuellement, un programme est compilé à l'aide de trois étapes :

1. Transformation, qui convertit un fichier d'un répertoire de caractères particulier et d'un schéma d'encodage en une séquence de caractères Unicode.
2. Analyse lexicale, qui convertit un flux de caractères d'entrée Unicode en un flux de jetons.
3. Analyse syntaxique, qui convertit le flux de jetons en code exécutable.

Grammaires

Cette spécification présente la syntaxe du langage de programmation C# à l'aide de deux grammaires. La ***grammaire lexicale** _ (grammaire lexicale) définit la façon dont les caractères Unicode sont combinés pour former des marques de fin de ligne, des espaces blancs, des commentaires, des jetons et des directives de pré-traitement. La syntaxe _ *syntaxe** (syntaxe syntaxique) définit la façon dont les jetons résultant de la grammaire lexicale sont combinés pour former les programmes C#.

Notation grammaticale

Les grammaires lexicales et syntaxiques sont présentées sous forme de Backus-Naur à l'aide de la notation de l'outil de grammaire ANTLR.

Grammaire lexicale

La grammaire lexicale de C# est présentée dans l' analyse lexicale, les jetonset les directives de pré-traitement. Les symboles terminaux de la grammaire lexicale sont les caractères du jeu de caractères Unicode, et la grammaire lexicale spécifie la manière

dont les caractères sont combinés pour former des jetons (jetons), un espace blanc (espace blanc), des commentaires (Commentaires) et des directives de prétraitement (directives de prétraitement).

Chaque fichier source d'un programme C# doit se conformer à la production d' *entrée* de la grammaire lexicale (analyse lexicale).

Grammaire syntaxique

La grammaire syntaxique de C# est présentée dans les chapitres et les annexes qui suivent ce chapitre. Les symboles terminaux de la grammaire syntaxique sont les jetons définis par la grammaire lexicale, et la grammaire syntaxique spécifie la manière dont les jetons sont combinés pour former les programmes C#.

Chaque fichier source d'un programme C# doit se conformer aux *compilation_unit* la production de la grammaire syntaxique (unités de compilation).

Analyse lexicale

La production *d'entrée* définit la structure lexicale d'un fichier source C#. Chaque fichier source d'un programme C# doit se conformer à cette production grammaticale lexicale.

antlr

```
input
    : input_section?
    ;

input_section
    : input_section_part+
    ;

input_section_part
    : input_element* new_line
    | pp_directive
    ;

input_element
    : whitespace
    | comment
    | token
    ;
```

Cinq éléments de base composent la structure lexicale d'un fichier source C# : les terminateurs de ligne (indicateurs de fin de ligne), les espaces blancs (espacesblancs), les commentaires (Commentaires), les jetons (jetons) et les directives de pré-traitement

(directives de pré-traitement). Parmi ces éléments de base, seuls les jetons sont significatifs dans la syntaxe syntaxique d'un programme C# (syntaxe syntaxique).

Le traitement lexical d'un fichier source C# consiste à réduire le fichier en une séquence de jetons qui devient l'entrée de l'analyse syntaxique. Les indicateurs de fin de ligne, les espaces blancs et les commentaires peuvent servir à séparer les jetons, tandis que les directives de prétraitement peuvent ignorer des sections du fichier source, mais ces éléments lexicaux n'ont aucun impact sur la structure syntaxique d'un programme C#.

Dans le cas de littéraux de chaîne interpolés (littéraux de chaîne interpolés), un seul jeton est initialement généré par l'analyse lexicale, mais il est divisé en plusieurs éléments d'entrée qui sont soumis à plusieurs reprises à l'analyse lexicale jusqu'à ce que tous les littéraux de chaîne interpolés aient été résolus. Les jetons résultants servent ensuite d'entrée à l'analyse syntaxique.

Lorsque plusieurs productions grammaticales lexicales correspondent à une séquence de caractères dans un fichier source, le traitement lexical forme toujours l'élément lexical le plus long possible. Par exemple, la séquence `//` de caractères est traitée comme le début d'un commentaire sur une seule ligne, car cet élément lexical est plus long qu'un seul `/` jeton.

Marques de fin de ligne

Les marques de fin de ligne divisent les caractères d'un fichier source C# en lignes.

antlr

```
new_line
    : '<Carriage return character (U+000D)>'
    | '<Line feed character (U+000A)>'
    | '<Carriage return character (U+000D) followed by line feed character
(U+000A)>'
    | '<Next line character (U+0085)>'
    | '<Line separator character (U+2028)>'
    | '<Paragraph separator character (U+2029)>'
    ;
```

Pour la compatibilité avec les outils d'édition de code source qui ajoutent des marqueurs de fin de fichier et pour permettre à un fichier source d'être affiché sous la forme d'une séquence de lignes correctement terminées, les transformations suivantes sont appliquées, dans l'ordre, à chaque fichier source dans un programme C# :

- Si le dernier caractère du fichier source est un caractère de contrôle Z (`U+001A`), ce caractère est supprimé.

- Un caractère de retour chariot (`U+000D`) est ajouté à la fin du fichier source si ce fichier source n'est pas vide et si le dernier caractère du fichier source n'est pas un retour chariot (`U+000D`), un saut de ligne (`U+000A`), un séparateur de ligne (`U+2028`) ou un séparateur de paragraphe (`U+2029`).

Commentaires

Deux formes de commentaires sont pris en charge : les commentaires sur une seule ligne et les commentaires délimités.\ ***Commentaires sur une seule ligne*** _ *commencent par les caractères* `//` *et s'étendent jusqu'à la fin de la ligne source. Les* _Commentaires *délimités*_ commencent par les caractères `/_` et se terminent par les caractères `*/` . Les commentaires délimités peuvent s'étendre sur plusieurs lignes.

antlr

```
comment
    : single_line_comment
    | delimited_comment
    ;

single_line_comment
    : '//' input_character*
    ;

input_character
    : '<Any Unicode character except a new_line_character>'
    ;

new_line_character
    : '<Carriage return character (U+000D)>'
    | '<Line feed character (U+000A)>'
    | '<Next line character (U+0085)>'
    | '<Line separator character (U+2028)>'
    | '<Paragraph separator character (U+2029)>'
    ;

delimited_comment
    : '/*' delimited_comment_section* asterisk+ '/'
    ;

delimited_comment_section
    : '/'
    | asterisk* not_slash_or_asterisk
    ;

asterisk
    : '*'
    ;

not_slash_or_asterisk
```

```
    : '<Any Unicode character except / or *>'
    ;
```

Les commentaires ne peuvent pas être imbriqués. Les séquences de caractères `/*` et n' `*/` ont aucune signification particulière dans un `//` commentaire, et les séquences de caractères `//` et n' `/*` ont aucune signification particulière dans un commentaire délimité.

Les commentaires ne sont pas traités dans des littéraux de chaîne et de caractère.

L'exemple

C#

```
/* Hello, world program
   This program writes "hello, world" to the console
*/
class Hello
{
    static void Main() {
        System.Console.WriteLine("hello, world");
    }
}
```

comprend un commentaire délimité.

L'exemple

C#

```
// Hello, world program
// This program writes "hello, world" to the console
//
class Hello // any name will do for this class
{
    static void Main() { // this method must be named "Main"
        System.Console.WriteLine("hello, world");
    }
}
```

illustre plusieurs commentaires sur une seule ligne.

Espace blanc

Un espace blanc est défini comme n'importe quel caractère avec la classe Unicode Zs (qui comprend le caractère espace), ainsi que le caractère de tabulation horizontale, le

caractère de tabulation verticale et le caractère de saut de forme.

```antlr
whitespace
    : '<Any character with Unicode class Zs>'
    | '<Horizontal tab character (U+0009)>'
    | '<Vertical tab character (U+000B)>'
    | '<Form feed character (U+000C)>'
    ;
```

Jetons

Il existe plusieurs genres de jetons : les identificateurs, les mots clés, les littéraux, les opérateurs et les signes de ponctuation. Les espaces blancs et les commentaires ne sont pas des jetons, bien qu'ils agissent comme séparateurs pour les jetons.

```antlr
token
    : identifier
    | keyword
    | integer_literal
    | real_literal
    | character_literal
    | string_literal
    | interpolated_string_literal
    | operator_or_punctuator
    ;
```

Séquences d'échappement de caractères Unicode

Une séquence d'échappement de caractère Unicode représente un caractère Unicode. Les séquences d'échappement de caractères Unicode sont traitées dans les identificateurs (identificateurs), les littéraux de caractère (littéraux de caractère) et les littéraux de chaîne normaux (littéraux de chaîne). Un caractère d'échappement Unicode n'est pas traité à un autre emplacement (par exemple, pour former un opérateur, un signe de ponctuation ou un mot clé).

```antlr
unicode_escape_sequence
    : '\\u' hex_digit hex_digit hex_digit hex_digit
    | '\\U' hex_digit hex_digit hex_digit hex_digit hex_digit hex_digit
```

```
hex_digit hex_digit
    ;
```

Une séquence d'échappement Unicode représente le caractère Unicode unique formé par le nombre hexadécimal suivant les `\u` caractères « » ou « » `\U` . Étant donné que C# utilise un encodage 16 bits de points de code Unicode en caractères et valeurs de chaîne, un caractère Unicode dans la plage comprise entre U + 10000 et U + 10FFFF n'est pas autorisé dans un littéral de caractère et est représenté à l'aide d'une paire de substitution Unicode dans un littéral de chaîne. Les caractères Unicode avec des points de code au-dessus de 0x10FFFF ne sont pas pris en charge.

Plusieurs traductions ne sont pas effectuées. Par exemple, le littéral de chaîne « `\u005Cu005C` » est équivalent à « `\u005C` » plutôt qu'à «» `\` . La valeur Unicode `\u005C` est le caractère « `\` ».

L'exemple

C#

```
class Class1
{
    static void Test(bool \u0066) {
        char c = '\u0066';
        if (\u0066)
            System.Console.WriteLine(c.ToString());
    }
}
```

montre plusieurs utilisations de `\u0066` , qui est la séquence d'échappement pour la lettre « `f` ». Le programme est équivalent à

C#

```
class Class1
{
    static void Test(bool f) {
        char c = 'f';
        if (f)
            System.Console.WriteLine(c.ToString());
    }
}
```

Identificateurs

Les règles relatives aux identificateurs fournis dans cette section correspondent exactement à celles recommandées par la norme Unicode annexe 31, sauf que le trait de soulignement est autorisé comme caractère initial (comme c'est le cas dans le langage de programmation C), que les séquences d'échappement Unicode sont autorisées dans les identificateurs et que le `@` caractère «

antlr

```
identifier
    : available_identifier
    | '@' identifier_or_keyword
    ;

available_identifier
    : '<An identifier_or_keyword that is not a keyword>'
    ;

identifier_or_keyword
    : identifier_start_character identifier_part_character*
    ;

identifier_start_character
    : letter_character
    | '_'
    ;

identifier_part_character
    : letter_character
    | decimal_digit_character
    | connecting_character
    | combining_character
    | formatting_character
    ;

letter_character
    : '<A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nl>'
    | '<A unicode_escape_sequence representing a character of classes Lu,
Ll, Lt, Lm, Lo, or Nl>'
    ;

combining_character
    : '<A Unicode character of classes Mn or Mc>'
    | '<A unicode_escape_sequence representing a character of classes Mn or
Mc>'
    ;

decimal_digit_character
    : '<A Unicode character of the class Nd>'
    | '<A unicode_escape_sequence representing a character of the class Nd>'
    ;

connecting_character
```

```
    : '<A Unicode character of the class Pc>'
    | '<A unicode_escape_sequence representing a character of the class Pc>'
    ;

formatting_character
    : '<A Unicode character of the class Cf>'
    | '<A unicode_escape_sequence representing a character of the class Cf>'
    ;
```

Pour plus d'informations sur les classes de caractères Unicode mentionnées ci-dessus, consultez la norme Unicode, version 3,0, section 4,5.

« `identifier1` », « `_identifier2` » Et «» sont des exemples d'identificateurs valides `@if` .

Un identificateur dans un programme conforme doit être au format canonique défini par le formulaire de normalisation Unicode C, comme défini par la norme Unicode annexe 15. Le comportement lors de la rencontre d'un identificateur qui ne figure pas dans le formulaire de normalisation C est défini par l'implémentation ; Toutefois, aucun diagnostic n'est nécessaire.

Le préfixe « `@` » permet d'utiliser des mots clés en tant qu'identificateurs, ce qui est utile lors de l'interfaçage avec d'autres langages de programmation. Le caractère ne `@` faisant pas réellement partie de l'identificateur, l'identificateur peut être vu dans d'autres langages comme un identificateur normal, sans le préfixe. Un identificateur avec un `@` préfixe est appelé ***identificateur textuel***. L'utilisation du `@` préfixe pour les identificateurs qui ne sont pas des mots clés est autorisée, mais elle est fortement déconseillée comme une question de style.

L'exemple :

C#

```
class @class
{
    public static void @static(bool @bool) {
        if (@bool)
            System.Console.WriteLine("true");
        else
            System.Console.WriteLine("false");
    }
}

class Class1
{
    static void M() {
        cl\u0061ss.st\u0061tic(true);
    }
}
```

définit une classe nommée « `class` » avec une méthode statique nommée « » `static` qui prend un paramètre nommé « `bool` ». Notez que dans la mesure où les échappements Unicode ne sont pas autorisés dans les mots clés, le jeton « `cl\u0061ss` » est un identificateur et est le même identificateur que « `@class` ».

Deux identificateurs sont considérés comme identiques s'ils sont identiques après l'application des transformations suivantes, dans l'ordre :

- Le préfixe « `@` », s'il est utilisé, est supprimé.
- Chaque *unicode_escape_sequence* est transformée en son caractère Unicode correspondant.
- Les *formatting_character* s sont supprimés.

Les identificateurs contenant deux traits de soulignement consécutifs (`U+005F`) sont réservés pour une utilisation par l'implémentation. Par exemple, une implémentation peut fournir des mots clés étendus qui commencent par deux traits de soulignement.

Mots clés

Un ***mot clé*** est une séquence de caractères de type identificateur, qui est réservée et ne peut pas être utilisée en tant qu'identificateur, sauf s'il est précédé du `@` caractère.

antlr

```
keyword
    : 'abstract' | 'as'       | 'base'       | 'bool'      | 'break'
    | 'byte'     | 'case'     | 'catch'      | 'char'      | 'checked'
    | 'class'    | 'const'    | 'continue'   | 'decimal'   | 'default'
    | 'delegate' | 'do'       | 'double'     | 'else'      | 'enum'
    | 'event'    | 'explicit' | 'extern'     | 'false'     | 'finally'
    | 'fixed'    | 'float'    | 'for'        | 'foreach'   | 'goto'
    | 'if'       | 'implicit' | 'in'         | 'int'       | 'interface'
    | 'internal' | 'is'       | 'lock'       | 'long'      | 'namespace'
    | 'new'      | 'null'     | 'object'     | 'operator'  | 'out'
    | 'override' | 'params'   | 'private'    | 'protected' | 'public'
    | 'readonly' | 'ref'      | 'return'     | 'sbyte'     | 'sealed'
    | 'short'    | 'sizeof'   | 'stackalloc' | 'static'    | 'string'
    | 'struct'   | 'switch'   | 'this'       | 'throw'     | 'true'
    | 'try'      | 'typeof'   | 'uint'       | 'ulong'     | 'unchecked'
    | 'unsafe'   | 'ushort'   | 'using'      | 'virtual'   | 'void'
    | 'volatile' | 'while'
    ;
```

À certains endroits de la grammaire, les identificateurs spécifiques ont une signification particulière, mais ne sont pas des mots clés. Ces identificateurs sont parfois appelés

« Mots-clés contextuels ». Par exemple, dans une déclaration de propriété, les `get` identificateurs « » et « » `set` ont une signification spéciale (accesseurs). Comme un identificateur autre que `get` ou `set` n'est jamais autorisé dans ces emplacements, cette utilisation n'est pas en conflit avec l'utilisation de ces mots comme identificateurs. Dans d'autres cas, comme avec l'identificateur « `var` » dans les déclarations de variables locales implicitement typées (déclarations de variables locales), un mot clé contextuel peut être en conflit avec des noms déclarés. Dans ce cas, le nom déclaré est prioritaire par rapport à l'utilisation de l'identificateur en tant que mot clé contextuel.

Littéraux

Un ***littéral*** est une représentation en code source d'une valeur.

antlr

```
literal
    : boolean_literal
    | integer_literal
    | real_literal
    | character_literal
    | string_literal
    | null_literal
    ;
```

littéraux booléens

Il existe deux valeurs littérales booléennes : `true` et `false` .

antlr

```
boolean_literal
    : 'true'
    | 'false'
    ;
```

Le type d'un *boolean_literal* est `bool` .

Littéraux d'entier

Les littéraux entiers sont utilisés pour écrire des valeurs de types `int` ,, `uint` `long` et `ulong` . Les littéraux entiers ont deux formes possibles : decimal et hexadécimal.

antlr

```
integer_literal
    : decimal_integer_literal
    | hexadecimal_integer_literal
    ;

decimal_integer_literal
    : decimal_digit+ integer_type_suffix?
    ;

decimal_digit
    : '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
    ;

integer_type_suffix
    : 'U' | 'u' | 'L' | 'l' | 'UL' | 'Ul' | 'uL' | 'ul' | 'LU' | 'Lu' | 'lU'
| 'lu'
    ;

hexadecimal_integer_literal
    : '0x' hex_digit+ integer_type_suffix?
    | '0X' hex_digit+ integer_type_suffix?
    ;

hex_digit
    : '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
    | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f';
```

Le type d'un littéral entier est déterminé comme suit :

- Si le littéral n'a pas de suffixe, il a le premier de ces types dans lequel sa valeur peut être représentée : `int` , `uint` , `long` , `ulong` .
- Si le littéral est suivi d'un suffixe par `U` ou `u` , il a le premier de ces types dans lequel sa valeur peut être représentée : `uint` , `ulong` .
- Si le littéral est suivi d'un suffixe par `L` ou `l` , il a le premier de ces types dans lequel sa valeur peut être représentée : `long` , `ulong` .
- Si le littéral est précédé d'un suffixe `UL` , `Ul` , `uL` , `ul` , `LU` , `Lu` , `lU` ou `lu` , il est de type `ulong` .

Si la valeur représentée par un littéral entier est en dehors de la plage du `ulong` type, une erreur de compilation se produit.

En guise de style, il est suggéré d' `L` utiliser « » à la place de « » lors de l' `l` écriture de littéraux de type `long` , car il est facile de confondre la lettre « `l` » avec le chiffre « `1` ».

Pour que les valeurs et les plus petites `int` `long` soient écrites en tant que littéraux décimaux entiers, les deux règles suivantes existent :

- Lorsqu'un *decimal_integer_literal* avec la valeur 2147483648 (2 ^ 31) et qu'aucun *integer_type_suffix* n'apparaît en tant que jeton juste après un jeton d'opérateur moins unaire (opérateur moins unaire), le résultat est une constante de type `int` avec la valeur-2147483648 (-2 ^ 31). Dans toutes les autres situations, une telle *decimal_integer_literal* est de type `uint` .
- Lorsqu'un *decimal_integer_literal* avec la valeur 9223372036854775808 (2 ^ 63) et aucune *integer_type_suffix* ou le *integer_type_suffix* `L` ou `l` apparaît en tant que jeton qui suit immédiatement un jeton d'opérateur moins unaire (opérateur moins unaire), le résultat est une constante de type `long` avec la valeur-9223372036854775808 (-2 ^ 63). Dans toutes les autres situations, une telle *decimal_integer_literal* est de type `ulong` .

Littéraux réels

Les littéraux réels sont utilisés pour écrire des valeurs de types `float` , `double` et `decimal` .

antlr

```
real_literal
    : decimal_digit+ '.' decimal_digit+ exponent_part? real_type_suffix?
    | '.' decimal_digit+ exponent_part? real_type_suffix?
    | decimal_digit+ exponent_part real_type_suffix?
    | decimal_digit+ real_type_suffix
    ;

exponent_part
    : 'e' sign? decimal_digit+
    | 'E' sign? decimal_digit+
    ;

sign
    : '+'
    | '-'
    ;

real_type_suffix
    : 'F' | 'f' | 'D' | 'd' | 'M' | 'm'
    ;
```

Si aucun *real_type_suffix* n'est spécifié, le type du littéral réel est `double` . Dans le cas contraire, le suffixe de type réel détermine le type du littéral réel, comme suit :

- Littéral réel dont le suffixe `F` `f` est ou est de type `float` . Par exemple, les littéraux `1f` , `1.5f` , `1e10f` et `123.456F` sont tous de type `float` .

- Littéral réel dont le suffixe `D` `d` est ou est de type `double` . Par exemple, les littéraux `1d` , `1.5d` , `1e10d` et `123.456D` sont tous de type `double` .
- Littéral réel dont le suffixe `M` `m` est ou est de type `decimal` . Par exemple, les littéraux `1m` , `1.5m` , `1e10m` et `123.456M` sont tous de type `decimal` . Ce littéral est converti en une `decimal` valeur en acceptant la valeur exacte et, si nécessaire, en arrondissant à la valeur représentable la plus proche à l'aide de l'arrondi de Banker (type décimal). Toute échelle apparente dans le littéral est conservée, sauf si la valeur est arrondie ou si la valeur est égale à zéro (dans ce dernier cas, le signe et l'échelle sont 0). Par conséquent, le littéral `2.900m` sera analysé pour former la décimale avec le signe `0` , le coefficient et l' `2900` échelle `3` .

Si le littéral spécifié ne peut pas être représenté dans le type indiqué, une erreur de compilation se produit.

La valeur d'un littéral réel de type `float` ou `double` est déterminée à l'aide du mode IEEE « arrondi au plus proche ».

Notez que dans un littéral réel, les chiffres décimaux sont toujours requis après la virgule décimale. Par exemple, `1.3F` est un littéral réel, mais `1.F` pas.

Littéraux de caractère

Un littéral de caractère représente un caractère unique et se compose généralement d'un caractère entre guillemets, comme dans `'a'` .

Remarque : la notation de la grammaire ANTLR rend ce qui est confus. Dans ANTLR, lorsque vous écrivez, `\'` il s'agit d'un guillemet simple `'` . Lorsque vous écrivez, `\\` il s'agit d'une barre oblique inverse unique `\` . Par conséquent, la première règle pour un littéral de caractère signifie qu'il commence par un guillemet simple, puis par un caractère, puis par un guillemet simple. Et les onze séquences d'échappement simples possibles sont `\'` , `\"` ,, `\\` `\0` , `\a` , `\b` , `\f` ,,, `\n` `\r` `\t` , `\v` .

antlr

```
character_literal
    : '\'' character '\''
    ;

character
    : single_character
    | simple_escape_sequence
    | hexadecimal_escape_sequence
    | unicode_escape_sequence
    ;
```

```
single_character
    : '<Any character except \' (U+0027), \\ (U+005C), and
new_line_character>'
    ;

simple_escape_sequence
    : '\\\'' | '\\"' | '\\\\' | '\\0' | '\\a' | '\\b' | '\\f' | '\\n' |
'\\r' | '\\t' | '\\v'
    ;

hexadecimal_escape_sequence
    : '\\x' hex_digit hex_digit? hex_digit? hex_digit?;
```

Un caractère qui suit une barre oblique inverse (`\`) dans un *caractère* doit être l'un des caractères suivants :,,,, `'` `"` `\` `0` `a` , `b` , `f` , `n` , `r` , `t` , `u` , `U` , `x` , `v` . Sinon, une erreur de compilation se produit.

Une séquence d'échappement hexadécimale représente un caractère Unicode unique, avec la valeur formée par le nombre hexadécimal suivant « `\x` ».

Si la valeur représentée par un littéral de caractère est supérieure à `U+FFFF` , une erreur de compilation se produit.

Une séquence d'échappement de caractère Unicode (séquences d'échappement de caractère Unicode) dans un littéral de caractère doit être comprise dans la plage `U+0000` de `U+FFFF` .

Une séquence d'échappement simple représente un encodage de caractères Unicode, comme décrit dans le tableau ci-dessous.

Séquence d'échappement	Nom du caractère	Codage Unicode
`\'`	Guillemet simple	`0x0027`
`\"`	Guillemet double	`0x0022`
`\\`	Barre oblique inverse	`0x005C`
`\0`	Null	`0x0000`
`\a`	Alerte	`0x0007`
`\b`	Retour arrière	`0x0008`
`\f`	Saut de page	`0x000C`
`\n`	Nouvelle ligne	`0x000A`

Séquence d'échappement	Nom du caractère	Codage Unicode
`\r`	Retour chariot	`0x000D`
`\t`	Tabulation horizontale	`0x0009`
`\v`	Tabulation verticale	`0x000B`

Le type d'un *character_literal* est `char` .

Littéraux de chaîne

C# prend en charge deux formes de littéraux de chaîne : *__littéraux__ de chaîne normaux _ et _ *littéraux de chaîne Verbatim* *.

Un littéral de chaîne standard se compose de zéro, un ou plusieurs caractères placés entre guillemets doubles, comme dans `"hello"` , et peut inclure à la fois des séquences d'échappement simples (comme `\t` pour le caractère de tabulation) et des séquences d'échappement hexadécimales et Unicode.

Un littéral de chaîne Verbatim se compose d'un `@` caractère suivi d'un guillemet double, de zéro, d'un ou de plusieurs caractères et d'un guillemet double fermant. Un exemple simple est `@"hello"` . Dans un littéral de chaîne Verbatim, les caractères entre les délimiteurs sont interprétés textuellement, la seule exception étant un *quote_escape_sequence*. En particulier, les séquences d'échappement simples et les séquences d'échappement hexadécimales et Unicode ne sont pas traitées dans des littéraux de chaîne Verbatim. Un littéral de chaîne Verbatim peut s'étendre sur plusieurs lignes.

antlr

```
string_literal
    : regular_string_literal
    | verbatim_string_literal
    ;

regular_string_literal
    : '"' regular_string_literal_character* '"'
    ;

regular_string_literal_character
    : single_regular_string_literal_character
    | simple_escape_sequence
    | hexadecimal_escape_sequence
    | unicode_escape_sequence
    ;
```

```
single_regular_string_literal_character
    : '<Any character except " (U+0022), \\ (U+005C), and
new_line_character>'
    ;

verbatim_string_literal
    : '@"' verbatim_string_literal_character* '"'
    ;

verbatim_string_literal_character
    : single_verbatim_string_literal_character
    | quote_escape_sequence
    ;

single_verbatim_string_literal_character
    : '<any character except ">'
    ;

quote_escape_sequence
    : '""'
    ;
```

Un caractère qui suit une barre oblique inverse (`\`) dans un *regular_string_literal_character* doit être l'un des caractères suivants :,,,, `'` `"` `\` `0` `a` , `b` , `f` , `n` , `r` , `t` , `u` , `U` , `x` , `v` . Sinon, une erreur de compilation se produit.

L'exemple

C#

```
string a = "hello, world";                  // hello, world
string b = @"hello, world";                 // hello, world

string c = "hello \t world";                // hello      world
string d = @"hello \t world";               // hello \t world

string e = "Joe said \"Hello\" to me";      // Joe said "Hello" to me
string f = @"Joe said ""Hello"" to me";     // Joe said "Hello" to me

string g = "\\\\server\\share\\file.txt";   // \\server\share\file.txt
string h = @"\\server\share\file.txt";      // \\server\share\file.txt

string i = "one\r\ntwo\r\nthree";
string j = @"one
two
three";
```

affiche divers littéraux de chaîne. Le dernier littéral de chaîne, `j` , est un littéral de chaîne Verbatim qui s'étend sur plusieurs lignes. Les caractères entre guillemets, y compris les espaces blancs tels que les caractères de nouvelle ligne, sont conservés textuellement.

Comme une séquence d'échappement hexadécimale peut avoir un nombre variable de chiffres hexadécimaux, le littéral de chaîne `"\x123"` contient un caractère unique avec la valeur hexadécimale 123. Pour créer une chaîne contenant le caractère avec la valeur hexadécimale 12 suivi du caractère 3, il est possible d'écrire ou à la `"\x00123"` `"\x12" + "3"` place.

Le type d'un *string_literal* est `string` .

Chaque littéral de chaîne ne produit pas nécessairement une nouvelle instance de chaîne. Lorsque deux littéraux de chaîne ou plus qui sont équivalents en fonction de l'opérateur d'égalité de chaînes (opérateurs d'égalité de chaînes) apparaissent dans le même programme, ces littéraux de chaîne font référence à la même instance de chaîne. Par exemple, la sortie produite par

C#

```
class Test
{
    static void Main() {
        object a = "hello";
        object b = "hello";
        System.Console.WriteLine(a == b);
    }
}
```

est `True` dû au fait que les deux littéraux font référence à la même instance de chaîne.

Littéraux de chaîne interpolés

Les littéraux de chaîne interpolés sont similaires aux littéraux de chaîne, mais contiennent des trous délimités par `{` et `}` , où les expressions peuvent se produire. Lors de l'exécution, les expressions sont évaluées dans le but de faire en sorte que leurs formes textuelles se substituent à la chaîne à l'endroit où se trouve le trou. La syntaxe et la sémantique de l'interpolation de chaîne sont décrites dans la section (chaînes interpolées).

À l'instar des littéraux de chaîne, les littéraux de chaîne interpolés peuvent être normaux ou textuels. Les littéraux de chaîne réguliers interpolés sont délimités par `$"` et `"` , et les littéraux de chaîne Verbatim interpolés sont délimités par `$@"` et `"` .

Comme pour les autres littéraux, l'analyse lexicale d'un littéral de chaîne interpolée produit initialement un jeton unique, conformément à la grammaire ci-dessous. Toutefois, avant l'analyse syntaxique, le jeton unique d'un littéral de chaîne interpolé est

divisé en plusieurs jetons pour les parties de la chaîne qui entourent les trous, et les éléments d'entrée qui se produisent dans les trous sont analysés de façon lexicale. Cela peut, à son tour, générer davantage de littéraux de chaîne interpolés à traiter, mais, s'il est correct, finit par entraîner une séquence de jetons pour l'analyse syntaxique.

antlr

```
interpolated_string_literal
    : '$' interpolated_regular_string_literal
    | '$' interpolated_verbatim_string_literal
    ;

interpolated_regular_string_literal
    : interpolated_regular_string_whole
    | interpolated_regular_string_start
interpolated_regular_string_literal_body interpolated_regular_string_end
    ;

interpolated_regular_string_literal_body
    : regular_balanced_text
    | interpolated_regular_string_literal_body
interpolated_regular_string_mid regular_balanced_text
    ;

interpolated_regular_string_whole
    : '"' interpolated_regular_string_character* '"'
    ;

interpolated_regular_string_start
    : '"' interpolated_regular_string_character* '{'
    ;

interpolated_regular_string_mid
    : interpolation_format? '}'
interpolated_regular_string_characters_after_brace? '{'
    ;

interpolated_regular_string_end
    : interpolation_format? '}'
interpolated_regular_string_characters_after_brace? '"'
    ;

interpolated_regular_string_characters_after_brace
    : interpolated_regular_string_character_no_brace
    | interpolated_regular_string_characters_after_brace
interpolated_regular_string_character
    ;

interpolated_regular_string_character
    : single_interpolated_regular_string_character
    | simple_escape_sequence
    | hexadecimal_escape_sequence
    | unicode_escape_sequence
```

```
    | open_brace_escape_sequence
    | close_brace_escape_sequence
    ;

interpolated_regular_string_character_no_brace
    : '<Any interpolated_regular_string_character except
close_brace_escape_sequence and any hexadecimal_escape_sequence or
unicode_escape_sequence designating } (U+007D)>'
    ;

single_interpolated_regular_string_character
    : '<Any character except \" (U+0022), \\ (U+005C), { (U+007B), }
(U+007D), and new_line_character>'
    ;

open_brace_escape_sequence
    : '{{'
    ;

close_brace_escape_sequence
    : '}}'
    ;

regular_balanced_text
    : regular_balanced_text_part+
    ;

regular_balanced_text_part
    : single_regular_balanced_text_character
    | delimited_comment
    | '@' identifier_or_keyword
    | string_literal
    | interpolated_string_literal
    | '(' regular_balanced_text ')'
    | '[' regular_balanced_text ']'
    | '{' regular_balanced_text '}'
    ;

single_regular_balanced_text_character
    : '<Any character except / (U+002F), @ (U+0040), \" (U+0022), $
(U+0024), ( (U+0028), ) (U+0029), [ (U+005B), ] (U+005D), { (U+007B), }
(U+007D) and new_line_character>'
    | '</ (U+002F), if not directly followed by / (U+002F) or * (U+002A)>'
    ;

interpolation_format
    : ':' interpolation_format_character+
    ;

interpolation_format_character
    : '<Any character except \" (U+0022), : (U+003A), { (U+007B) and }
(U+007D)>'
    ;

interpolated_verbatim_string_literal
```

```
    : interpolated_verbatim_string_whole
    | interpolated_verbatim_string_start
interpolated_verbatim_string_literal_body interpolated_verbatim_string_end
    ;

interpolated_verbatim_string_literal_body
    : verbatim_balanced_text
    | interpolated_verbatim_string_literal_body
interpolated_verbatim_string_mid verbatim_balanced_text
    ;

interpolated_verbatim_string_whole
    : '@"' interpolated_verbatim_string_character* '"'
    ;

interpolated_verbatim_string_start
    : '@"' interpolated_verbatim_string_character* '{'
    ;

interpolated_verbatim_string_mid
    : interpolation_format? '}'
interpolated_verbatim_string_characters_after_brace? '{'
    ;

interpolated_verbatim_string_end
    : interpolation_format? '}'
interpolated_verbatim_string_characters_after_brace? '"'
    ;

interpolated_verbatim_string_characters_after_brace
    : interpolated_verbatim_string_character_no_brace
    | interpolated_verbatim_string_characters_after_brace
interpolated_verbatim_string_character
    ;

interpolated_verbatim_string_character
    : single_interpolated_verbatim_string_character
    | quote_escape_sequence
    | open_brace_escape_sequence
    | close_brace_escape_sequence
    ;

interpolated_verbatim_string_character_no_brace
    : '<Any interpolated_verbatim_string_character except
close_brace_escape_sequence>'
    ;

single_interpolated_verbatim_string_character
    : '<Any character except \" (U+0022), { (U+007B) and } (U+007D)>'
    ;

verbatim_balanced_text
    : verbatim_balanced_text_part+
    ;
```

```
verbatim_balanced_text_part
    : single_verbatim_balanced_text_character
    | comment
    | '@' identifier_or_keyword
    | string_literal
    | interpolated_string_literal
    | '(' verbatim_balanced_text ')'
    | '[' verbatim_balanced_text ']'
    | '{' verbatim_balanced_text '}'
    ;

single_verbatim_balanced_text_character
    : '<Any character except / (U+002F), @ (U+0040), \" (U+0022), $
(U+0024), ( (U+0028), ) (U+0029), [ (U+005B), ] (U+005D), { (U+007B) and }
(U+007D)>'
    | '</ (U+002F), if not directly followed by / (U+002F) or * (U+002A)>'
    ;
```

Un jeton de *interpolated_string_literal* est réinterprété comme plusieurs jetons et autres éléments d'entrée comme suit, dans l'ordre des occurrences dans le *interpolated_string_literal*:

- Les occurrences des éléments suivants sont réinterprétées comme des jetons individuels distincts : le signe de début `$` , *interpolated_regular_string_whole*, *interpolated_regular_string_start*, *interpolated_regular_string_mid*, *interpolated_regular_string_end*, *interpolated_verbatim_string_whole*, *interpolated_verbatim_string_start*, *interpolated_verbatim_string_mid* et *interpolated_verbatim_string_end*.
- Les occurrences de *regular_balanced_text* et *verbatim_balanced_text* entre celles-ci sont retraitées comme une *input_section* (analyse lexicale) et sont réinterprétées comme la séquence résultante d'éléments d'entrée. Elles peuvent à leur tour inclure des jetons de littéral de chaîne interpolés à réinterpréter.

L'analyse syntaxique regroupe les jetons en un *interpolated_string_expression* (chaînes interpolées).

Exemples TODO

Littéral null

antlr

```
null_literal
    : 'null'
    ;
```

Le *null_literal* peut être converti implicitement en un type référence ou un type Nullable.

Opérateurs et signes de ponctuation

Il existe plusieurs genres d'opérateurs et de signes de ponctuation. Les opérateurs sont utilisés dans les expressions pour décrire des opérations impliquant un ou plusieurs opérandes. Par exemple, l'expression `a + b` utilise l'opérateur `+` pour ajouter les deux opérandes `a` et `b`. Les signes de ponctuation sont destinés au regroupement et à la séparation.

antlr

```
operator_or_punctuator
    : '{'  | '}'  | '['  | ']'  | '('   | ')'  | '.'  | ','  | ':'  | ';'
    | '+'  | '-'  | '*'  | '/'  | '%'   | '&'  | '|'  | '^'  | '!'  | '~'
    | '='  | '<'  | '>'  | '?'  | '??'  | '::' | '++' | '--' | '&&' | '||'
    | '->' | '==' | '!=' | '<=' | '>='  | '+=' | '-=' | '*=' | '/=' | '%='
    | '&=' | '|=' | '^=' | '<<' | '<<=' | '=>'
    ;

right_shift
    : '>>'
    ;

right_shift_assignment
    : '>>='
    ;
```

La barre verticale dans le *right_shift* et les productions *right_shift_assignment* sont utilisées pour indiquer que, contrairement aux autres productions dans la syntaxe syntaxique, aucun caractère de type (pas d'espace blanc) n'est autorisé entre les jetons. Ces productions sont traitées de façon spéciale pour permettre la gestion correcte des *type_parameter_list* s (paramètres de type).

Directives de prétraitement

Les directives de prétraitement offrent la possibilité d'ignorer conditionnellement des sections de fichiers sources, de signaler des conditions d'erreur et d'avertissement, et de détourer des régions distinctes du code source. Le terme « directives de prétraitement » est utilisé uniquement à des fins de cohérence avec les langages de programmation C et C++. En C#, il n'y a pas d'étape de pré-traitement distincte ; les directives de pré-traitement sont traitées dans le cadre de la phase d'analyse lexicale.

antlr

```
pp_directive
    : pp_declaration
    | pp_conditional
    | pp_line
    | pp_diagnostic
    | pp_region
    | pp_pragma
    ;
```

Les directives de pré-traitement suivantes sont disponibles :

- `#define` et `#undef` , qui sont utilisés pour définir et annuler la définition, respectivement, des symboles de compilation conditionnelle (directives de déclaration).
- `#if`, `#elif` , `#else` et `#endif` , qui sont utilisés pour ignorer de manière conditionnelle les sections du code source (directives de compilation conditionnelle).
- `#line`, qui est utilisé pour contrôler les numéros de ligne émis pour les erreurs et les avertissements (directives de ligne).
- `#error` et `#warning` , qui sont utilisés pour émettre des erreurs et des avertissements, respectivement (directives de diagnostic).
- `#region` et `#endregion` , qui sont utilisés pour marquer explicitement les sections du code source (directives de région).
- `#pragma`, qui est utilisé pour spécifier des informations contextuelles facultatives pour le compilateur (directives pragma).

Une directive de prétraitement occupe toujours une ligne distincte de code source et commence toujours par un `#` caractère et un nom de directive de prétraitement. Un espace blanc peut se produire avant le `#` caractère et entre le `#` caractère et le nom de la directive.

Une ligne source contenant une `#define` `#undef` directive,, `#if` ,,,, `#elif` `#else` `#endif` `#line` ou `#endregion` peut se terminer par un commentaire sur une seule ligne. Les commentaires délimités (`/* */` style de commentaires) ne sont pas autorisés sur les lignes sources contenant des directives de prétraitement.

Les directives de pré-traitement ne sont pas des jetons et ne font pas partie de la syntaxe syntaxique de C#. Toutefois, les directives de prétraitement peuvent être utilisées pour inclure ou exclure des séquences de jetons et peuvent, de cette façon, affecter la signification d'un programme C#. Par exemple, une fois compilé, le programme :

```
C#
```

```
#define A
#undef B

class C
{
#if A
    void F() {}
#else
    void G() {}
#endif

#if B
    void H() {}
#else
    void I() {}
#endif
}
```

produit exactement la même séquence de jetons que le programme :

C#

```
class C
{
    void F() {}
    void I() {}
}
```

Par conséquent, contrairement à l'lexicale, les deux programmes sont très différents, syntaxiquement identiques.

Symboles de compilation conditionnelle

La fonctionnalité de compilation conditionnelle fournie par `#if` les `#elif` directives,, `#else` et `#endif` est contrôlée par le biais d'expressions de prétraitement (expressions de pré-traitement) et de symboles de compilation conditionnelle.

antlr

```
conditional_symbol
    : '<Any identifier_or_keyword except true or false>'
    ;
```

Un symbole de compilation conditionnelle a deux États possibles : ***défini** _ ou _ *non défini* *. Au début du traitement lexical d'un fichier source, un symbole de compilation conditionnelle n'est pas défini, sauf s'il a été défini explicitement par un mécanisme

externe (tel qu'une option de compilateur de ligne de commande). Quand une `#define` directive est traitée, le symbole de compilation conditionnelle nommé dans cette directive devient défini dans ce fichier source. Le symbole reste défini jusqu'à ce qu'une `#undef` directive pour ce même symbole soit traitée ou jusqu'à ce que la fin du fichier source soit atteinte. L'une des conséquences est que `#define` les `#undef` directives et dans un fichier source n'ont aucun effet sur les autres fichiers sources du même programme.

Lorsqu'il est référencé dans une expression de pré-traitement, un symbole de compilation conditionnelle défini a la valeur booléenne `true` et un symbole de compilation conditionnelle non défini a la valeur booléenne `false` . Il n'est pas nécessaire que les symboles de compilation conditionnelle soient explicitement déclarés avant d'être référencés dans des expressions de prétraitement. Au lieu de cela, les symboles non déclarés sont simplement non définis et ont donc la valeur `false` .

L'espace de noms pour les symboles de compilation conditionnelle est distinct et distinct de toutes les autres entités nommées dans un programme C#. Les symboles de compilation conditionnelle peuvent uniquement être référencés dans `#define` les `#undef` directives et et dans les expressions de prétraitement.

Expressions de pré-traitement

Les expressions de prétraitement peuvent se produire dans `#if` les `#elif` directives et. Les opérateurs `!` , `==` , `!=` `&&` et `||` sont autorisés dans les expressions de prétraitement, et les parenthèses peuvent être utilisées pour le regroupement.

antlr

```
pp_expression
    : whitespace? pp_or_expression whitespace?
    ;

pp_or_expression
    : pp_and_expression
    | pp_or_expression whitespace? '||' whitespace? pp_and_expression
    ;

pp_and_expression
    : pp_equality_expression
    | pp_and_expression whitespace? '&&' whitespace? pp_equality_expression
    ;

pp_equality_expression
    : pp_unary_expression
    | pp_equality_expression whitespace? '==' whitespace?
```

```
pp_unary_expression
    | pp_equality_expression whitespace? '!=' whitespace?
pp_unary_expression
    ;

pp_unary_expression
    : pp_primary_expression
    | '!' whitespace? pp_unary_expression
    ;

pp_primary_expression
    : 'true'
    | 'false'
    | conditional_symbol
    | '(' whitespace? pp_expression whitespace? ')'
    ;
```

Lorsqu'il est référencé dans une expression de pré-traitement, un symbole de compilation conditionnelle défini a la valeur booléenne `true` et un symbole de compilation conditionnelle non défini a la valeur booléenne `false` .

L'évaluation d'une expression de prétraitement génère toujours une valeur booléenne. Les règles d'évaluation pour une expression de prétraitement sont les mêmes que celles d'une expression constante (expressions constantes), sauf que les seules entités définies par l'utilisateur qui peuvent être référencées sont des symboles de compilation conditionnelle.

Directives de déclaration

Les directives de déclaration sont utilisées pour définir ou annuler la définition des symboles de compilation conditionnelle.

antlr

```
pp_declaration
    : whitespace? '#' whitespace? 'define' whitespace conditional_symbol
pp_new_line
    | whitespace? '#' whitespace? 'undef' whitespace conditional_symbol
pp_new_line
    ;

pp_new_line
    : whitespace? single_line_comment? new_line
    ;
```

Le traitement d'une `#define` directive a pour effet de définir le symbole de compilation conditionnelle donné, en commençant par la ligne source qui suit la directive. De même,

le traitement d'une `#undef` directive rend le symbole de compilation conditionnelle donné non défini, en commençant par la ligne source qui suit la directive.

Les `#define` `#undef` directives et dans un fichier source doivent se trouver avant le premier *jeton* (jetons) dans le fichier source ; sinon, une erreur de compilation se produit. En termes intuitifs, les `#define` `#undef` directives et doivent précéder tout « code réel » dans le fichier source.

L'exemple :

C#

```
#define Enterprise

#if Professional || Enterprise
    #define Advanced
#endif

namespace Megacorp.Data
{
    #if Advanced
    class PivotTable {...}
    #endif
}
```

est valide, car les `#define` directives précèdent le premier jeton (le `namespace` mot clé) dans le fichier source.

L'exemple suivant génère une erreur au moment de la compilation, car un `#define` code réel suit :

C#

```
#define A
namespace N
{
    #define B
    #if B
    class Class1 {}
    #endif
}
```

Un `#define` peut définir un symbole de compilation conditionnelle qui est déjà défini, sans qu'il n'y ait d'intermédiaire `#undef` pour ce symbole. L'exemple ci-dessous définit un symbole de compilation conditionnelle `A` , puis le définit à nouveau.

C#

```
#define A
#define A
```

Un `#undef` peut « annuler la définition » d'un symbole de compilation conditionnelle qui n'est pas défini. L'exemple ci-dessous définit un symbole de compilation conditionnelle `A` , puis l'annule deux fois ; bien que le second n' `#undef` ait aucun effet, il est toujours valide.

C#

```
#define A
#undef A
#undef A
```

Directives de compilation conditionnelle

Les directives de compilation conditionnelle permettent d'inclure ou d'exclure de manière conditionnelle des parties d'un fichier source.

antlr

```
pp_conditional
    : pp_if_section pp_elif_section* pp_else_section? pp_endif
    ;

pp_if_section
    : whitespace? '#' whitespace? 'if' whitespace pp_expression pp_new_line
conditional_section?
    ;

pp_elif_section
    : whitespace? '#' whitespace? 'elif' whitespace pp_expression
pp_new_line conditional_section?
    ;

pp_else_section:
    | whitespace? '#' whitespace? 'else' pp_new_line conditional_section?
    ;

pp_endif
    : whitespace? '#' whitespace? 'endif' pp_new_line
    ;

conditional_section
    : input_section
    | skipped_section
    ;
```

```
skipped_section
    : skipped_section_part+
    ;

skipped_section_part
    : skipped_characters? new_line
    | pp_directive
    ;

skipped_characters
    : whitespace? not_number_sign input_character*
    ;

not_number_sign
    : '<Any input_character except #>'
    ;
```

Comme indiqué par la syntaxe, les directives de compilation conditionnelle doivent être écrites sous la forme de jeux composés de, dans l'ordre, d'une directive, de zéro, d'une `#if` ou de plusieurs `#elif` directives, de zéro ou d'une directive `#else` et d'une `#endif` directive. Les sections conditionnelles du code source sont placées entre les directives. Chaque section est contrôlée par la directive qui précède immédiatement. Une section conditionnelle peut elle-même contenir des directives de compilation conditionnelle imbriquées, à condition que ces directives forment des jeux complets.

Une *pp_conditional* sélectionne au maximum l'un des *conditional_section* s contenus pour le traitement lexical normal :

- Les *pp_expression* s des `#if` directives et `#elif` sont évaluées dans l'ordre jusqu'à ce que l'un d'eux produise `true` . Si une expression génère `true` , le *conditional_section* de la directive correspondante est sélectionné.
- Si tous les *pp_expression* s `false` sont générés et, si une `#else` directive est présente, la *conditional_section* de la `#else` directive est sélectionnée.
- Dans le cas contraire, aucune *conditional_section* n'est sélectionnée.

Le *conditional_section* sélectionné, le cas échéant, est traité comme un *input_section* normal : le code source contenu dans la section doit adhérer à la grammaire lexicale. les jetons sont générés à partir du code source dans la section. et les directives de pré-traitement dans la section ont les effets imposés.

Les autres *conditional_section* s, le cas échéant, sont traités comme des *skipped_section* s : à l'exception des directives de prétraitement, le code source de la section ne doit pas être conforme à la grammaire lexicale. aucun jeton n'est généré à partir du code source dans la section ; et les directives de prétraitement dans la section doivent être lexicalement correctes, mais elles ne sont pas traitées autrement. Au sein d'une

conditional_section traitée en tant que *skipped_section*, les *conditional_section* imbriqués (contenues dans des constructions imbriquées. `#if` .. `#endif` et `#region` ... `#endregion`) sont également traitées comme des *skipped_section* s.

L'exemple suivant illustre la façon dont les directives de compilation conditionnelle peuvent imbriquer :

C#

```
#define Debug       // Debugging on
#undef Trace        // Tracing off

class PurchaseTransaction
{
    void Commit() {
        #if Debug
            CheckConsistency();
            #if Trace
                WriteToLog(this.ToString());
            #endif
        #endif
        CommitHelper();
    }
}
```

À l'exception des directives de pré-traitement, le code source ignoré n'est pas soumis à l'analyse lexicale. Par exemple, le code suivant est valide malgré le Commentaire inachevé dans la `#else` section :

C#

```
#define Debug        // Debugging on

class PurchaseTransaction
{
    void Commit() {
        #if Debug
            CheckConsistency();
        #else
            /* Do something else
        #endif
    }
}
```

Notez, toutefois, que les directives de pré-traitement doivent être correctes lexicalement, même dans les sections ignorées du code source.

Les directives de pré-traitement ne sont pas traitées quand elles apparaissent dans des éléments d'entrée multiligne. Par exemple, le programme :

C#

```
class Hello
{
    static void Main() {
        System.Console.WriteLine(@"hello,
#if Debug
        world
#else
        Nebraska
#endif
        ");
    }
}
```

génère la sortie :

Console

```
hello,
#if Debug
        world
#else
        Nebraska
#endif
```

Dans les cas particuliers, l'ensemble de directives de prétraitement qui est traité peut dépendre de l'évaluation de la *pp_expression*. L'exemple :

C#

```
#if X
    /*
#else
    /* */ class Q { }
#endif
```

produit toujours le même flux de jetons (`class` `Q` `{` `}`), qu'il soit défini ou non `X` . Si `X` est défini, les seules directives traitées sont `#if` et `#endif` , en raison du commentaire sur plusieurs lignes. Si `X` n'est pas défini, alors trois directives (`#if` , `#else` , `#endif`) font partie de la directive définie.

Directives de diagnostic

Les directives de diagnostic sont utilisées pour générer explicitement des messages d'erreur et d'avertissement signalés de la même façon que d'autres erreurs et avertissements au moment de la compilation.

antlr

```
pp_diagnostic
    : whitespace? '#' whitespace? 'error' pp_message
    | whitespace? '#' whitespace? 'warning' pp_message
    ;

pp_message
    : new_line
    | whitespace input_character* new_line
    ;
```

L'exemple :

C#

```
#warning Code review needed before check-in

#if Debug && Retail
    #error A build can't be both debug and retail
#endif

class Test {...}
```

génère toujours un avertissement (« révision du code nécessaire avant l'archivage ») et génère une erreur au moment de la compilation (« une build ne peut pas être à la fois Debug et Retail ») si les symboles conditionnels `Debug` et `Retail` sont tous deux définis. Notez qu'une *pp_message* peut contenir du texte arbitraire. plus précisément, il n'a pas besoin de contenir des jetons correctement formés, comme indiqué par le guillemet simple dans le mot `can't` .

Directives de région

Les directives de région sont utilisées pour marquer explicitement les zones de code source.

antlr

```
pp_region
    : pp_start_region conditional_section? pp_end_region
    ;
```

```
pp_start_region
    : whitespace? '#' whitespace? 'region' pp_message
    ;

pp_end_region
    : whitespace? '#' whitespace? 'endregion' pp_message
    ;
```

Aucune signification sémantique n'est attachée à une région ; les régions sont destinées à être utilisées par le programmeur ou par des outils automatisés pour marquer une section du code source. Le message spécifié dans une `#region` directive ou n' `#endregion` a pas la même signification sémantique ; il sert simplement à identifier la région. `#region`Les `#endregion` directives et peuvent avoir différentes *pp_message* s.

Traitement lexical d'une région :

C#

```
#region
...
#endregion
```

correspond exactement au traitement lexical d'une directive de compilation conditionnelle au format suivant :

C#

```
#if true
...
#endif
```

Directives de ligne

Les directives de ligne peuvent être utilisées pour modifier les numéros de ligne et les noms de fichiers sources qui sont signalés par le compilateur dans la sortie, tels que les avertissements et les erreurs, et qui sont utilisés par les attributs d'informations de l'appelant (attributs des informationsde l'appelant).

Les directives de ligne sont le plus souvent utilisées dans les outils de méta-programmation qui génèrent du code source C# à partir d'autres entrées de texte.

antlr

```
pp_line
    : whitespace? '#' whitespace? 'line' whitespace line_indicator
```

```
pp_new_line
    ;

line_indicator
    : decimal_digit+ whitespace file_name
    | decimal_digit+
    | 'default'
    | 'hidden'
    ;

file_name
    : '"' file_name_character+ '"'
    ;

file_name_character
    : '<Any input_character except ">'
    ;
```

Quand aucune `#line` directive n'est présente, le compilateur signale les véritables numéros de ligne et les noms de fichiers sources dans sa sortie. Lors du traitement d'une `#line` directive qui comprend un *line_indicator* qui n'est pas `default` , le compilateur traite la ligne après la directive comme ayant le numéro de ligne donné (et le nom de fichier, si spécifié).

Une `#line default` directive inverse l'effet de toutes les directives de #line précédentes. Le compilateur signale des informations de ligne vraies pour les lignes suivantes, exactement comme si aucune `#line` directive n'avait été traitée.

Une `#line hidden` directive n'a aucun effet sur le fichier et les numéros de ligne signalés dans les messages d'erreur, mais affecte le débogage au niveau de la source. Lors du débogage, toutes les lignes entre une `#line hidden` directive et la `#line` directive suivante (qui n'est pas `#line hidden`) n'ont pas d'informations sur le numéro de ligne. Quand vous exécutez le code pas à pas dans le débogueur, ces lignes sont entièrement ignorées.

Notez qu'un *file_name* diffère d'un littéral de chaîne standard en ce que les caractères d'échappement ne sont pas traités ; le `\` caractère «» désigne simplement un caractère barre oblique inverse ordinaire dans un *file_name*.

Directives pragma

La `#pragma` directive de prétraitement est utilisée pour spécifier des informations contextuelles facultatives pour le compilateur. Les informations fournies dans une `#pragma` directive ne modifient jamais la sémantique du programme.

antlr

```
pp_pragma
    : whitespace? '#' whitespace? 'pragma' whitespace pragma_body
pp_new_line
    ;

pragma_body
    : pragma_warning_body
    ;
```

C# fournit des `#pragma` directives pour contrôler les avertissements du compilateur. Les futures versions du langage peuvent inclure des `#pragma` directives supplémentaires. Pour garantir l'interopérabilité avec d'autres compilateurs C#, le compilateur Microsoft C# n'émet pas d'erreurs de compilation pour les directives inconnues `#pragma` ; ces directives génèrent toutefois des avertissements.

Pragma warning

La `#pragma warning` directive est utilisée pour désactiver ou restaurer tout ou un ensemble particulier de messages d'avertissement lors de la compilation du texte de programme suivant.

antlr

```
pragma_warning_body
    : 'warning' whitespace warning_action
    | 'warning' whitespace warning_action whitespace warning_list
    ;

warning_action
    : 'disable'
    | 'restore'
    ;

warning_list
    : decimal_digit+ (whitespace? ',' whitespace? decimal_digit+)*
    ;
```

Une `#pragma warning` directive qui omet la liste d'avertissements affecte tous les avertissements. Une `#pragma warning` directive qui comprend une liste d'avertissements affecte uniquement les avertissements qui sont spécifiés dans la liste.

Une `#pragma warning disable` directive désactive tout ou l'ensemble d'avertissements donné.

Une `#pragma warning restore` directive restaure tout ou partie des avertissements donnés à l'État qui était en vigueur au début de l'unité de compilation. Notez que si un avertissement particulier a été désactivé en externe, a `#pragma warning restore` (que ce soit pour tous ou l'avertissement spécifique) ne réactivera pas cet avertissement.

L'exemple suivant illustre l'utilisation de `#pragma warning` pour désactiver temporairement l'avertissement signalé quand des membres obsolètes sont référencés, à l'aide du numéro d'avertissement du compilateur Microsoft C#.

C#

```
using System;

class Program
{
    [Obsolete]
    static void Foo() {}

    static void Main() {
#pragma warning disable 612
    Foo();
#pragma warning restore 612
    }
}
```

Concepts de base

Article • 13/05/2021

Démarrage de l'application

Un assembly qui a un **point d'entrée*** _ est appelé *application*. Lorsqu'une application est exécutée, un nouveau *domaine d'application** est créé. Plusieurs instanciations différentes d'une application peuvent exister sur le même ordinateur en même temps et chacune possède son propre domaine d'application.

Un domaine d'application permet l'isolation d'applications en jouant le rôle d'un conteneur pour l'état de l'application. Un domaine d'application agit comme un conteneur et une limite pour les types définis dans l'application et les bibliothèques de classes qu'il utilise. Les types chargés dans un domaine d'application sont différents du même type chargé dans un autre domaine d'application, et les instances d'objets ne sont pas directement partagées entre les domaines d'application. Par exemple, chaque domaine d'application possède sa propre copie de variables statiques pour ces types, et un constructeur statique pour un type est exécuté au plus une fois par domaine d'application. Les implémentations sont gratuites pour fournir une stratégie ou des mécanismes spécifiques à l'implémentation pour la création et la destruction de domaines d'application.

Le démarrage de l' ***application*** se produit lorsque l'environnement d'exécution appelle une méthode désignée, appelée point d'entrée de l'application. Cette méthode de point d'entrée est toujours nommée `Main` et peut avoir l'une des signatures suivantes :

C#

```
static void Main() {...}

static void Main(string[] args) {...}

static int Main() {...}

static int Main(string[] args) {...}
```

Comme indiqué, le point d'entrée peut éventuellement retourner une `int` valeur. Cette valeur de retour est utilisée dans l'arrêt de l'application (arrêt de l'application).

Le point d'entrée peut éventuellement avoir un paramètre formel. Le paramètre peut avoir n'importe quel nom, mais le type du paramètre doit être `string[]` . Si le paramètre formel est présent, l'environnement d'exécution crée et passe un `string[]`

argument contenant les arguments de ligne de commande spécifiés au démarrage de l'application. L' `string[]` argument n'est jamais null, mais peut avoir une longueur de zéro si aucun argument de ligne de commande n'a été spécifié.

Étant donné que C# prend en charge la surcharge de méthode, une classe ou un struct peut contenir plusieurs définitions d'une méthode, à condition que chacune ait une signature différente. Toutefois, au sein d'un même programme, aucune classe ou structure ne peut contenir plusieurs méthodes appelées `Main` dont la définition la qualifie à utiliser comme point d'entrée de l'application. Toutefois, d'autres versions surchargées de `Main` sont autorisées, à condition qu'elles aient plusieurs paramètres, ou que leur seul paramètre soit autre que type `string[]` .

Une application peut être composée de plusieurs classes ou structures. Il est possible que plus d'une de ces classes ou structures contiennent une méthode appelée dont la `Main` définition la qualifie à utiliser comme point d'entrée de l'application. Dans ce cas, un mécanisme externe (tel qu'une option du compilateur de ligne de commande) doit être utilisé pour sélectionner l'une de ces `Main` méthodes comme point d'entrée.

En C#, chaque méthode doit être définie en tant que membre d'une classe ou d'un struct. En règle générale, l'accessibilité déclarée (accessibilité déclarée) d'une méthode est déterminée par les modificateurs d'accès (modificateurs d'accès) spécifiés dans sa déclaration, et de la même façon, l'accessibilité déclarée d'un type est déterminée par les modificateurs d'accès spécifiés dans sa déclaration. Pour qu'une méthode donnée d'un type donné puisse être appelée, le type et le membre doivent être accessibles. Toutefois, le point d'entrée de l'application est un cas particulier. Plus précisément, l'environnement d'exécution peut accéder au point d'entrée de l'application, quelle que soit l'accessibilité déclarée et indépendamment de l'accessibilité déclarée de ses déclarations de type englobant.

La méthode de point d'entrée de l'application ne peut pas figurer dans une déclaration de classe générique.

Dans tous les autres aspects, les méthodes de point d'entrée se comportent comme celles qui ne sont pas des points d'entrée.

Arrêt d'une application

L'arrêt de l' ***application*** retourne le contrôle à l'environnement d'exécution.

Si le type de retour de la méthode **point d'entrée*** de l'application est `int` , la valeur retournée est le code d'*État* _ * de fin de l'application * *. L'objectif de ce code est de permettre la communication de la réussite ou de l'échec à l'environnement d'exécution.

Si le type de retour de la méthode de point d'entrée est, si vous `void` atteignez l'accolade fermante (`}`) qui termine cette méthode ou si vous exécutez une `return` instruction qui n'a pas d'expression, le code d'état d'arrêt est `0` .

Avant l'arrêt d'une application, les destructeurs de tous ses objets qui n'ont pas encore été récupérés par le garbage collector sont appelés, à moins que ce nettoyage ait été supprimé (par exemple, par un appel à la méthode de la bibliothèque `GC.SuppressFinalize`).

Déclarations

Dans un programme C#, les déclarations définissent les éléments constitutifs du programme. Les programmes C# sont organisés à l'aide d'espaces de noms (espaces de noms), qui peuvent contenir des déclarations de type et des déclarations d'espaces de noms imbriqués. Les déclarations de type (déclarations de type) sont utilisées pour définir des classes (classes), des structures (structs), des interfaces (interfaces), des enums (énumérations) et des délégués (délégués). Les genres de membres autorisés dans une déclaration de type dépendent de la forme de la déclaration de type. Par exemple, les déclarations de classe peuvent contenir des déclarations pour les constantes (constantes), champs (champs), méthodes (méthodes), propriétés (Propriétés), événements (événements), indexeurs (indexeurs), opérateurs (opérateurs), constructeurs d'instance (constructeurs d'instance), constructeurs statiques (constructeurs statiques), destructeurs (destructeurs) et types imbriqués (types imbriqués).

Une déclaration définit un nom dans l' ***espace de déclaration*** auquel la déclaration appartient. À l'exception des membres surchargés (signatures et surcharge), il s'agit d'une erreur au moment de la compilation d'avoir au moins deux déclarations qui introduisent des membres portant le même nom dans un espace de déclaration. Il n'est jamais possible pour un espace de déclaration de contenir différents genres de membres portant le même nom. Par exemple, un espace de déclaration ne peut jamais contenir un champ et une méthode du même nom.

Il existe différents types d'espaces de déclaration, comme décrit dans l'exemple suivant.

- Dans tous les fichiers sources d'un programme, *namespace_member_declaration* s sans *namespace_declaration* englobante sont membres d'un espace de déclaration combiné unique, appelé ***espace de déclaration globale***.
- Dans tous les fichiers sources d'un programme, *namespace_member_declaration* dans *namespace_declaration* s qui ont le même nom d'espace de noms qualifié complet sont membres d'un espace de déclaration combiné unique.

- Chaque déclaration de classe, de struct ou d'interface crée un nouvel espace de déclaration. Les noms sont introduits dans cet espace de déclaration par le biais de *class_member_declaration* s, *struct_member_declaration* s, *interface_member_declaration* s ou *type_parameter* s. Sauf pour les déclarations de constructeur d'instance surchargé et les déclarations de constructeur statiques, une classe ou un struct ne peut pas contenir une déclaration de membre portant le même nom que la classe ou le struct. Une classe, un struct ou une interface autorise la déclaration de méthodes surchargées et d'indexeurs. En outre, une classe ou un struct autorise la déclaration de constructeurs et d'opérateurs d'instance surchargés. Par exemple, une classe, un struct ou une interface peut contenir plusieurs déclarations de méthode portant le même nom, à condition que ces déclarations de méthode diffèrent dans leur signature (signatures et surcharge). Notez que les classes de base ne contribuent pas à l'espace de déclaration d'une classe et que les interfaces de base ne contribuent pas à l'espace de déclaration d'une interface. Ainsi, une classe ou une interface dérivée est autorisée à déclarer un membre portant le même nom qu'un membre hérité. Ce membre est dit de ***Masquer*** le membre hérité.
- Chaque déclaration Delegate crée un nouvel espace de déclaration. Les noms sont introduits dans cet espace de déclaration via des paramètres formels (*fixed_parameter* s et *parameter_array* s) et *type_parameter* s.
- Chaque déclaration d'énumération crée un nouvel espace de déclaration. Les noms sont introduits dans cet espace de déclaration par le biais de *enum_member_declarations*.
- Chaque déclaration de méthode, déclaration d'indexeur, déclaration d'opérateur, déclaration de constructeur d'instance et fonction anonyme crée un espace de déclaration appelé «**espace de déclaration de variable locale**». Les noms sont introduits dans cet espace de déclaration via des paramètres formels (_fixed_parameter * s et *parameter_array* s) et *type_parameter* s. Le corps de la fonction membre ou de la fonction anonyme, le cas échéant, est considéré comme imbriqué dans l'espace de déclaration de la variable locale. Il s'agit d'une erreur pour un espace de déclaration de variable locale et un espace de déclaration de variable locale imbriqué pour contenir des éléments portant le même nom. Ainsi, dans un espace de déclaration imbriqué, il n'est pas possible de déclarer une variable locale ou une constante portant le même nom qu'une variable locale ou une constante dans un espace de déclaration englobant. Il est possible que deux espaces de déclaration contiennent des éléments portant le même nom, à condition que l'espace de déclaration ne contienne pas l'autre.
- Chaque *bloc* ou *switch_block* , ainsi qu'une instruction *for*, *foreach* et *using* , crée un espace de déclaration de variable locale pour les variables locales et les constantes locales. Les noms sont introduits dans cet espace de déclaration par le biais de

local_variable_declaration s et *local_constant_declaration* s. Notez que les blocs qui se produisent en tant que ou dans le corps d'un membre de fonction ou d'une fonction anonyme sont imbriqués dans l'espace de déclaration de variable locale déclaré par ces fonctions pour leurs paramètres. Par conséquent, il s'agit d'une erreur d'avoir une méthode avec une variable locale et un paramètre du même nom.

- Chaque *bloc* ou *switch_block* crée un espace de déclaration distinct pour les étiquettes. Les noms sont introduits dans cet espace de déclaration par le biais de *labeled_statement* s, et les noms sont référencés par le biais de *goto_statement* s. L'***espace de déclaration d'étiquette*** d'un bloc comprend tous les blocs imbriqués. Ainsi, dans un bloc imbriqué, il n'est pas possible de déclarer une étiquette portant le même nom qu'une étiquette dans un bloc englobant.

L'ordre textuel dans lequel les noms sont déclarés n'a généralement pas d'importance. En particulier, l'ordre textuel n'est pas significatif pour la déclaration et l'utilisation des espaces de noms, des constantes, des méthodes, des propriétés, des événements, des indexeurs, des opérateurs, des constructeurs d'instance, des destructeurs, des constructeurs statiques et des types. L'ordre de déclaration est significatif des manières suivantes :

- L'ordre de déclaration pour les déclarations de champ et les déclarations de variable locale détermine l'ordre dans lequel leurs initialiseurs (le cas échéant) sont exécutés.
- Les variables locales doivent être définies avant d'être utilisées (étendues).
- L'ordre de déclaration pour les déclarations de membre enum (membres enum) est significatif lorsque *constant_expression* valeurs sont omises.

L'espace de déclaration d'un espace de noms est « Open Closed » et deux déclarations d'espaces de noms portant le même nom qualifié complet participent au même espace de déclaration. Par exemple

C#

```
namespace Megacorp.Data
{
    class Customer
    {
        ...
    }
}

namespace Megacorp.Data
{
    class Order
    {
```

```
        ...
    }
}
```

Les deux déclarations d'espaces de noms ci-dessus contribuent au même espace de déclaration, dans ce cas, en déclarant deux classes avec les noms qualifiés complets `Megacorp.Data.Customer` et `Megacorp.Data.Order` . Étant donné que les deux déclarations contribuent au même espace de déclaration, elles auraient provoqué une erreur au moment de la compilation si chacune contient une déclaration d'une classe portant le même nom.

Comme indiqué ci-dessus, l'espace de déclaration d'un bloc comprend tous les blocs imbriqués. Ainsi, dans l'exemple suivant, les `F` `G` méthodes et entraînent une erreur de compilation, car le nom `i` est déclaré dans le bloc externe et ne peut pas être redéclaré dans le bloc interne. Toutefois, les `H` `I` méthodes et sont valides, car les deux `i` sont déclarées dans des blocs non imbriqués distincts.

C#

```
class A
{
    void F() {
        int i = 0;
        if (true) {
            int i = 1;
        }
    }

    void G() {
        if (true) {
            int i = 0;
        }
        int i = 1;
    }

    void H() {
        if (true) {
            int i = 0;
        }
        if (true) {
            int i = 1;
        }
    }

    void I() {
        for (int i = 0; i < 10; i++)
            H();
        for (int i = 0; i < 10; i++)
            H();
```

```
    }
}
```

Membres

Les espaces de noms et les types ont des ***membres***. Les membres d'une entité sont généralement disponibles via l'utilisation d'un nom qualifié qui commence par une référence à l'entité, suivie d'un `.` jeton «», suivi du nom du membre.

Les membres d'un type sont déclarés dans la déclaration de type ou ***hérités*** de la classe de base du type. Lorsqu'un type hérite d'une classe de base, tous les membres de la classe de base, à l'exception des constructeurs d'instance, des destructeurs et des constructeurs statiques, deviennent membres du type dérivé. L'accessibilité déclarée d'un membre de classe de base ne contrôle pas si le membre est hérité ; l'héritage s'étend à tout membre qui n'est pas un constructeur d'instance, un constructeur statique ou un destructeur. Toutefois, un membre hérité peut ne pas être accessible dans un type dérivé, soit en raison de son accessibilité déclarée (accessibilité déclarée), soit parce qu'il est masqué par une déclaration dans le type lui-même (masquage par héritage).

Membres d'espaces de noms

Les espaces de noms et les types qui n'ont pas d'espace de noms englobant sont membres de l' ***espace de noms global***. Cela correspond directement aux noms déclarés dans l'espace de déclaration globale.

Les espaces de noms et les types déclarés dans un espace de noms sont membres de cet espace de noms. Cela correspond directement aux noms déclarés dans l'espace de déclaration de l'espace de noms.

Les espaces de noms ne présentent aucune limitation d'accès. Il n'est pas possible de déclarer des espaces de noms privés, protégés ou internes, et les noms d'espaces de noms sont toujours accessibles publiquement.

Membres de struct

Les membres d'un struct sont les membres déclarés dans le struct et les membres hérités de la classe de base directe du struct `System.ValueType` et de la classe de base indirecte `object` .

Les membres d'un type simple correspondent directement aux membres du type struct dont le type est simple :

- Les membres de `sbyte` sont les membres de la `System.SByte` structure.
- Les membres de `byte` sont les membres de la `System.Byte` structure.
- Les membres de `short` sont les membres de la `System.Int16` structure.
- Les membres de `ushort` sont les membres de la `System.UInt16` structure.
- Les membres de `int` sont les membres de la `System.Int32` structure.
- Les membres de `uint` sont les membres de la `System.UInt32` structure.
- Les membres de `long` sont les membres de la `System.Int64` structure.
- Les membres de `ulong` sont les membres de la `System.UInt64` structure.
- Les membres de `char` sont les membres de la `System.Char` structure.
- Les membres de `float` sont les membres de la `System.Single` structure.
- Les membres de `double` sont les membres de la `System.Double` structure.
- Les membres de `decimal` sont les membres de la `System.Decimal` structure.
- Les membres de `bool` sont les membres de la `System.Boolean` structure.

Membres de l'énumération

Les membres d'une énumération sont les constantes déclarées dans l'énumération et les membres hérités de la classe de base directe de l'énumération, ainsi que `System.Enum` les classes de base indirectes et `System.ValueType` `object` .

Membres de classe

Les membres d'une classe sont les membres déclarés dans la classe et les membres hérités de la classe de base (à l'exception de la classe `object` qui n'a pas de classe de base). Les membres hérités de la classe de base incluent les constantes, les champs, les méthodes, les propriétés, les événements, les indexeurs, les opérateurs et les types de la classe de base, mais pas les constructeurs d'instance, les destructeurs et les constructeurs statiques de la classe de base. Les membres de la classe de base sont hérités sans tenir compte de leur accessibilité.

Une déclaration de classe peut contenir des déclarations de constantes, des champs, des méthodes, des propriétés, des événements, des indexeurs, des opérateurs, des constructeurs d'instance, des destructeurs, des constructeurs statiques et des types.

Les membres de `object` et `string` correspondent directement aux membres des types de classe qu'ils aliasent :

- Les membres de `object` sont les membres de la `System.Object` classe.
- Les membres de `string` sont les membres de la `System.String` classe.

Membres d'interfaces

Les membres d'une interface sont les membres déclarés dans l'interface et dans toutes les interfaces de base de l'interface. Les membres de la classe `object` ne sont pas, strictement parlant, des membres de n'importe quelle interface (membres d'interface). Toutefois, les membres de la classe `object` sont disponibles via la recherche de membres dans n'importe quel type d'interface (recherche de membre).

Membres de tableau

Les membres d'un tableau sont les membres hérités de la classe `System.Array` .

Membres délégués

Les membres d'un délégué sont les membres hérités de la classe `System.Delegate` .

Accès au membre

Les déclarations de membres permettent de contrôler l'accès aux membres. L'accessibilité d'un membre est établie par l'accessibilité déclarée (accessibilité déclarée) du membre combiné avec l'accessibilité du type conteneur immédiatement, le cas échéant.

Lorsque l'accès à un membre particulier est autorisé, le membre est dit : ***accessible** _. Inversement, lorsque l'accès à un membre particulier n'est pas autorisé, le membre est dit _ *inaccessible* *. L'accès à un membre est autorisé lorsque l'emplacement du texte dans lequel l'accès a lieu est inclus dans le domaine d'accessibilité (domaines d'accessibilité) du membre.

Accessibilité déclarée

L' ***accessibilité déclarée*** d'un membre peut être l'une des suivantes :

- Public, qui est sélectionné en incluant un `public` modificateur dans la déclaration de membre. La signification intuitive de `public` est « accès non limité ».
- Protégé, qui est sélectionné en incluant un `protected` modificateur dans la déclaration de membre. La signification intuitive de `protected` est « accès limité à

la classe conteneur ou aux types dérivés de la classe conteneur ».
- Interne, qui est sélectionné en incluant un `internal` modificateur dans la déclaration de membre. La signification intuitive de `internal` est « accès limité à ce programme ».
- Protected internal (c'est-à-dire protected ou Internal), qui est sélectionné en incluant à la fois un `protected` et un `internal` modificateur dans la déclaration de membre. La signification intuitive de `protected internal` est « accès limité à ce programme ou aux types dérivés de la classe conteneur ».
- Privé, qui est sélectionné en incluant un `private` modificateur dans la déclaration de membre. La signification intuitive de `private` est « accès limité au type conteneur ».

En fonction du contexte dans lequel la déclaration d'un membre a lieu, seuls certains types d'accessibilité déclarés sont autorisés. En outre, lorsqu'une déclaration de membre n'inclut pas de modificateur d'accès, le contexte dans lequel la déclaration a lieu détermine l'accessibilité déclarée par défaut.

- Les espaces de noms ont implicitement l' `public` accessibilité déclarée. Aucun modificateur d'accès n'est autorisé sur les déclarations d'espace de noms.
- Les types déclarés dans les unités de compilation ou les espaces de noms peuvent avoir `public` ou `internal` déclarés accessibilité et par défaut pour l' `internal` accessibilité déclarée.
- Les membres de classe peuvent avoir l'un des cinq types d'accessibilité déclarée et la valeur par défaut est l' `private` accessibilité déclarée. (Notez qu'un type déclaré en tant que membre d'une classe peut avoir l'un des cinq types d'accessibilité déclarés, alors qu'un type déclaré en tant que membre d'un espace de noms ne peut avoir que l' `public` `internal` accessibilité ou déclarée.)
- Les membres de struct peuvent avoir une `public` `internal` `private` accessibilité déclarée, ou déclarée et la valeur par défaut pour l' `private` accessibilité déclarée, car les structs sont implicitement sealed. Les membres de struct introduits dans un struct (c'est-à-dire, qui n'est pas hérité par ce struct) ne peuvent pas avoir ou avoir une `protected` `protected internal` accessibilité déclarée. (Notez qu'un type déclaré en tant que membre d'un struct peut avoir une `public` `internal` accessibilité, ou `private` déclarée, alors qu'un type déclaré comme membre d'un espace de noms ne peut avoir que l' `public` `internal` accessibilité ou déclarée.)
- Les membres d'interface ont implicitement une `public` accessibilité déclarée. Aucun modificateur d'accès n'est autorisé sur les déclarations de membres d'interface.
- Les membres de l'énumération ont implicitement une `public` accessibilité déclarée. Aucun modificateur d'accès n'est autorisé sur les déclarations de membre

d'énumération.

Domaines d'accessibilité

Le **domaine d'accessibilité** _ d'un membre se compose des sections (éventuellement disjointes) du texte de programme dans lesquelles l'accès au membre est autorisé. Dans le cadre de la définition du domaine d'accessibilité d'un membre, un membre est dit comme étant de *niveau supérieur* s'il n'est pas déclaré dans un type, et un membre est dit *imbriqué* s'il est déclaré dans un autre type. En outre, le *texte de programme* d'un programme est défini comme tout le texte de programme contenu dans tous les fichiers sources du programme, et le texte de programme d'un type est défini comme tout le texte de programme contenu dans le _type_declaration * s de ce type (y compris, éventuellement, les types imbriqués dans le type).

Le domaine d'accessibilité d'un type prédéfini (tel que `object` , `int` ou `double`) est illimité.

Le domaine d'accessibilité d'un type indépendant de niveau supérieur `T` (types liés et indépendants) qui est déclaré dans un programme `P` est défini comme suit :

- Si l'accessibilité déclarée de `T` est `public` , le domaine d'accessibilité de `T` est le texte de programme de `P` et tout programme qui référence `P` .
- Si l'accessibilité déclarée de `T` est `internal`, le domaine d'accessibilité de `T` correspond au texte de programme de `P`.

À partir de ces définitions, il en résulte que le domaine d'accessibilité d'un type indépendant de niveau supérieur est toujours au moins le texte de programme du programme dans lequel ce type est déclaré.

Le domaine d'accessibilité d'un type construit `T<A1, ..., An>` est l'intersection du domaine d'accessibilité du type générique indépendant `T` et des domaines d'accessibilité des arguments de type `A1, ..., An` .

Le domaine d'accessibilité d'un membre imbriqué `M` déclaré dans un type `T` au sein d'un programme est défini de la manière `P` suivante (en notant qu' `M` il peut s'agir d'un type) :

- Si l'accessibilité déclarée de `M` est `public`, le domaine d'accessibilité de `M` correspond à celui de `T`.
- Si l'accessibilité déclarée de `M` est `protected internal` , donnez `D` l'Union du texte de programme de `P` et du texte de programme de n'importe quel type dérivé de

`T` , qui est déclaré à l'extérieur de `P` . Le domaine d'accessibilité de `M` correspond à l'intersection du domaine d'accessibilité de `T` avec `D` .

- Si l'accessibilité déclarée de `M` est `protected` , donnez `D` l'Union du texte de programme de `T` et du texte de programme de n'importe quel type dérivé de `T` . Le domaine d'accessibilité de `M` correspond à l'intersection du domaine d'accessibilité de `T` avec `D` .
- Si l'accessibilité déclarée de `M` est `internal`, le domaine d'accessibilité de `M` correspond à l'intersection du domaine d'accessibilité de `T` avec le texte de programme de `P`.
- Si l'accessibilité déclarée de `M` est `private`, le domaine d'accessibilité de `M` correspond au texte de programme de `T`.

À partir de ces définitions, il s'ensuit que le domaine d'accessibilité d'un membre imbriqué est toujours au moins le texte de programme du type dans lequel le membre est déclaré. En outre, il s'ensuit que le domaine d'accessibilité d'un membre n'est jamais plus inclusif que le domaine d'accessibilité du type dans lequel le membre est déclaré.

En termes intuitifs, lors de l'accès à un type ou à un membre `M` , les étapes suivantes sont évaluées pour s'assurer que l'accès est autorisé :

- Tout d'abord, si `M` est déclaré dans un type (par opposition à une unité de compilation ou un espace de noms), une erreur de compilation se produit si ce type n'est pas accessible.
- Ensuite, si `M` est `public` , l'accès est autorisé.
- Sinon, si `M` est `protected internal` , l'accès est autorisé s'il se produit dans le programme dans lequel `M` est déclaré, ou s'il se produit dans une classe dérivée de la classe dans laquelle `M` est déclaré et s'effectue via le type de classe dérivée (accès protégé pour les membres d'instance).
- Sinon, si `M` est `protected` , l'accès est autorisé s'il se produit dans la classe dans laquelle `M` est déclaré, ou s'il se produit dans une classe dérivée de la classe dans laquelle `M` est déclaré et s'effectue par le biais du type de classe dérivée (accès protégé pour les membres d'instance).
- Sinon, si `M` est `internal` , l'accès est autorisé s'il se produit dans le programme dans lequel `M` est déclaré.
- Sinon, si `M` est `private` , l'accès est autorisé s'il se produit dans le type dans lequel `M` est déclaré.
- Dans le cas contraire, le type ou le membre n'est pas accessible et une erreur de compilation se produit.

Dans l'exemple

C#

```
public class A
{
    public static int X;
    internal static int Y;
    private static int Z;
}

internal class B
{
    public static int X;
    internal static int Y;
    private static int Z;

    public class C
    {
        public static int X;
        internal static int Y;
        private static int Z;
    }

    private class D
    {
        public static int X;
        internal static int Y;
        private static int Z;
    }
}
```

les classes et les membres ont les domaines d'accessibilité suivants :

- Le domaine d'accessibilité de `A` et `A.X` est illimité.
- Le domaine d'accessibilité de `A.Y` ,, `B` ,, `B.X` `B.Y` `B.C` , `B.C.X` et `B.C.Y` est le texte de programme du programme conteneur.
- Le domaine d'accessibilité de `A.Z` est le texte de programme de `A` .
- Le domaine d'accessibilité de `B.Z` et `B.D` est le texte de programme de `B` , y compris le texte de programme de `B.C` et `B.D` .
- Le domaine d'accessibilité de `B.C.Z` est le texte de programme de `B.C` .
- Le domaine d'accessibilité de `B.D.X` et `B.D.Y` est le texte de programme de `B` , y compris le texte de programme de `B.C` et `B.D` .
- Le domaine d'accessibilité de `B.D.Z` est le texte de programme de `B.D` .

Comme l'illustre l'exemple, le domaine d'accessibilité d'un membre n'est jamais plus grand que celui d'un type conteneur. Par exemple, même si tous les `X` membres ont une accessibilité déclarée publique, tout sauf les `A.X` domaines d'accessibilité qui sont limités par un type conteneur.

Comme décrit dans membres, tous les membres d'une classe de base, à l'exception des constructeurs d'instance, des destructeurs et des constructeurs statiques, sont hérités par les types dérivés. Cela comprend même les membres privés d'une classe de base. Toutefois, le domaine d'accessibilité d'un membre privé comprend uniquement le texte de programme du type dans lequel le membre est déclaré. Dans l'exemple

C#

```
class A
{
    int x;

    static void F(B b) {
        b.x = 1;        // Ok
    }
}

class B: A
{
    static void F(B b) {
        b.x = 1;        // Error, x not accessible
    }
}
```

la `B` classe hérite du membre privé `x` de la `A` classe. Étant donné que le membre est privé, il est uniquement accessible dans le *class_body* de `A` . Ainsi, l'accès à `b.x` réussit dans la `A.F` méthode, mais échoue dans la `B.F` méthode.

Accès protégé pour les membres d'instance

Lorsqu'un `protected` membre d'instance est accessible en dehors du texte de programme de la classe dans laquelle il est déclaré, et quand un `protected internal` membre d'instance est accessible en dehors du texte de programme du programme dans lequel il est déclaré, l'accès doit avoir lieu dans une déclaration de classe qui dérive de la classe dans laquelle il est déclaré. En outre, l'accès doit être effectué par le biais d'une instance de ce type de classe dérivée ou d'un type de classe construit à partir de celui-ci. Cette restriction empêche une classe dérivée d'accéder aux membres protégés d'autres classes dérivées, même lorsque les membres sont hérités de la même classe de base.

`B`Il s'agit d'une classe de base qui déclare un membre `M` d'instance protégé et `D` qui est une classe qui dérive de `B` . Dans le *class_body* de `D` , l'accès à `M` peut prendre l'une des formes suivantes :

- *Type_name* ou *primary_expression* non qualifié du formulaire `M` .

- *Primary_expression* du formulaire `E.M` , à condition que le type `E` soit `T` ou une classe dérivée de `T` , où `T` est le type de classe `D` , ou un type de classe construit à partir de`D`
- *Primary_expression* du formulaire `base.M` .

En plus de ces formes d'accès, une classe dérivée peut accéder à un constructeur d'instance protégé d'une classe de base dans un *constructor_initializer* (initialiseurs de constructeur).

Dans l'exemple

```
C#

public class A
{
    protected int x;

    static void F(A a, B b) {
        a.x = 1;         // Ok
        b.x = 1;         // Ok
    }
}

public class B: A
{
    static void F(A a, B b) {
        a.x = 1;         // Error, must access through instance of B
        b.x = 1;         // Ok
    }
}
```

dans `A` , il est possible d'accéder `x` via des instances de `A` et `B` , car dans les deux cas, l'accès a lieu via une instance de `A` ou une classe dérivée de `A` . Toutefois, dans `B` , il n'est pas possible d'accéder à `x` par le biais d'une instance de `A` , car `A` ne dérive pas de `B` .

Dans l'exemple

```
C#

class C<T>
{
    protected T x;
}

class D<T>: C<T>
{
    static void F() {
```

```
        D<T> dt = new D<T>();
        D<int> di = new D<int>();
        D<string> ds = new D<string>();
        dt.x = default(T);
        di.x = 123;
        ds.x = "test";
    }
}
```

les trois assignations à `x` sont autorisées parce qu'elles ont toutes lieu via des instances de types de classe construits à partir du type générique.

Contraintes d'accessibilité

Plusieurs constructions en langage C# requièrent qu'un type soit ***au moins aussi accessible qu***' un membre ou un autre type. Un type `T` est considéré comme étant au moins aussi accessible qu'un membre ou un type `M` si le domaine d'accessibilité de `T` est un sur-ensemble du domaine d'accessibilité de `M` . En d'autres termes, `T` est au moins aussi accessible que `M` si `T` est accessible dans tous les contextes dans lesquels `M` est accessible.

Les contraintes d'accessibilité suivantes existent :

- La classe de base directe d'un type de classe doit être au moins aussi accessible que le type de classe lui-même.
- Les interfaces de base explicites d'un type d'interface doivent être au moins aussi accessibles que le type d'interface lui-même.
- Le type de retour et les types de paramètres d'un type délégué doivent être au moins aussi accessibles que le type délégué lui-même.
- Le type d'une constante doit être au moins aussi accessible que la constante elle-même.
- Le type d'un champ doit être au moins aussi accessible que le champ lui-même.
- Le type de retour et les types de paramètres d'une méthode doivent être au moins aussi accessibles que la méthode elle-même.
- Le type d'une propriété doit être au moins aussi accessible que la propriété elle-même.
- Le type d'un événement doit être au moins aussi accessible que l'événement lui-même.
- Le type et les types de paramètres d'un indexeur doivent être au moins aussi accessibles que l'indexeur lui-même.
- Le type de retour et les types de paramètres d'un opérateur doivent être au moins aussi accessibles que l'opérateur lui-même.

- Les types de paramètres d'un constructeur d'instance doivent être au moins aussi accessibles que le constructeur d'instance lui-même.

Dans l'exemple

```
C#

class A {...}

public class B: A {...}
```

la `B` classe génère une erreur au moment de la compilation, car `A` n'est pas au moins aussi accessible que `B` .

De même, dans l'exemple

```
C#

class A {...}

public class B
{
    A F() {...}

    internal A G() {...}

    public A H() {...}
}
```

la `H` méthode dans `B` génère une erreur de compilation, car le type de retour `A` n'est pas au moins aussi accessible que la méthode.

Signatures et surcharge

Les méthodes, constructeurs d'instance, indexeurs et opérateurs sont caractérisés par leurs ***signatures***:

- La signature d'une méthode se compose du nom de la méthode, du nombre de paramètres de type et du type et du type (valeur, référence ou sortie) de chacun de ses paramètres formels, pris en compte dans l'ordre de gauche à droite. Pour ces raisons, tout paramètre de type de la méthode qui se produit dans le type d'un paramètre formel est identifié par son nom, mais par sa position ordinale dans la liste d'arguments de type de la méthode. La signature d'une méthode n'inclut pas spécifiquement le type de retour, le `params` modificateur qui peut être spécifié

pour le paramètre le plus à droite, ni les contraintes de paramètre de type facultatives.
- La signature d'un constructeur d'instance se compose du type et du type (valeur, référence ou sortie) de chacun de ses paramètres formels, pris en compte dans l'ordre de gauche à droite. La signature d'un constructeur d'instance n'inclut pas spécifiquement le `params` modificateur qui peut être spécifié pour le paramètre le plus à droite.
- La signature d'un indexeur se compose du type de chacun de ses paramètres formels, pris en compte dans l'ordre de gauche à droite. La signature d'un indexeur n'inclut pas spécifiquement le type d'élément et n'inclut pas non plus le `params` modificateur qui peut être spécifié pour le paramètre le plus à droite.
- La signature d'un opérateur se compose du nom de l'opérateur et du type de chacun de ses paramètres formels, pris en compte dans l'ordre de gauche à droite. La signature d'un opérateur n'inclut pas spécifiquement le type de résultat.

Les signatures sont le mécanisme d'activation pour la ***surcharge*** des membres dans des classes, des structures et des interfaces :

- La surcharge de méthodes permet à une classe, un struct ou une interface de déclarer plusieurs méthodes portant le même nom, à condition que leurs signatures soient uniques dans cette classe, structure ou interface.
- La surcharge des constructeurs d'instance permet à une classe ou un struct de déclarer plusieurs constructeurs d'instance, à condition que leurs signatures soient uniques au sein de cette classe ou de ce struct.
- La surcharge d'indexeurs permet à une classe, un struct ou une interface de déclarer plusieurs indexeurs, à condition que leurs signatures soient uniques au sein de cette classe, structure ou interface.
- La surcharge des opérateurs permet à une classe ou un struct de déclarer plusieurs opérateurs portant le même nom, à condition que leurs signatures soient uniques au sein de cette classe ou de ce struct.

Bien que les `out` `ref` modificateurs de paramètres soient considérés comme faisant partie d'une signature, les membres déclarés dans un même type ne peuvent pas être différents dans la signature uniquement par `ref` et `out` . Une erreur de compilation se produit si deux membres sont déclarés dans le même type avec des signatures qui seraient les mêmes si tous les paramètres dans les deux méthodes avec des `out` modificateurs ont été modifiés en `ref` modificateurs. À d'autres fins de correspondance de signature (par exemple, le masquage ou la substitution) `ref` et `out` sont considérés comme faisant partie de la signature et ne correspondent pas l'un à l'autre. (Cette restriction consiste à permettre aux programmes C# d'être facilement traduits pour

s'exécuter sur le Common Language Infrastructure (interface de commande), ce qui ne permet pas de définir des méthodes qui diffèrent uniquement par `ref` et `out` .)

Pour les besoins des signatures, les types `object` et `dynamic` sont considérés comme identiques. Les membres déclarés dans un même type ne peuvent donc pas être différents dans la signature uniquement par `object` et `dynamic` .

L'exemple suivant montre un ensemble de déclarations de méthode surchargées, ainsi que leurs signatures.

C#

```
interface ITest
{
    void F();                       // F()

    void F(int x);                  // F(int)

    void F(ref int x);              // F(ref int)

    void F(out int x);              // F(out int)      error

    void F(int x, int y);           // F(int, int)

    int F(string s);                // F(string)

    int F(int x);                   // F(int)          error

    void F(string[] a);             // F(string[])

    void F(params string[] a);      // F(string[])     error
}
```

Notez que tous `ref` les `out` modificateurs de paramètres et (paramètres de méthode) font partie d'une signature. Ainsi, `F(int)` et `F(ref int)` sont des signatures uniques. Toutefois, `F(ref int)` et `F(out int)` ne peuvent pas être déclarés dans la même interface, car leurs signatures diffèrent uniquement par `ref` et `out` . Notez également que le type de retour et le `params` modificateur ne font pas partie d'une signature. il n'est donc pas possible de se surcharger uniquement en fonction du type de retour ou de l'inclusion ou de l'exclusion du `params` modificateur. Ainsi, les déclarations des méthodes `F(int)` et `F(params string[])` identifiées ci-dessus entraînent une erreur de compilation.

Étendues

La valeur *__scope__ _ d'un nom est la région du texte de programme dans laquelle il est possible de faire référence à l'entité déclarée par le nom sans qualification du nom. Les portées peuvent être *imbriquées* et une portée interne peut redéclarer la signification d'un nom à partir d'une étendue externe (Toutefois, cela ne supprime pas la restriction imposée par les déclarations dans un bloc imbriqué, il n'est pas possible de déclarer une variable locale portant le même nom qu'une variable locale dans un bloc englobant). Le nom de l'étendue externe est ensuite dit _ *Hidden** dans la région du texte de programme couvert par l'étendue interne, et l'accès au nom externe n'est possible qu'en qualifiant le nom.

- La portée d'un membre d'espace de noms déclaré par un *namespace_member_declaration* (membres d'espace de noms) sans *namespace_declaration* englobante correspond au texte de programme entier.
- La portée d'un membre d'espace de noms déclaré par un *namespace_member_declaration* dans un *namespace_declaration* dont le nom qualifié complet est `N` est le *namespace_body* de chaque *namespace_declaration* dont le nom qualifié complet est `N` ou commence par `N` , suivi d'un point.
- L'étendue de nom définie par un *extern_alias_directive* s'étend sur le *using_directive* s, *global_attributes* et *namespace_member_declaration* s de son corps d'unité de compilation ou d'espace de noms contenant immédiatement. Un *extern_alias_directive* ne contribue pas à la création de nouveaux membres dans l'espace de déclaration sous-jacent. En d'autres termes, un *extern_alias_directive* n'est pas transitif, mais affecte uniquement l'unité de compilation ou le corps de l'espace de noms dans lequel il se produit.
- L'étendue d'un nom défini ou importé par un *using_directive* (à l'aide de directives) s'étend sur le *namespace_member_declaration* s du *compilation_unit* ou *namespace_body* dans lequel la *using_directive* se produit. Une *using_directive* peut rendre zéro ou plusieurs noms d'espaces de noms, de types ou de membres disponibles dans une *compilation_unit* ou *namespace_body* particulière, mais ne contribue pas à de nouveaux membres dans l'espace de déclaration sous-jacent. En d'autres termes, un *using_directive* n'est pas transitif, mais affecte uniquement le *compilation_unit* ou *namespace_body* dans lequel il se produit.
- La portée d'un paramètre de type déclaré par un *type_parameter_list* sur un *class_declaration* (déclarations de classe) est la *class_base*, *type_parameter_constraints_clause* s et *class_body* de cette *class_declaration*.
- La portée d'un paramètre de type déclaré par un *type_parameter_list* sur un *struct_declaration* (déclarations de struct) est la *struct_interfaces*, *type_parameter_constraints_clause* s et *struct_body* de cette *struct_declaration*.
- La portée d'un paramètre de type déclaré par un *type_parameter_list* sur une *interface_declaration* (déclarations d'interface) est le *interface_base*,

type_parameter_constraints_clause s et *interface_body* de cette *interface_declaration*.

- La portée d'un paramètre de type déclaré par un *type_parameter_list* sur un *delegate_declaration* (déclarations de délégué) est le *return_type*, le *formal_parameter_list* et *type_parameter_constraints_clause* s de ce *delegate_declaration*.
- La portée d'un membre déclaré par un *class_member_declaration* (corps de classe) est le *class_body* dans lequel la déclaration se produit. En outre, l'étendue d'un membre de classe s'étend au *class_body* de ces classes dérivées qui sont incluses dans le domaine d'accessibilité (domaines d'accessibilité) du membre.
- La portée d'un membre déclaré par un *struct_member_declaration* (membres struct) est le *struct_body* dans lequel la déclaration se produit.
- La portée d'un membre déclaré par un *enum_member_declaration* (membres enum) est le *enum_body* dans lequel la déclaration se produit.
- La portée d'un paramètre déclaré dans une *method_declaration* (méthodes) est le *method_body* de ce *method_declaration*.
- La portée d'un paramètre déclaré dans un *indexer_declaration* (indexeurs) est le *accessor_declarations* de ce *indexer_declaration*.
- La portée d'un paramètre déclaré dans un *operator_declaration* (opérateurs) est le *bloc* de ce *operator_declaration*.
- La portée d'un paramètre déclaré dans une *constructor_declaration* (constructeurs d'instance) est le *constructor_initializer* et le *bloc* de ce *constructor_declaration*.
- La portée d'un paramètre déclaré dans une *lambda_expression* (expressions de fonction anonymes) est la *anonymous_function_body* de ce *lambda_expression*
- La portée d'un paramètre déclaré dans une *anonymous_method_expression* (expressions de fonction anonymes) est le *bloc* de ce *anonymous_method_expression*.
- La portée d'une étiquette déclarée dans une *labeled_statement* (instructions étiquetées) est le *bloc* dans lequel la déclaration se produit.
- La portée d'une variable locale déclarée dans une *local_variable_declaration* (déclarations de variables locales) est le bloc dans lequel la déclaration se produit.
- La portée d'une variable locale déclarée dans une *switch_block* d'une `switch` instruction (instruction switch) est la *switch_block*.
- La portée d'une variable locale déclarée dans *une for_initializer* d' `for` une instruction (l'instruction for) est le *for_initializer*, le *for_condition*, l' *for_iterator* et l' *instruction* contenue de l' `for` instruction.
- La portée d'une constante locale déclarée dans une *local_constant_declaration* (déclarations de constantes locales) est le bloc dans lequel la déclaration se produit. Il s'agit d'une erreur au moment de la compilation pour faire référence à

une constante locale dans une position textuelle qui précède son *constant_declarator*.

- La portée d'une variable déclarée dans le cadre d'un *foreach_statement*, *using_statement lock_statement* ou *query_expression* est déterminée par le développement de la construction donnée.

Dans l'étendue d'un espace de noms, d'une classe, d'une structure ou d'un membre d'énumération, il est possible de faire référence au membre dans une position textuelle qui précède la déclaration du membre. Par exemple

C#

```
class A
{
    void F() {
        i = 1;
    }

    int i = 0;
}
```

Ici, il est valide pour `F` que fasse référence à `i` avant d'être déclaré.

Dans l'étendue d'une variable locale, il s'agit d'une erreur de compilation qui fait référence à la variable locale dans une position textuelle qui précède la *local_variable_declarator* de la variable locale. Par exemple

C#

```
class A
{
    int i = 0;

    void F() {
        i = 1;                  // Error, use precedes declaration
        int i;
        i = 2;
    }

    void G() {
        int j = (j = 1);        // Valid
    }

    void H() {
        int a = 1, b = ++a;     // Valid
    }
}
```

Dans la `F` méthode ci-dessus, la première assignation à `i` ne fait pas spécifiquement référence au champ déclaré dans l'étendue externe. Au lieu de cela, il fait référence à la variable locale et génère une erreur de compilation, car il précède la déclaration de la variable dans le texte. Dans la `G` méthode, l'utilisation de `j` dans l'initialiseur pour la déclaration de `j` est valide, car l'utilisation ne précède pas la *local_variable_declarator*. Dans la `H` méthode, une *local_variable_declarator* suivante fait correctement référence à une variable locale déclarée dans une *local_variable_declarator* antérieure au sein du même *local_variable_declaration*.

Les règles de portée pour les variables locales sont conçues pour garantir que la signification d'un nom utilisé dans un contexte d'expression est toujours la même dans un bloc. Si la portée d'une variable locale ne doit s'étendre qu'à partir de sa déclaration à la fin du bloc, alors, dans l'exemple ci-dessus, la première assignation est assignée à la variable d'instance et la deuxième assignation est assignée à la variable locale, ce qui peut entraîner des erreurs de compilation si les instructions du bloc ont été réorganisées ultérieurement.

La signification d'un nom dans un bloc peut varier selon le contexte dans lequel le nom est utilisé. Dans l'exemple

C#

```
using System;

class A {}

class Test
{
    static void Main() {
        string A = "hello, world";
        string s = A;                           // expression context

        Type t = typeof(A);                     // type context

        Console.WriteLine(s);                   // writes "hello, world"
        Console.WriteLine(t);                   // writes "A"
    }
}
```

le nom `A` est utilisé dans un contexte d'expression pour faire référence à la variable locale `A` et dans un contexte de type pour faire référence à la classe `A` .

Masquage de nom

La portée d'une entité comprend généralement plus de texte de programme que l'espace de déclaration de l'entité. En particulier, l'étendue d'une entité peut inclure des déclarations qui introduisent de nouveaux espaces de déclaration contenant des entités du même nom. De telles déclarations provoquent l'origine de l'entité *__Hidden__ _. À l'inverse, une entité est dite «*_visible_** » lorsqu'elle n'est pas masquée.

Le masquage de nom se produit lorsque les étendues se chevauchent lors de l'imbrication et lorsque les étendues chevauchent l'héritage. Les caractéristiques des deux types de masquage sont décrites dans les sections suivantes.

Masquer par imbrication

Le masquage de nom via l'imbrication peut se produire en raison de l'imbrication d'espaces de noms ou de types dans les espaces de noms, en raison de l'imbrication de types dans des classes ou des structs, et de la suite de déclarations de variables locales et de paramètres.

Dans l'exemple

C#

```
class A
{
    int i = 0;

    void F() {
        int i = 1;
    }

    void G() {
        i = 1;
    }
}
```

dans la `F` méthode, la variable d'instance `i` est masquée par la variable locale `i` , mais dans la `G` méthode, `i` fait toujours référence à la variable d'instance.

Lorsqu'un nom dans une portée interne masque un nom dans une portée externe, il masque toutes les occurrences surchargées de ce nom. Dans l'exemple

C#

```
class Outer
{
    static void F(int i) {}
```

```
    static void F(string s) {}

    class Inner
    {
        void G() {
            F(1);              // Invokes Outer.Inner.F
            F("Hello");        // Error
        }

        static void F(long l) {}
    }
}
```

l'appel `F(1)` appelle le `F` déclaré dans `Inner` , car toutes les occurrences externes de `F` sont masquées par la déclaration interne. Pour la même raison, l'appel `F("Hello")` génère une erreur au moment de la compilation.

Masquer par héritage

Le masquage de nom par héritage se produit lorsque des classes ou des structs déclarent des noms qui ont été hérités de classes de base. Ce type de masquage de nom prend l'une des formes suivantes :

- Une constante, un champ, une propriété, un événement ou un type introduit dans une classe ou un struct masque tous les membres de la classe de base portant le même nom.
- Une méthode introduite dans une classe ou un struct masque tous les membres de classe de base non-méthode portant le même nom, ainsi que toutes les méthodes de la classe de base ayant la même signature (nom de méthode et nombre de paramètres, modificateurs et types).
- Un indexeur introduit dans une classe ou un struct masque tous les indexeurs de la classe de base avec la même signature (nombre de paramètres et types).

Les règles gouvernant les déclarations d'opérateur (opérateurs) rendent impossible une classe dérivée de déclarer un opérateur avec la même signature qu'un opérateur dans une classe de base. Ainsi, les opérateurs ne sont jamais masqués.

Contrairement au masquage d'un nom dans une portée externe, le masquage d'un nom accessible à partir d'une portée héritée entraîne le signalement d'un avertissement. Dans l'exemple

C#

```
class Base
{
```

```
    public void F() {}
}

class Derived: Base
{
    public void F() {}        // Warning, hiding an inherited name
}
```

la déclaration de `F` dans `Derived` provoque le signalement d'un avertissement. Le masquage d'un nom hérité n'est pas une erreur, car cela empêche l'évolution distincte des classes de base. Par exemple, la situation ci-dessus peut être due au fait qu'une version ultérieure de a `Base` introduit une `F` méthode qui n'était pas présente dans une version antérieure de la classe. Si la situation précédente était une erreur, toute modification apportée à une classe de base dans une bibliothèque de classes avec version distincte pourrait entraîner la non-validité des classes dérivées.

L'avertissement provoqué par le masquage d'un nom hérité peut être éliminé grâce à l'utilisation du `new` modificateur :

C#

```
class Base
{
    public void F() {}
}

class Derived: Base
{
    new public void F() {}
}
```

Le `new` modificateur indique que le `F` dans `Derived` est « New », et qu'il est effectivement destiné à masquer le membre hérité.

Une déclaration d'un nouveau membre masque un membre hérité uniquement dans la portée du nouveau membre.

C#

```
class Base
{
    public static void F() {}
}

class Derived: Base
{
    new private static void F() {}    // Hides Base.F in Derived only
}
```

```
class MoreDerived: Derived
{
    static void G() { F(); }          // Invokes Base.F
}
```

Dans l'exemple ci-dessus, la déclaration de `F` dans `Derived` masque le `F` qui a été hérité de `Base` , mais étant donné que le nouveau `F` dans `Derived` a un accès privé, sa portée ne s'étend pas à `MoreDerived` . Ainsi, l'appel `F()` de `MoreDerived.G` est valide et appellera `Base.F` .

Noms des espaces de noms et des types

Plusieurs contextes dans un programme C# requièrent la spécification d'une *namespace_name* ou d'un *type_name* .

antlr

```
namespace_name
    : namespace_or_type_name
    ;

type_name
    : namespace_or_type_name
    ;

namespace_or_type_name
    : identifier type_argument_list?
    | namespace_or_type_name '.' identifier type_argument_list?
    | qualified_alias_member
    ;
```

Un *namespace_name* est un *namespace_or_type_name* qui fait référence à un espace de noms. Après la résolution comme décrit ci-dessous, la *namespace_or_type_name* d'un *namespace_name* doit faire référence à un espace de noms ou une erreur de compilation se produit. Aucun argument de type (arguments de type) ne peut être présent dans une *namespace_name* (seuls les types peuvent avoir des arguments de type).

Un *type_name* est un *namespace_or_type_name* qui fait référence à un type. Après la résolution comme décrit ci-dessous, la *namespace_or_type_name* d'un *type_name* doit faire référence à un type, ou une erreur de compilation se produit.

Si le *namespace_or_type_name* est un membre d'alias qualifié, son sens est décrit dans qualificateurs d'alias d'espace de noms. Dans le cas contraire, un

namespace_or_type_name a l'une des quatre formes suivantes :

- `I`
- `I<A1, ..., Ak>`
- `N.I`
- `N.I<A1, ..., Ak>`

où `I` est un identificateur unique, `N` est un *namespace_or_type_name* et `<A1, ..., Ak>` est un *type_argument_list* facultatif. Si aucun *type_argument_list* n'est spécifié, pensez `k` à être égal à zéro.

La signification d'un *namespace_or_type_name* est déterminée comme suit :

- Si le *namespace_or_type_name* se présente sous la forme `I` ou sous la forme `I<A1, ..., Ak>` :
  - Si `K` est égal à zéro et que le *namespace_or_type_name* apparaît dans une déclaration de méthode générique (méthodes) et si cette déclaration comprend un paramètre de type (paramètres de type) avec `I` le nom, le *namespace_or_type_name* fait référence à ce paramètre de type.
  - Sinon, si le *namespace_or_type_name* apparaît dans une déclaration de type, pour chaque type d'instance `T` (type d'instance), en commençant par le type d'instance de cette déclaration de type et en poursuivant avec le type d'instance de chaque classe englobante ou déclaration de struct (le cas échéant) :
    - Si `K` est égal à zéro et que la déclaration de `T` comprend un paramètre de type portant `I` le nom, le *namespace_or_type_name* fait référence à ce paramètre de type.
    - Sinon, si le *namespace_or_type_name* apparaît dans le corps de la déclaration de type, et ou si l' `T` un de ses types de base contient un type accessible imbriqué ayant des `I` paramètres Name et `K` type, alors le *namespace_or_type_name* fait référence à ce type construit avec les arguments de type donnés. S'il existe plusieurs types de ce type, le type déclaré dans le type plus dérivé est sélectionné. Notez que les membres qui ne sont pas de type (constantes, champs, méthodes, propriétés, indexeurs, opérateurs, constructeurs d'instance, destructeurs et constructeurs statiques) et les membres de type avec un nombre différent de paramètres de type sont ignorés lors de la détermination de la signification de la *namespace_or_type_name*.
  - Si les étapes précédentes n'ont pas réussi, pour chaque espace de noms `N` , en commençant par l'espace de noms dans lequel le *namespace_or_type_name* se produit, en continuant avec chaque espace de noms englobant (le cas échéant)

et en terminant par l'espace de noms global, les étapes suivantes sont évaluées jusqu'à ce qu'une entité soit localisée :

- Si `K` est égal à zéro et `I` est le nom d'un espace de noms dans `N`, alors :
  - Si l'emplacement où le *namespace_or_type_name* se produit est entouré d'une déclaration d'espace de noms pour `N` et que la déclaration d'espace de noms contient une *extern_alias_directive* ou *using_alias_directive* qui associe le nom `I` à un espace de noms ou à un type, la *namespace_or_type_name* est ambiguë et une erreur de compilation se produit.
  - Dans le cas contraire, le *namespace_or_type_name* fait référence à l'espace de noms nommé `I` dans `N`.
- Sinon, si `N` contient un type accessible avec `I` des paramètres Name et `K` type, alors :
  - Si `K` est égal à zéro et que l'emplacement où le *namespace_or_type_name* se produit est entouré d'une déclaration d'espace de noms pour `N` et que la déclaration d'espace de noms contient une *extern_alias_directive* ou *using_alias_directive* qui associe le nom `I` à un espace de noms ou à un type, le *namespace_or_type_name* est ambigu et une erreur de compilation se produit.
  - Sinon, le *namespace_or_type_name* fait référence au type construit avec les arguments de type donnés.
- Sinon, si l'emplacement où le *namespace_or_type_name* se produit est entouré d'une déclaration d'espace de noms pour `N` :
  - Si `K` est égal à zéro et que la déclaration d'espace de noms contient une *extern_alias_directive* ou *using_alias_directive* qui associe le nom `I` à un espace de noms ou un type importé, le *namespace_or_type_name* fait référence à cet espace de noms ou type.
  - Sinon, si les espaces de noms et les déclarations de type importés par les *using_namespace_directive* s et *using_alias_directive* s de la déclaration d'espace de noms contiennent exactement un type accessible `I` avec des paramètres de type et de nom `K`, le *namespace_or_type_name* fait référence à ce type construit avec les arguments de type donnés.
  - Sinon, si les espaces de noms et les déclarations de type importés par les *using_namespace_directive* s et *using_alias_directive* s de la déclaration d'espace de noms contiennent plusieurs types accessibles ayant des `I` paramètres name et `K` type, la *namespace_or_type_name* est ambiguë et une erreur se produit.
- Dans le cas contraire, la *namespace_or_type_name* n'est pas définie et une erreur de compilation se produit.

- Dans le cas contraire, le *namespace_or_type_name* se présente sous la forme `N.I` ou sous la forme `N.I<A1, ..., Ak>` . `N` est d'abord résolu en tant que *namespace_or_type_name*. Si la résolution de `N` échoue, une erreur de compilation se produit. Sinon, `N.I` ou `N.I<A1, ..., Ak>` est résolu comme suit :
  - Si `K` est égal à zéro et `N` fait référence à un espace de noms et `N` qu'il contient un espace de noms imbriqué portant `I` le nom, le *namespace_or_type_name* fait référence à cet espace de noms imbriqué.
  - Sinon, si `N` fait référence à un espace de noms et `N` contient un type accessible avec des `I` paramètres Name et `K` type, alors le *namespace_or_type_name* fait référence à ce type construit avec les arguments de type donnés.
  - Sinon, si `N` fait référence à un type de classe ou de struct (éventuellement construit) et ou si l'une `N` de ses classes de base contient un type accessible imbriqué ayant des `I` paramètres de type et de nom `K` , le *namespace_or_type_name* fait référence à ce type construit avec les arguments de type donnés. S'il existe plusieurs types de ce type, le type déclaré dans le type plus dérivé est sélectionné. Notez que si la signification de `N.I` est déterminée dans le cadre de la résolution de la spécification de classe de base de `N` , la classe de base directe de `N` est considérée comme Object (classes de base).
  - Sinon, `N.I` est un *namespace_or_type_name* non valide et une erreur de compilation se produit.

Un *namespace_or_type_name* est autorisé à référencer une classe statique (classes statiques) uniquement si

- Le *namespace_or_type_name* est `T` dans un *namespace_or_type_name* de la forme `T.I` , ou
- Le *namespace_or_type_name* est le `T` dans un *typeof_expression* (liste d'arguments1) du formulaire `typeof(T)` .

Noms qualifiés complets

Chaque espace de noms et type a un ***nom qualifié complet***, qui identifie de façon unique l'espace de noms ou le type parmi tous les autres. Le nom qualifié complet d'un espace de noms ou d'un type `N` est déterminé comme suit :

- Si `N` est un membre de l'espace de noms global, son nom complet est `N` .
- Sinon, son nom qualifié complet est `S.N` , où `S` est le nom qualifié complet de l'espace de noms ou du type dans lequel `N` est déclaré.

En d'autres termes, le nom complet de `N` est le chemin d'accès hiérarchique complet des identificateurs qui mènent à `N` , à partir de l'espace de noms global. Étant donné que chaque membre d'un espace de noms ou d'un type doit avoir un nom unique, il suit que le nom qualifié complet d'un espace de noms ou d'un type est toujours unique.

L'exemple ci-dessous montre plusieurs déclarations d'espace de noms et de type, ainsi que leurs noms complets associés.

C#

```
class A {}                  // A

namespace X                 // X
{
    class B                 // X.B
    {
        class C {}          // X.B.C
    }

    namespace Y             // X.Y
    {
        class D {}          // X.Y.D
    }
}

namespace X.Y               // X.Y
{
    class E {}              // X.Y.E
}
```

Gestion automatique de la mémoire

C# utilise la gestion automatique de la mémoire, qui évite aux développeurs d'allouer et de libérer manuellement la mémoire occupée par les objets. Les stratégies de gestion automatique de la mémoire sont implémentées par un ***garbage collector***. Le cycle de vie de la gestion de la mémoire d'un objet est le suivant :

1. Lorsque l'objet est créé, la mémoire est allouée, le constructeur est exécuté et l'objet est considéré comme dynamique.
2. Si l'objet, ou une partie de celui-ci, n'est pas accessible par une continuation d'exécution possible, autre que l'exécution de destructeurs, l'objet est considéré comme n'étant plus utilisé et il est éligible pour la destruction. Le compilateur C# et le garbage collector peuvent choisir d'analyser le code pour déterminer les références à un objet qui peuvent être utilisées ultérieurement. Par exemple, si une variable locale qui se trouve dans l'étendue est la seule référence existante à un

objet, mais que cette variable locale n'est jamais référencée dans une éventuelle continuation d'exécution à partir du point d'exécution actuel de la procédure, le garbage collector peut traiter l'objet comme n'étant plus en cours d'utilisation.

3. Une fois que l'objet est éligible pour la destruction, à une date non spécifiée plus tard, ledestructeur (le cas échéant) del'objet est exécuté. Dans des circonstances normales, le destructeur de l'objet est exécuté une seule fois, bien que les API spécifiques à l'implémentation puissent autoriser le remplacement de ce comportement.
4. Une fois que le destructeur pour un objet est exécuté, si cet objet, ou une partie de celui-ci, n'est pas accessible par toute continuation d'exécution possible, y compris l'exécution de destructeurs, l'objet est considéré comme étant inaccessible et l'objet devient éligible pour la collecte.
5. Enfin, une fois que l'objet est éligible pour la collecte, le garbage collector libère la mémoire associée à cet objet.

Le garbage collector conserve des informations sur l'utilisation des objets et utilise ces informations pour prendre des décisions de gestion de la mémoire, telles que l'emplacement de la mémoire pour localiser un objet nouvellement créé, quand déplacer un objet, et quand un objet n'est plus utilisé ou n'est pas accessible.

Comme les autres langages qui supposent l'existence d'un garbage collector, C# est conçu de sorte que le garbage collector puisse implémenter une large gamme de stratégies de gestion de la mémoire. Par exemple, C# ne nécessite pas l'exécution des destructeurs ou la collecte des objets dès qu'ils sont éligibles, ou l'exécution des destructeurs dans un ordre particulier ou sur un thread particulier.

Le comportement du garbage collector peut être contrôlé, à un certain degré, via des méthodes statiques sur la classe `System.GC` . Cette classe peut être utilisée pour demander qu'une collection se produise, que les destructeurs soient exécutés (ou non), et ainsi de suite.

Dans la mesure où le garbage collector est autorisé à atteindre une grande latitude en déterminant quand collecter des objets et exécuter des destructeurs, une implémentation conforme peut produire une sortie qui diffère de celle indiquée par le code suivant. Le programme

C#

```
using System;

class A
{
    ~A() {
        Console.WriteLine("Destruct instance of A");
```

```
    }
}

class B
{
    object Ref;

    public B(object o) {
        Ref = o;
    }

    ~B() {
        Console.WriteLine("Destruct instance of B");
    }
}

class Test
{
    static void Main() {
        B b = new B(new A());
        b = null;
        GC.Collect();
        GC.WaitForPendingFinalizers();
    }
}
```

crée une instance de la classe `A` et une instance de la classe `B` . Ces objets deviennent éligibles pour garbage collection lorsque la variable `b` est assignée à la valeur, dans la `null` mesure où, à ce moment-là, il est impossible pour un code écrit par l'utilisateur d'y accéder. La sortie peut être

Console

```
Destruct instance of A
Destruct instance of B
```

or

Console

```
Destruct instance of B
Destruct instance of A
```

étant donné que le langage n'impose aucune contrainte sur l'ordre dans lequel les objets sont récupérés par le garbage collector.

Dans les cas subtils, la distinction entre « éligible à la destruction » et « éligible pour la collecte » peut être importante. Par exemple,

C#

```
using System;

class A
{
    ~A() {
        Console.WriteLine("Destruct instance of A");
    }

    public void F() {
        Console.WriteLine("A.F");
        Test.RefA = this;
    }
}

class B
{
    public A Ref;

    ~B() {
        Console.WriteLine("Destruct instance of B");
        Ref.F();
    }
}

class Test
{
    public static A RefA;
    public static B RefB;

    static void Main() {
        RefB = new B();
        RefA = new A();
        RefB.Ref = RefA;
        RefB = null;
        RefA = null;

        // A and B now eligible for destruction
        GC.Collect();
        GC.WaitForPendingFinalizers();

        // B now eligible for collection, but A is not
        if (RefA != null)
            Console.WriteLine("RefA is not null");
    }
}
```

Dans le programme ci-dessus, si le garbage collector choisit d'exécuter le destructeur de `A` avant le destructeur de `B` , la sortie de ce programme peut être :

Console

```
Destruct instance of A
Destruct instance of B
A.F
RefA is not null
```

Notez que même si l'instance de `A` n'était pas en cours d'utilisation et que le `A` destructeur de a été exécuté, il est toujours possible pour les méthodes de `A` (dans ce cas, `F`) d'être appelées à partir d'un autre destructeur. Notez également que l'exécution d'un destructeur peut entraîner la réutilisation d'un objet à partir du programme principal. Dans ce cas, le destructeur de en cours d'exécution `B` a provoqué une instance de `A` qui n'était pas utilisée précédemment pour devenir accessible à partir de la référence active `Test.RefA` . Après l'appel à `WaitForPendingFinalizers` , l'instance de `B` est éligible pour la collection, mais l'instance de `A` n'est pas, en raison de la référence `Test.RefA` .

Pour éviter toute confusion et un comportement inattendu, il est généralement judicieux que les destructeurs effectuent uniquement un nettoyage sur les données stockées dans les champs de leur propre objet, et non sur les objets référencés ou les champs statiques.

Une alternative à l'utilisation des destructeurs consiste à permettre à une classe d'implémenter l' `System.IDisposable` interface. Cela permet au client de l'objet de déterminer à quel moment libérer les ressources de l'objet, en utilisant généralement l'accès à l'objet en tant que ressource dans une `using` instruction (l'instruction using).

Ordre d'exécution

L'exécution d'un programme C# continue de telle sorte que les effets secondaires de chaque thread d'exécution soient conservés au niveau des points d'exécution critiques. Un ***effet secondaire*** est défini comme une lecture ou une écriture d'un champ volatile, une écriture dans une variable non volatile, une écriture dans une ressource externe et la levée d'une exception. Les points d'exécution critiques auxquels l'ordre de ces effets secondaires doivent être conservés sont des références à des champs volatiles (champs volatils), des `lock` instructions (instruction lock) et la création et l'arrêt des threads. L'environnement d'exécution est libre de modifier l'ordre d'exécution d'un programme C#, selon les contraintes suivantes :

- La dépendance des données est conservée dans un thread d'exécution. Autrement dit, la valeur de chaque variable est calculée comme si toutes les instructions du thread étaient exécutées dans l'ordre du programme d'origine.

- Les règles de classement d'initialisation sont conservées (initialisation de champ et initialiseurs de variable).
- L'ordre des effets secondaires est préservé en ce qui concerne les lectures et les écritures volatiles (champs volatiles). En outre, l'environnement d'exécution n'a pas besoin d'évaluer une partie d'une expression s'il peut déduire que la valeur de cette expression n'est pas utilisée et qu'aucun effet secondaire nécessaire n'est produit (y compris les effets causés par l'appel d'une méthode ou l'accès à un champ volatile). Lorsque l'exécution du programme est interrompue par un événement asynchrone (par exemple, une exception levée par un autre thread), il n'est pas garanti que les effets secondaires observables sont visibles dans l'ordre du programme d'origine.

Types

Article • 13/05/2021

Les types du langage C# sont divisés en deux catégories principales : ***types valeur*** _ *et* _*types référence*_.. Les types valeur et les types référence peuvent être des *types génériques*, qui prennent un ou plusieurs *paramètres* _ * type * *. Les paramètres de type peuvent désigner à la fois les types valeur et les types référence.

antlr

```
type
    : value_type
    | reference_type
    | type_parameter
    | type_unsafe
    ;
```

La catégorie finale de types, pointeurs, est uniquement disponible dans du code unsafe. Ce sujet est abordé plus en détail dans types de pointeurs.

Les types valeur diffèrent des types référence dans le fait que les variables des types valeur contiennent directement leurs données, alors que les variables des types référence stockent ***font référence** à leurs données, ce dernier étant appelé _ *Objects* *. Avec les types référence, deux variables peuvent faire référence au même objet et, par conséquent, les opérations sur une variable peuvent affecter l'objet référencé par l'autre variable. Avec les types valeur, les variables disposent chacune de leur propre copie des données, et il n'est pas possible que les opérations sur l'une d'elles affectent l'autre.

Le système de type C# est unifié de telle façon qu'une valeur de n'importe quel type peut être traitée en tant qu'objet. Chaque type dans C# dérive directement ou indirectement du type `object`, et `object` est la classe de base fondamentale de tous les types. Les valeurs des types référence sont considérées comme des objets simplement en affichant les valeurs en tant que type `object`. Les valeurs des types valeur sont traitées comme des objets en effectuant des opérations de boxing et d'unboxing (boxing et unboxing).

Types de valeur

Un type valeur est soit un type struct, soit un type énumération. C# fournit un ensemble de types struct prédéfinis appelés ***types simples***. Les types simples sont identifiés à l'aide de mots réservés.

antlr

```
value_type
    : struct_type
    | enum_type
    ;

struct_type
    : type_name
    | simple_type
    | nullable_type
    ;

simple_type
    : numeric_type
    | 'bool'
    ;

numeric_type
    : integral_type
    | floating_point_type
    | 'decimal'
    ;

integral_type
    : 'sbyte'
    | 'byte'
    | 'short'
    | 'ushort'
    | 'int'
    | 'uint'
    | 'long'
    | 'ulong'
    | 'char'
    ;

floating_point_type
    : 'float'
    | 'double'
    ;

nullable_type
    : non_nullable_value_type '?'
    ;

non_nullable_value_type
    : type
    ;

enum_type
    : type_name
    ;
```

Contrairement à une variable d'un type référence, une variable d'un type valeur peut contenir la valeur `null` uniquement si le type valeur est un type Nullable. Pour chaque type valeur qui n'autorise pas les valeurs NULL, il existe un type valeur Nullable correspondant indiquant le même jeu de valeurs plus la valeur `null` .

L'assignation à une variable d'un type valeur crée une copie de la valeur assignée. Cela diffère d'une assignation à une variable d'un type référence, qui copie la référence, mais pas l'objet identifié par la référence.

Type System. ValueType

Tous les types valeur héritent implicitement de la classe `System.ValueType` , qui, à son tour, hérite de la classe `object` . Il n'est pas possible pour un type de dériver d'un type valeur, et les types valeur sont donc implicitement sealed (classes sealed).

Notez qu' `System.ValueType` il ne s'agit pas d'un *Value_type*. Au lieu de cela, il s'agit d'un *class_type* duquel tous les *Value_type* sont dérivés automatiquement.

Constructeurs par défaut

Tous les types valeur déclarent implicitement un constructeur d'instance sans paramètre public appelé le ***constructeur par défaut** _. Le constructeur par défaut retourne une instance initialisée à zéro appelée _ *valeur par défaut** pour le type de valeur :

- Pour tous les *simple_type* s, la valeur par défaut est la valeur produite par un modèle binaire de zéros :
  - Pour `sbyte` , `byte` ,,,,, `short` `ushort` `int` `uint` `long` , et `ulong` , la valeur par défaut est `0` .
  - Pour `char` , la valeur par défaut est `'\x0000'` .
  - Pour `float` , la valeur par défaut est `0.0f` .
  - Pour `double` , la valeur par défaut est `0.0d` .
  - Pour `decimal` , la valeur par défaut est `0.0m` .
  - Pour `bool` , la valeur par défaut est `false` .
- Pour un *enum_type* `E` , la valeur par défaut est `0` , convertie en type `E` .
- Pour un *struct_type*, la valeur par défaut est la valeur produite en affectant à tous les champs de type valeur leur valeur par défaut et à tous les champs de type référence la valeur `null` .
- Pour un *nullable_type* la valeur par défaut est une instance pour laquelle la `HasValue` propriété a la valeur false et la `Value` propriété n'est pas définie. La

valeur par défaut est également connue sous le nom de ***valeur null*** du type Nullable.

Comme tout autre constructeur d'instance, le constructeur par défaut d'un type valeur est appelé à l'aide de l' `new` opérateur. Pour des raisons d'efficacité, cette exigence n'est pas destinée à faire en sorte que l'implémentation génère un appel de constructeur. Dans l'exemple ci-dessous `i` , `j` les variables et sont toutes deux initialisées à zéro.

C#

```
class A
{
    void F() {
        int i = 0;
        int j = new int();
    }
}
```

Étant donné que chaque type valeur a implicitement un constructeur d'instance sans paramètre public, il n'est pas possible pour un type struct de contenir une déclaration explicite d'un constructeur sans paramètre. Un type struct est toutefois autorisé à déclarer des constructeurs d'instances paramétrables (constructeurs).

Types struct

Un type struct est un type valeur qui peut déclarer des constantes, des champs, des méthodes, des propriétés, des indexeurs, des opérateurs, des constructeurs d'instance, des constructeurs statiques et des types imbriqués. La déclaration des types struct est décrite dans déclarations de struct.

Types simples

C# fournit un ensemble de types struct prédéfinis appelés ***types simples***. Les types simples sont identifiés par le biais de mots réservés, mais ces mots réservés sont simplement des alias pour les types struct prédéfinis dans l' `System` espace de noms, comme décrit dans le tableau ci-dessous.

Mot réservé	Type avec alias
`sbyte`	`System.SByte`
`byte`	`System.Byte`
`short`	`System.Int16`

Mot réservé	Type avec alias
`ushort`	`System.UInt16`
`int`	`System.Int32`
`uint`	`System.UInt32`
`long`	`System.Int64`
`ulong`	`System.UInt64`
`char`	`System.Char`
`float`	`System.Single`
`double`	`System.Double`
`bool`	`System.Boolean`
`decimal`	`System.Decimal`

Étant donné qu'un type simple est un alias de type struct, chaque type simple a des membres. Par exemple, `int` a les membres déclarés dans `System.Int32` et les membres hérités de `System.Object` , et les instructions suivantes sont autorisées :

C#

```
int i = int.MaxValue;           // System.Int32.MaxValue constant
string s = i.ToString();        // System.Int32.ToString() instance method
string t = 123.ToString();      // System.Int32.ToString() instance method
```

Les types simples diffèrent des autres types struct en ce qu'ils autorisent des opérations supplémentaires :

- La plupart des types simples autorisent la création de valeurs en écrivant des *littéraux* (littéraux). Par exemple, `123` est un littéral de type `int` et `'a'` est un littéral de type `char` . C# ne fournit pas de configuration pour les littéraux de types struct en général, et les valeurs autres que celles par défaut des autres types struct sont finalement toujours créées via les constructeurs d'instance de ces types struct.
- Quand les opérandes d'une expression sont toutes des constantes de type simple, il est possible que le compilateur évalue l'expression au moment de la compilation. Une telle expression est appelée *constant_expression* (expressions constantes). Les expressions qui impliquent des opérateurs définis par d'autres types struct ne sont pas considérées comme des expressions constantes.

- Par `const` le biais de déclarations, il est possible de déclarer des constantes des types simples (constantes). Il n'est pas possible d'avoir des constantes d'autres types struct, mais un effet similaire est fourni par les `static readonly` champs.
- Les conversions impliquant des types simples peuvent participer à l'évaluation des opérateurs de conversion définis par d'autres types struct, mais un opérateur de conversion défini par l'utilisateur ne peut jamais participer à l'évaluation d'un autre opérateur défini par l'utilisateur (évaluation de conversions définies parl'utilisateur).

Types intégraux

C# prend en charge neuf types intégraux : `sbyte` , `byte` ,,,, `short` `ushort` `int` `uint` , `long` , `ulong` et `char` . Les types intégraux ont les tailles et plages de valeurs suivantes :

- Le `sbyte` type représente des entiers 8 bits signés dont les valeurs sont comprises entre-128 et 127.
- Le `byte` type représente des entiers 8 bits non signés dont la valeur est comprise entre 0 et 255.
- Le `short` type représente des entiers 16 bits signés dont les valeurs sont comprises entre-32768 et 32767.
- Le `ushort` type représente des entiers 16 bits non signés dont la valeur est comprise entre 0 et 65535.
- Le `int` type représente des entiers 32 bits signés dont les valeurs sont comprises entre-2147483648 et 2147483647.
- Le `uint` type représente des entiers 32 bits non signés dont la valeur est comprise entre 0 et 4294967295.
- Le `long` type représente des entiers 64 bits signés dont la valeur est comprise entre-9223372036854775808 et 9223372036854775807.
- Le `ulong` type représente des entiers 64 bits non signés dont la valeur est comprise entre 0 et 18446744073709551615.
- Le `char` type représente des entiers 16 bits non signés dont la valeur est comprise entre 0 et 65535. Le jeu de valeurs possibles pour le type `char` correspond au jeu de caractères Unicode. Bien que `char` ait la même représentation que `ushort` , les opérations autorisées sur un type ne sont pas toutes autorisées sur l'autre.

Les opérateurs unaires et binaires de type intégral fonctionnent toujours avec la précision 32 bits signée, la précision 32 bits non signée, la précision 64 bits signée ou la précision non signée de 64 bits :

- Pour les opérateurs unaires `+` et `~` , l'opérande est converti en type `T` , où `T` est le premier de `int` ,, `uint` `long` et `ulong` qui peut représenter entièrement toutes les valeurs possibles de l'opérande. L'opération est ensuite effectuée à l'aide de la précision de type `T` et le type du résultat est `T` .
- Pour l'opérateur unaire `-` , l'opérande est converti en type `T` , où `T` est le premier de `int` et `long` qui peut représenter entièrement toutes les valeurs possibles de l'opérande. L'opération est ensuite effectuée à l'aide de la précision de type `T` et le type du résultat est `T` . L'opérateur unaire `-` ne peut pas être appliqué aux opérandes de type `ulong` .
- Pour les opérateurs binaires,,,,,, `+` `-` `*` `/` `%` `&` `^` , `|` ,, `==` `!=` , `>` ,, `<` `>=` et `<=` , les opérandes sont convertis en type `T` , où `T` est le premier de `int` , `uint` , `long` et `ulong` qui peut représenter entièrement toutes les valeurs possibles des deux opérandes. L'opération est ensuite effectuée à l'aide de la précision de type `T` et le type du résultat est `T` (ou `bool` pour les opérateurs relationnels). Il n'est pas autorisé qu'un opérande soit de type `long` et que l'autre soit de type `ulong` avec les opérateurs binaires.
- Pour les opérateurs binaires `<<` et `>>` , l'opérande de gauche est converti en type `T` , où `T` est le premier de `int` ,, `uint` `long` et `ulong` qui peut représenter entièrement toutes les valeurs possibles de l'opérande. L'opération est ensuite effectuée à l'aide de la précision de type `T` et le type du résultat est `T` .

Le `char` type est classé en tant que type intégral, mais il diffère des autres types intégraux de deux manières :

- Il n'existe pas de conversion implicite d'autres types vers le `char` type. En particulier, même si les `sbyte` `byte` types, et `ushort` ont des plages de valeurs qui sont entièrement représentables à l'aide du `char` type, les conversions implicites de `sbyte` , `byte` ou `ushort` en `char` n'existent pas.
- Les constantes du `char` type doivent être écrites en tant que *character_literal* s ou en tant que *integer_literal* s en combinaison avec un cast en type `char` . Par exemple, `(char)10` est identique à `'\x000A'`.

Les `checked` `unchecked` opérateurs et et sont utilisés pour contrôler le contrôle de dépassement de capacité pour les opérations arithmétiques de type intégral et les conversions (opérateurs activés et désactivés). Dans un `checked` contexte, un dépassement de capacité génère une erreur au moment de la compilation ou `System.OverflowException` lève une exception. Dans un `unchecked` contexte, les dépassements de capacité sont ignorés et tous les bits de poids fort qui ne tiennent pas dans le type de destination sont ignorés.

Types à virgule flottante

C# prend en charge deux types à virgule flottante : `float` et `double` . Les `float` `double` types et sont représentés à l'aide des formats IEEE 754 à double précision 32 bits et 64 bits, qui fournissent les ensembles de valeurs suivants :

- Zéro positif et zéro négatif. Dans la plupart des cas, le zéro positif et le zéro négatif se comportent de la même façon que la valeur simple zéro, mais certaines opérations distinguent les deux (opérateur de division).
- Infini positif et infini négatif. Les infinis sont produits par des opérations telles que la division d'un nombre différent de zéro par zéro. Par exemple, `1.0 / 0.0` produit l'infini positif et `-1.0 / 0.0` produit l'infini négatif.
- Valeur ***not-a-Number*** , souvent abrégée Nan. Les valeurs NaN sont produites par des opérations à virgule flottante non valides, telles que la division de zéro par zéro.
- Jeu fini de valeurs non null de la forme `s * m * 2^e` , où `s` est 1 ou-1, et `m` et `e` sont déterminés par le type à virgule flottante particulier : pour, `float` `0 < m < 2^24` et `-149 <= e <= 104` , et pour `double` , `0 < m < 2^53` et `-1075 <= e <= 970` . Les nombres à virgule flottante dénormalisés sont considérés comme des valeurs non nulles valides.

Le `float` type peut représenter des valeurs comprises entre approximativement `1.5 * 10^-45` et `3.4 * 10^38` avec une précision de 7 chiffres.

Le `double` type peut représenter des valeurs comprises entre approximativement `5.0 * 10^-324` et `1.7 × 10^308` avec une précision de 15-16 chiffres.

Si l'un des opérandes d'un opérateur binaire est d'un type à virgule flottante, l'autre opérande doit être d'un type intégral ou d'un type à virgule flottante, et l'opération est évaluée comme suit :

- Si l'un des opérandes est d'un type intégral, cet opérande est converti en type à virgule flottante de l'autre opérande.
- Ensuite, si l'un des opérandes est de type `double` , l'autre opérande est converti en `double` , l'opération est effectuée à l'aide d'au moins la `double` plage et la précision, et le type du résultat est `double` (ou `bool` pour les opérateurs relationnels).
- Dans le cas contraire, l'opération est effectuée à l'aide d'au moins la `float` plage et la précision, et le type du résultat est `float` (ou `bool` pour les opérateurs relationnels).

Les opérateurs à virgule flottante, y compris les opérateurs d'assignation, ne produisent jamais d'exceptions. À la place, dans des situations exceptionnelles, les opérations à virgule flottante produisent zéro, Infinity ou NaN, comme décrit ci-dessous :

- Si le résultat d'une opération à virgule flottante est trop petit pour le format de destination, le résultat de l'opération devient un zéro positif ou un zéro négatif.
- Si le résultat d'une opération à virgule flottante est trop grand pour le format de destination, le résultat de l'opération devient une infini positif ou négatif.
- Si une opération à virgule flottante n'est pas valide, le résultat de l'opération devient NaN.
- Si l'un ou les deux opérandes d'une opération à virgule flottante sont NaN, le résultat de l'opération devient NaN.

Les opérations à virgule flottante peuvent être effectuées avec une précision supérieure à celle du type de résultat de l'opération. Par exemple, certaines architectures matérielles prennent en charge un type à virgule flottante « étendu » ou « long double » avec une plage et une précision supérieures à celles du `double` type, et effectuent implicitement toutes les opérations en virgule flottante à l'aide de ce type de précision plus élevée. En ce qui concerne uniquement le coût excessif des performances, de telles architectures matérielles peuvent être effectuées pour effectuer des opérations à virgule flottante avec moins de précision, et au lieu d'exiger une implémentation pour perdre à la fois les performances et la précision, C# permet l'utilisation d'un type de précision plus élevé pour toutes les opérations à virgule flottante. En dehors de la diffusion de résultats plus précis, cela a rarement des effets mesurables. Toutefois, dans les expressions de la forme `x * y / z` , où la multiplication produit un résultat en dehors de la `double` plage, mais la Division suivante ramène le résultat temporaire `double` à la plage, le fait que l'expression soit évaluée dans un format de plage plus élevé peut entraîner la production d'un résultat fini au lieu d'une infini.

Type décimal

Le type `decimal` est un type de données 128 bits adapté aux calculs financiers et monétaires. Le `decimal` type peut représenter des valeurs comprises entre `1.0 * 10^-28` et environ `7.9 * 10^28` 28-29 chiffres significatifs.

L'ensemble fini de valeurs de type `decimal` se présente sous la forme `(-1)^s * c * 10^-e` , où le signe `s` est 0 ou 1, le coefficient `c` est donné par `0 <= *c* < 2^96` , et l'échelle `e` est telle que `0 <= e <= 28` . Le `decimal` type ne prend pas en charge les zéros, les infinis ou les Nan signés. Un `decimal` est représenté sous la forme d'un entier 96 bits mis à l'échelle par une puissance de dix. Pour les `decimal` s avec une valeur absolue inférieure à `1.0m` , la valeur est exacte jusqu'au 28ème décimal, mais pas plus. Pour

`decimal` les s avec une valeur absolue supérieure ou égale à `1.0m` , la valeur est exacte à 28 ou 29 chiffres. Contrairement aux `float` types de `double` données et, les nombres fractionnaires décimaux tels que 0,1 peuvent être représentés exactement dans la `decimal` représentation. Dans les `float` `double` représentations et, ces nombres sont souvent des fractions infinies, ce qui rend ces représentations plus sujettes aux erreurs d'arrondi.

Si l'un des opérandes d'un opérateur binaire est de type `decimal` , l'autre opérande doit être de type intégral ou de type `decimal` . Si un opérande de type intégral est présent, il est converti en `decimal` avant l'exécution de l'opération.

Le résultat d'une opération sur des valeurs de type `decimal` est celui qui résulterait du calcul d'un résultat exact (préservation de l'échelle, tel que défini pour chaque opérateur), puis arrondi pour s'ajuster à la représentation. Les résultats sont arrondis à la valeur représentable la plus proche et, lorsqu'un résultat est proche de deux valeurs représentables, à la valeur qui a un nombre pair dans la position de chiffre la moins significative (c'est ce qu'on appelle le « arrondi bancaire »). Un résultat zéro a toujours un signe de 0 et une échelle de 0.

Si une opération arithmétique décimale produit une valeur inférieure ou égale à la `5 * 10^-29` valeur absolue, le résultat de l'opération devient zéro. Si une `decimal` opération arithmétique produit un résultat trop grand pour le `decimal` format, une `System.OverflowException` est levée.

Le `decimal` type a une plus grande précision, mais une plage plus petite que les types à virgule flottante. Ainsi, les conversions des types à virgule flottante en `decimal` peuvent générer des exceptions de dépassement de capacité et les conversions de `decimal` en types à virgule flottante peuvent entraîner une perte de précision. Pour ces raisons, il n'existe aucune conversion implicite entre les types à virgule flottante et `decimal` , et sans casts explicites, il n'est pas possible de mélanger des opérandes à virgule flottante et `decimal` des opérandes dans la même expression.

Type bool

Le `bool` type représente des quantités logiques booléennes. Les valeurs possibles de type `bool` sont `true` et `false` .

Il n'existe pas de conversion standard entre `bool` et d'autres types. En particulier, le `bool` type est distinct et distinct des types intégraux, et une `bool` valeur ne peut pas être utilisée à la place d'une valeur intégrale, et vice versa.

Dans les langages C et C++, il s'agit d'une valeur intégrale ou à virgule flottante nulle, ou un pointeur NULL peut être converti en valeur booléenne `false` , et une valeur intégrale ou à virgule flottante différente de zéro, ou un pointeur non NULL peut être converti en valeur booléenne `true` . En C#, ces conversions sont accomplies en comparant explicitement une valeur intégrale ou à virgule flottante à zéro, ou en comparant explicitement une référence d'objet à `null` .

Types d'énumération

Un type énumération est un type distinct avec des constantes nommées. Chaque type énumération a un type sous-jacent, qui doit être,,,, `byte` `sbyte` `short` `ushort` `int` `uint` `long` ou `ulong` . L'ensemble de valeurs du type énumération est le même que l'ensemble de valeurs du type sous-jacent. Les valeurs du type d'énumération ne sont pas limitées aux valeurs des constantes nommées. Les types énumération sont définis par le biais de déclarations d'énumération (déclarations enum).

Types Nullable

Un type Nullable peut représenter toutes les valeurs de son ***type sous-jacent*** plus une valeur null supplémentaire. Un type Nullable est écrit `T?` , où `T` est le type sous-jacent. Cette syntaxe est raccourcie pour `System.Nullable<T>` , et les deux formulaires peuvent être utilisés indifféremment.

Un ***type valeur qui n'autorise pas*** les valeurs NULL est inversement tout type valeur autre que `System.Nullable<T>` et son raccourci `T?` (pour Any `T`), plus tout paramètre de type qui est contrainte comme étant un type valeur non Nullable (autrement dit, tout paramètre de type avec une `struct` contrainte). Le `System.Nullable<T>` type spécifie la contrainte de type valeur pour `T` (contraintes de paramètre de type), ce qui signifie que le type sous-jacent d'un type Nullable peut être n'importe quel type valeur n'acceptant pas les valeurs NULL. Le type sous-jacent d'un type Nullable ne peut pas être un type Nullable ou un type référence. Par exemple, `int??` et `string?` sont des types non valides.

Une instance d'un type Nullable `T?` a deux propriétés publiques en lecture seule :

- Une `HasValue` propriété de type `bool`
- Une `Value` propriété de type `T`

Une instance pour laquelle `HasValue` a la valeur true est dite non null. Une instance non null contient une valeur connue et `Value` retourne cette valeur.

Une instance pour laquelle `HasValue` a la valeur false est considérée comme étant null. Une instance null a une valeur non définie. Si vous tentez de lire le `Value` d'une instance null, une `System.InvalidOperationException` exception est levée. Le processus d'accès à la `Value` propriété d'une instance Nullable est appelé ***désencapsulage***.

En plus du constructeur par défaut, chaque type Nullable `T?` a un constructeur public qui accepte un argument unique de type `T` . À partir d'une valeur `x` de type `T` , un appel de constructeur de la forme

C#

```
new T?(x)
```

crée une instance non null de `T?` pour laquelle la `Value` propriété a la valeur `x` . Le processus de création d'une instance non null d'un type Nullable pour une valeur donnée est appelé « ***Wrapping***».

Les conversions implicites sont disponibles à partir du `null` littéral vers `T?` (conversions de littéraux NULL) et de à (conversions `T` `T?` Nullable implicites).

Types référence

Un type référence est un type de classe, un type interface, un type tableau ou un type délégué.

antlr

```
reference_type
    : class_type
    | interface_type
    | array_type
    | delegate_type
    ;

class_type
    : type_name
    | 'object'
    | 'dynamic'
    | 'string'
    ;

interface_type
    : type_name
    ;

array_type
```

```
    : non_array_type rank_specifier+
    ;

non_array_type
    : type
    ;

rank_specifier
    : '[' dim_separator* ']'
    ;

dim_separator
    : ','
    ;

delegate_type
    : type_name
    ;
```

Une valeur de type référence est une référence à une **instance*** du type, ce dernier connu sous le nom d'*objet* _ * * *. La valeur spéciale `null` est compatible avec tous les types de référence et indique l'absence d'une instance.

Types de classe

Un type de classe définit une structure de données qui contient des données membres (constantes et champs), des fonctions membres (méthodes, propriétés, événements, indexeurs, opérateurs, constructeurs d'instance, destructeurs et constructeurs statiques) et des types imbriqués. Les types de classe prennent en charge l'héritage, mécanisme dans lequel les classes dérivées peuvent étendre et spécialiser les classes de base. Les instances de types de classe sont créées à l'aide de *object_creation_expression* s (expressions de création d'objet).

Les types de classe sont décrits dans les classes.

Certains types de classe prédéfinis ont une signification particulière dans le langage C#, comme décrit dans le tableau ci-dessous.

Type de classe	Description
`System.Object`	Classe de base fondamentale de tous les autres types. Consultez le type d'objet.
`System.String`	Type de chaîne du langage C#. Consultez type de chaîne.
`System.ValueType`	Classe de base de tous les types valeur. Consultez le type System. ValueType.

Type de classe	Description
`System.Enum`	Classe de base de tous les types ENUM. Consultez enums.
`System.Array`	Classe de base de tous les types tableau. Consultez Tableaux.
`System.Delegate`	Classe de base de tous les types délégués. Consultez délégués.
`System.Exception`	Classe de base de tous les types d'exception. Consultez exceptions.

Type d'objet.

Le `object` type de classe est la classe de base fondamentale de tous les autres types. Chaque type en C# dérive directement ou indirectement du `object` type de classe.

Le mot clé `object` est simplement un alias pour la classe prédéfinie `System.Object` .

Type dynamique

Le `dynamic` type, par exemple `object` , peut faire référence à n'importe quel objet. Lorsque des opérateurs sont appliqués à des expressions de type `dynamic` , leur résolution est différée jusqu'à ce que le programme soit exécuté. Ainsi, si l'opérateur ne peut pas être légalement appliqué à l'objet référencé, aucune erreur n'est donnée durant la compilation. Au lieu de cela, une exception est levée lorsque la résolution de l'opérateur échoue au moment de l'exécution.

Son objectif est d'autoriser la liaison dynamique, qui est décrite en détail dans liaison dynamique.

`dynamic` est considéré comme identique à `object` , sauf dans les points suivants :

- Les opérations sur les expressions de type `dynamic` peuvent être liées dynamiquement (liaison dynamique).
- L'inférence de type (inférence de type) préfère `dynamic` `object` si les deux sont des candidats.

En raison de cette équivalence, les éléments suivants sont les suivants :

- Il existe une conversion d'identité implicite entre `object` et `dynamic` , et entre les types construits qui sont identiques lors du remplacement `dynamic` de par `object`
- Les conversions implicites et explicites vers et à partir de `object` s'appliquent également à et à partir de `dynamic` .

- Les signatures de méthode qui sont les mêmes lors du remplacement `dynamic` de par `object` sont considérées comme la même signature
- Le type `dynamic` ne peut pas être distingué de `object` au moment de l'exécution.
- Une expression du type `dynamic` est appelée ***expression dynamique***.

Type de chaîne

Le `string` type est un type de classe sealed qui hérite directement de `object` . Les instances de la `string` classe représentent des chaînes de caractères Unicode.

Les valeurs du `string` type peuvent être écrites sous la forme de littéraux de chaîne (littéraux de chaîne).

Le mot clé `string` est simplement un alias pour la classe prédéfinie `System.String` .

Types interface

Une interface définit un contrat. Une classe ou un struct qui implémente une interface doit adhérer à son contrat. Une interface peut hériter de plusieurs interfaces de base, et une classe ou un struct peut implémenter plusieurs interfaces.

Les types d'interface sont décrits dans interfaces.

Types de tableaux

Un tableau est une structure de données qui contient zéro, une ou plusieurs variables accessibles par le biais d'index calculés. Les variables contenues dans un tableau, également appelé éléments du tableau, sont tous du même type, et ce type est appelé type d'élément du tableau.

Les types tableau sont décrits dans les tableaux.

Types délégués

Un délégué est une structure de données qui fait référence à une ou plusieurs méthodes. Pour les méthodes d'instance, elle fait également référence aux instances d'objet correspondantes.

L'équivalent le plus proche d'un délégué en C ou C++ est un pointeur de fonction, mais alors qu'un pointeur de fonction ne peut faire référence qu'à des fonctions statiques, un délégué peut faire référence à la fois à des méthodes statiques et d'instance. Dans ce

dernier cas, le délégué stocke non seulement une référence au point d'entrée de la méthode, mais également une référence à l'instance d'objet sur laquelle appeler la méthode.

Les types délégués sont décrits dans délégués.

Boxing et unboxing

Le concept de boxing et d'unboxing est central pour le système de type de C#. Il fournit un pont entre *Value_type* s et *reference_type* s en autorisant la conversion d'une valeur d'un *Value_type* en type et inversement `object` . Le boxing et l'unboxing permettent une vue unifiée du système de type dans lequel une valeur de n'importe quel type peut finalement être traitée comme un objet.

Conversions boxing

Une conversion boxing permet à un *Value_type* d'être implicitement converti en *reference_type*. Les conversions boxing suivantes existent :

- Depuis n'importe quel *Value_type* vers le type `object` .
- Depuis n'importe quel *Value_type* vers le type `System.ValueType` .
- De tout *non_nullable_value_type* à n'importe quel *INTERFACE_TYPE* implémenté par le *Value_type*.
- De tout *nullable_type* à n'importe quel *INTERFACE_TYPE* implémenté par le type sous-jacent de l' *nullable_type*.
- Depuis n'importe quel *enum_type* vers le type `System.Enum` .
- À partir de tout *nullable_type* avec un *enum_type* sous-jacent au type `System.Enum` .
- Notez qu'une conversion implicite à partir d'un paramètre de type sera exécutée comme conversion boxing si, au moment de l'exécution, elle finit par convertir un type valeur en un type référence (conversions implicites impliquant des paramètres de type).

Le boxing d'une valeur d'une *non_nullable_value_type* consiste à allouer une instance d'objet et à copier la valeur *non_nullable_value_type* dans cette instance.

Le boxing d'une valeur d'une *nullable_type* produit une référence null s'il s'agit de la `null` valeur (a la valeur `HasValue` `false`), ou le résultat de la désencapsulation et du boxing de la valeur sous-jacente dans le cas contraire.

Le processus réel de boxing d'une valeur d'une *non_nullable_value_type* est expliqué plus particulièrement en imaginant l'existence d'une ***classe Boxing*** générique, qui se

comporte comme si elle avait été déclarée comme suit :

```
C#

sealed class Box<T>: System.ValueType
{
    T value;

    public Box(T t) {
        value = t;
    }
}
```

Le boxing d'une valeur `v` de type `T` se compose maintenant de l'exécution de l'expression `new Box<T>(v)` et du retour de l'instance résultante sous la forme d'une valeur de type `object` . Ainsi, les instructions

```
C#

int i = 123;
object box = i;
```

correspondent de manière conceptuelle à

```
C#

int i = 123;
object box = new Box<int>(i);
```

Une classe de boxing comme `Box<T>` ci-dessus n'existe pas réellement et le type dynamique d'une valeur boxed n'est pas en fait un type de classe. Au lieu de cela, une valeur boxed de type `T` a le type dynamique `T` , et une vérification de type dynamique à l'aide de l' `is` opérateur peut simplement référencer le type `T` . Par exemple,

```
C#

int i = 123;
object box = i;
if (box is int) {
    Console.Write("Box contains an int");
}
```

génère la chaîne « `Box contains an int` » sur la console.

Une conversion boxing implique de faire une copie de la valeur convertie. Cela est différent d'une conversion d'un *reference_type* en type `object` , dans lequel la valeur continue à faire référence à la même instance et est simplement considérée comme le type le moins dérivé `object` . Par exemple, étant donné la déclaration

C#

```
struct Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

les instructions suivantes

C#

```
Point p = new Point(10, 10);
object box = p;
p.x = 20;
Console.Write(((Point)box).x);
```

génère la valeur 10 dans la console, car l'opération boxing implicite qui se produit dans l'assignation de `p` à `box` entraîne la copie de la valeur de `p` . A `Point` été déclaré `class` à la place, la valeur 20 est sortie, car `p` et `box` référencent la même instance.

Conversions unboxing

Une conversion unboxing permet à un *reference_type* d'être converti explicitement en *Value_type*. Les conversions unboxing suivantes existent :

- Du type `object` à n'importe quel *Value_type*.
- Du type `System.ValueType` à n'importe quel *Value_type*.
- De tout *INTERFACE_TYPE* à n'importe quel *non_nullable_value_type* qui implémente le *INTERFACE_TYPE*.
- De tout *INTERFACE_TYPE* à n'importe quel *nullable_type* dont le type sous-jacent implémente le *INTERFACE_TYPE*.
- Du type `System.Enum` à n'importe quel *enum_type*.
- Du type `System.Enum` à n'importe quel *nullable_type* avec un *enum_type* sous-jacent.

- Notez qu'une conversion explicite en paramètre de type sera exécutée comme conversion unboxing si, au moment de l'exécution, elle finit par convertir un type référence en un type valeur (conversions dynamiques explicites).

Une opération d'unboxing sur un *non_nullable_value_type* consiste à vérifier en premier lieu que l'instance d'objet est une valeur boxed du *non_nullable_value_type* donné, puis à copier la valeur hors de l'instance.

La conversion unboxing en *nullable_type* produit la valeur null du *nullable_type* si l'opérande source est `null` ou le résultat encapsulé de la conversion unboxing de l'instance d'objet vers le type sous-jacent de la *nullable_type* dans le cas contraire.

En faisant référence à la classe de boxing imaginaire décrite dans la section précédente, une conversion unboxing d'un objet `box` en *Value_type* `T` consiste à exécuter l'expression `((Box<T>)box).value` . Ainsi, les instructions

```
C#

object box = 123;
int i = (int)box;
```

correspondent de manière conceptuelle à

```
C#

object box = new Box<int>(123);
int i = ((Box<int>)box).value;
```

Pour qu'une conversion unboxing vers un *non_nullable_value_type* donné aboutisse au moment de l'exécution, la valeur de l'opérande source doit être une référence à une valeur boxed de ce *non_nullable_value_type*. Si l'opérande source est `null` , une `System.NullReferenceException` exception est levée. Si l'opérande source est une référence à un objet incompatible, une `System.InvalidCastException` exception est levée.

Pour qu'une conversion unboxing vers un *nullable_type* donné aboutisse au moment de l'exécution, la valeur de l'opérande source doit être `null` ou une référence à une valeur boxed du *non_nullable_value_type* sous-jacent de l' *nullable_type*. Si l'opérande source est une référence à un objet incompatible, une `System.InvalidCastException` exception est levée.

Types construits

Une déclaration de type générique, par elle-même, dénote un *__type générique non lié__, qui est utilisé comme « Blueprint » pour former de nombreux types différents, en appliquant des *arguments de type*. Les arguments de type sont écrits entre crochets pointus (`<` et `>`) juste après le nom du type générique. Un type qui comprend au moins un argument de type est appelé *type construit*. Un type construit peut être utilisé à la plupart des emplacements dans le langage dans lequel un nom de type peut apparaître. Un type générique indépendant peut uniquement être utilisé dans un _typeof_expression * (opérateur typeof).

Les types construits peuvent également être utilisés dans les expressions en tant que noms simples (noms simples) ou lors de l'accès à un membre (accès aux membres).

Lorsqu'un *namespace_or_type_name* est évalué, seuls les types génériques avec le nombre correct de paramètres de type sont pris en compte. Par conséquent, il est possible d'utiliser le même identificateur pour identifier les différents types, à condition que les types aient des nombres de paramètres de type différents. Cela est utile pour mélanger des classes génériques et non génériques dans le même programme :

C#

```
namespace Widgets
{
    class Queue {...}
    class Queue<TElement> {...}
}

namespace MyApplication
{
    using Widgets;

    class X
    {
        Queue q1;             // Non-generic Widgets.Queue
        Queue<int> q2;        // Generic Widgets.Queue
    }
}
```

Un *type_name* peut identifier un type construit bien qu'il ne spécifie pas directement les paramètres de type. Cela peut se produire lorsqu'un type est imbriqué dans une déclaration de classe générique et que le type d'instance de la déclaration conteneur est implicitement utilisé pour la recherche de nom (types imbriqués dans les classes génériques) :

C#

```
class Outer<T>
{
    public class Inner {...}

    public Inner i;                // Type of i is Outer<T>.Inner
}
```

Dans du code unsafe, un type construit ne peut pas être utilisé en tant que *unmanaged_type* (types pointeur).

Arguments de type

Chaque argument dans une liste d'arguments de type est simplement un *type*.

antlr

```
type_argument_list
    : '<' type_arguments '>'
    ;

type_arguments
    : type_argument (',' type_argument)*
    ;

type_argument
    : type
    ;
```

Dans du code non sécurisé (code non sécurisé), un *type_argument* ne peut pas être un type pointeur. Chaque argument de type doit satisfaire toutes les contraintes sur le paramètre de type correspondant (contraintes de paramètre de type).

Types ouverts et fermés

Tous les types peuvent être classés en tant que * types **ouverts** _ ou _ *types fermés* *. Un type ouvert est un type qui implique des paramètres de type. Plus précisément :

- Un paramètre de type définit un type ouvert.
- Un type tableau est un type ouvert si et uniquement si son type d'élément est un type ouvert.
- Un type construit est un type ouvert si et seulement si un ou plusieurs de ses arguments de type est un type ouvert. Un type imbriqué construit est un type ouvert si et seulement si un ou plusieurs de ses arguments de type ou les arguments de type de ses types conteneur sont un type ouvert.

Un type fermé est un type qui n'est pas un type ouvert.

Au moment de l'exécution, tout le code au sein d'une déclaration de type générique est exécuté dans le contexte d'un type construit fermé qui a été créé en appliquant des arguments de type à la déclaration générique. Chaque paramètre de type dans le type générique est lié à un type au moment de l'exécution particulier. Le traitement au moment de l'exécution de toutes les instructions et expressions se produit toujours avec les types fermés, et les types ouverts se produisent uniquement pendant le traitement au moment de la compilation.

Chaque type construit fermé a son propre ensemble de variables statiques, qui ne sont pas partagées avec d'autres types construits fermés. Étant donné qu'il n'existe pas de type ouvert au moment de l'exécution, aucune variable statique n'est associée à un type ouvert. Deux types construits fermés sont du même type s'ils sont construits à partir du même type générique indépendant, et que leurs arguments de type correspondants sont du même type.

Types liés et indépendants

Le terme *__type indépendant__ _ fait référence à un type non générique ou à un type générique indépendant. Le terme _ *Bound type** fait référence à un type non générique ou à un type construit.

Un type indépendant fait référence à l'entité déclarée par une déclaration de type. Un type générique indépendant n'est pas lui-même un type et ne peut pas être utilisé en tant que type d'une variable, d'un argument ou d'une valeur de retour, ou en tant que type de base. La seule construction dans laquelle un type générique indépendant peut être référencé est l' `typeof` expression (opérateur typeof).

Contraintes de satisfaction

Chaque fois qu'un type construit ou une méthode générique est référencé, les arguments de type fournis sont vérifiés par rapport aux contraintes de paramètre de type déclarées sur le type ou la méthode générique (contraintes de paramètre de type). Pour chaque `where` clause, l'argument de type `A` qui correspond au paramètre de type nommé est vérifié par rapport à chaque contrainte comme suit :

- Si la contrainte est un type de classe, un type d'interface ou un paramètre de type, Let `C` représente cette contrainte avec les arguments de type fournis, en remplaçant tous les paramètres de type qui apparaissent dans la contrainte. Pour satisfaire la contrainte, il faut que le type `A` soit convertible en type `C` par l'un des éléments suivants :

    - Conversion d'identité (conversion d'identité)
    - Conversion de référence implicite (conversions de référence implicites)
    - Conversion boxing (conversions boxing), à condition que le type a soit un type valeur n'acceptant pas les valeurs NULL.
    - Référence implicite, conversion boxing ou paramètre de type d'un paramètre de type `A` en `C` .
- Si la contrainte est la contrainte de type référence (`class`), le type `A` doit correspondre à l'un des éléments suivants :
    - `A` est un type d'interface, un type de classe, un type délégué ou un type tableau. Notez que `System.ValueType` et `System.Enum` sont des types référence qui satisfont cette contrainte.
    - `A` est un paramètre de type qui est connu comme étant un type référence (contraintes de paramètre de type).
- Si la contrainte est la contrainte de type valeur (`struct`), le type `A` doit correspondre à l'un des éléments suivants :
    - `A` est un type struct ou un type enum, mais pas un type Nullable. Notez que `System.ValueType` et `System.Enum` sont des types référence qui ne répondent pas à cette contrainte.
    - `A` est un paramètre de type ayant la contrainte de type valeur (contraintes de paramètre de type).
- Si la contrainte est la contrainte de constructeur `new()` , le type `A` ne doit pas être `abstract` et doit avoir un constructeur sans paramètre public. Cette condition est remplie si l'une des conditions suivantes est remplie :
    - `A` est un type valeur, car tous les types valeur ont un constructeur public par défaut (constructeurs par défaut).
    - `A` est un paramètre de type ayant la contrainte de constructeur (contraintes de paramètre de type).
    - `A` est un paramètre de type ayant la contrainte de type valeur (contraintes de paramètre de type).
    - `A` est une classe qui n'est pas `abstract` et qui contient un constructeur déclaré explicitement sans `public` paramètres.
    - `A` n'est pas `abstract` et a un constructeur par défaut (constructeurs par défaut).

Une erreur de compilation se produit si une ou plusieurs des contraintes d'un paramètre de type ne sont pas satisfaites par les arguments de type donnés.

Étant donné que les paramètres de type ne sont pas hérités, les contraintes ne sont jamais héritées non plus. Dans l'exemple ci-dessous, `D` doit spécifier la contrainte sur son paramètre de type `T` afin qu'elle `T` satisfasse à la contrainte imposée par la classe

de base `B<T>` . En revanche, `E` la classe n'a pas besoin de spécifier de contrainte, car `List<T>` implémente `IEnumerable` pour Any `T` .

C#

```
class B<T> where T: IEnumerable {...}

class D<T>: B<T> where T: IEnumerable {...}

class E<T>: B<List<T>> {...}
```

Paramètres de type

Un paramètre de type est un identificateur désignant un type valeur ou un type référence auquel le paramètre est lié au moment de l'exécution.

antlr

```
type_parameter
    : identifier
    ;
```

Étant donné qu'un paramètre de type peut être instancié avec un grand nombre d'arguments de type réels différents, les paramètres de type ont des opérations et des restrictions légèrement différentes de celles des autres types. notamment :

- Un paramètre de type ne peut pas être utilisé directement pour déclarer une classe de base (classe de base) ou une interface (listes deparamètres de type Variant).
- Les règles de recherche de membre sur les paramètres de type dépendent des contraintes, le cas échéant, appliquées au paramètre de type. Ils sont détaillés dans la recherche de membres.
- Les conversions disponibles pour un paramètre de type dépendent des contraintes, le cas échéant, appliquées au paramètre de type. Ils sont détaillés dans les conversions implicites impliquant des paramètres de type et des conversions dynamiques explicites.
- Le littéral `null` ne peut pas être converti en un type donné par un paramètre de type, sauf si le paramètre de type est connu comme étant un type référence (conversions implicites impliquant des paramètres de type). Toutefois, une `default` expression (expressions de valeur par défaut) peut être utilisée à la place. En outre, une valeur avec un type donné par un paramètre de type peut être comparée à `null` l'aide de `==` et `!=` (opérateurs d'égalité de type référence) à moins que le paramètre de type n'ait la contrainte de type valeur.

- Une `new` expression (expressions de création d'objet) peut être utilisée uniquement avec un paramètre de type si le paramètre de type est imposé par un *constructor_constraint* ou la contrainte de type valeur (contraintes de paramètre de type).
- Un paramètre de type ne peut pas être utilisé n'importe où dans un attribut.
- Un paramètre de type ne peut pas être utilisé dans un accès de membre (accès aux membres) ou un nom de type (noms d'espaces de nomset de types) pour identifier un membre statique ou un type imbriqué.
- Dans du code unsafe, un paramètre de type ne peut pas être utilisé en tant que *unmanaged_type* (types pointeur).

En tant que type, les paramètres de type sont purement une construction au moment de la compilation. Au moment de l'exécution, chaque paramètre de type est lié à un type au moment de l'exécution qui a été spécifié en fournissant un argument de type à la déclaration de type générique. Ainsi, le type d'une variable déclarée avec un paramètre de type sera, au moment de l'exécution, un type construit fermé (types ouverts et fermés). L'exécution de toutes les instructions et expressions impliquant des paramètres de type au moment de l'exécution utilise le type réel fourni comme argument de type pour ce paramètre.

Types d'arborescence d'expression

*Les **arborescences d'expressions** _ permettent aux expressions lambda d'être représentées en tant que structures de données au lieu du code exécutable. Les arborescences d'expressions sont des valeurs des *types d'arborescence d'expression** de la forme `System.Linq.Expressions.Expression<D>` , où `D` est un type délégué. Pour le reste de cette spécification, nous faisons référence à ces types à l'aide de la syntaxe abrégée `Expression<D>` .

S'il existe une conversion d'une expression lambda en un type délégué `D` , une conversion existe également dans le type d'arborescence de l'expression `Expression<D>` . Tandis que la conversion d'une expression lambda en un type délégué génère un délégué qui fait référence à du code exécutable pour l'expression lambda, la conversion en un type d'arborescence d'expression crée une représentation d'arborescence d'expression de l'expression lambda.

Les arborescences d'expressions sont des représentations efficaces des données en mémoire des expressions lambda et rendent la structure de l'expression lambda transparente et explicite.

Tout comme un type délégué `D` , `Expression<D>` est dit avoir des types de paramètre et de retour, qui sont identiques à ceux de `D` .

L'exemple suivant représente une expression lambda à la fois comme du code exécutable et comme une arborescence de l'expression. Étant donné qu'une conversion existe dans `Func<int,int>` , une conversion existe également pour `Expression<Func<int,int>>` :

```
C#

Func<int,int> del = x => x + 1;                  // Code

Expression<Func<int,int>> exp = x => x + 1;      // Data
```

À la suite de ces assignations, le délégué `del` référence une méthode qui retourne `x + 1` , et l'arborescence de l'expression `exp` fait référence à une structure de données qui décrit l'expression `x => x + 1` .

La définition exacte du type générique `Expression<D>` , ainsi que les règles précises pour la construction d'une arborescence d'expressions quand une expression lambda est convertie en un type d'arborescence d'expression, sont toutes deux en dehors de la portée de cette spécification.

Deux choses sont importantes pour rendre explicite :

- Toutes les expressions lambda ne peuvent pas être converties en arborescences d'expressions. Par exemple, les expressions lambda avec corps d'instruction et les expressions lambda contenant des expressions d'assignation ne peuvent pas être représentées. Dans ce cas, une conversion existe toujours, mais elle échoue au moment de la compilation. Ces exceptions sont détaillées dans les conversions de fonctions anonymes.

- `Expression<D>` offre une méthode d'instance `Compile` qui produit un délégué de type `D` :

  ```
  C#

  Func<int,int> del2 = exp.Compile();
  ```

  L'appel de ce délégué provoque l'exécution du code représenté par l'arborescence de l'expression. Par conséquent, étant donné les définitions ci-dessus, del et DEL2 sont équivalentes, et les deux instructions suivantes ont le même effet :

C#

```
int i1 = del(1);

int i2 = del2(1);
```

Après l'exécution de ce code `i1` , `i2` la valeur est `2` .

Variables

Article • 13/05/2021

Les variables représentent des emplacements de stockage. Chaque variable a un type qui détermine les valeurs qui peuvent être stockées dans la variable. C# est un langage de type sécurisé et le compilateur C# garantit que les valeurs stockées dans les variables sont toujours du type approprié. La valeur d'une variable peut être modifiée par le biais de l'assignation ou par le biais de l'utilisation des `++` `--` opérateurs et.

Une variable doit être ***assignée définitivement*** (assignation définie) avant que sa valeur puisse être obtenue.

Comme décrit dans les sections suivantes, les variables ont la valeur *__assignation initiale__ _ ou _ *initialement non assignées* *. Une variable initialement assignée a une valeur initiale bien définie et est toujours considérée comme assignée définitivement. Une variable initialement non assignée n'a pas de valeur initiale. Pour qu'une variable initialement non assignée soit considérée comme définitivement assignée à un certain emplacement, une assignation à la variable doit se produire dans chaque chemin d'exécution possible menant à cet emplacement.

Catégories de variables

C# définit sept catégories de variables : les variables statiques, les variables d'instance, les éléments de tableau, les paramètres de valeur, les paramètres de référence, les paramètres de sortie et les variables locales. Les sections qui suivent décrivent chacune de ces catégories.

Dans l'exemple

C#

```
class A
{
    public static int x;
    int y;

    void F(int[] v, int a, ref int b, out int c) {
        int i = 1;
        c = a + b++;
    }
}
```

`x` est une variable statique, `y` est une variable d'instance, `v[0]` est un élément de tableau, `a` est un paramètre de valeur, `b` est un paramètre de référence, `c` est un paramètre de sortie et `i` est une variable locale.

Variables statiques

Un champ déclaré avec le `static` modificateur est appelé une ***variable statique***. Une variable statique est entamée avant l'exécution du constructeur statique (constructeurs statiques) pour son type conteneur et cesse d'exister lorsque le domaine d'application associé cesse d'exister.

La valeur initiale d'une variable statique est la valeur par défaut (valeurs par défaut) du type de la variable.

À des fins de vérification de l'attribution définie, une variable statique est considérée comme initialement assignée.

Variables d'instance

Un champ déclaré sans le `static` modificateur est appelé une ***variable d'instance***.

Variables d'instance dans les classes

Une variable d'instance d'une classe intervient lorsqu'une nouvelle instance de cette classe est créée et cesse d'exister lorsqu'il n'existe aucune référence à cette instance et que le destructeur de l'instance (le cas échéant) a été exécuté.

La valeur initiale d'une variable d'instance d'une classe est la valeur par défaut (valeurs par défaut) du type de la variable.

Dans le cadre de la vérification de l'assignation définie, une variable d'instance d'une classe est considérée comme initialement assignée.

Variables d'instance dans les structs

Une variable d'instance d'un struct a exactement la même durée de vie que la variable struct à laquelle elle appartient. En d'autres termes, lorsqu'une variable d'un type struct entre en existence ou cesse d'exister, il en est de même pour les variables d'instance du struct.

L'état d'assignation initial d'une variable d'instance d'un struct est le même que celui de la variable de struct conteneur. En d'autres termes, lorsqu'une variable de struct est

considérée comme initialement assignée, il s'agit également de ses variables d'instance, et lorsqu'une variable de struct est considérée comme initialement non assignée, ses variables d'instance ne sont pas assignées de la même façon.

Éléments de tableau

Les éléments d'un tableau existent lorsqu'une instance de tableau est créée et cessent d'exister lorsqu'il n'existe aucune référence à cette instance de tableau.

La valeur initiale de chacun des éléments d'un tableau est la valeur par défaut (valeurs par défaut) du type des éléments du tableau.

Dans le cadre de la vérification de l'assignation définie, un élément de tableau est considéré comme initialement assigné.

Paramètres de valeur

Un paramètre déclaré sans `ref` `out` modificateur ou est un ***paramètre de valeur***.

Un paramètre de valeur intervient lors de l'appel de la fonction membre (méthode, constructeur d'instance, accesseur ou opérateur) ou de la fonction anonyme à laquelle le paramètre appartient, et est initialisée avec la valeur de l'argument fourni dans l'appel. Un paramètre de valeur cesse normalement d'exister lors du retour de la fonction membre ou de la fonction anonyme. Toutefois, si le paramètre de valeur est capturé par une fonction anonyme (expressions de fonction anonymes), sa durée de vie s'étend au moins jusqu'à ce que le délégué ou l'arborescence d'expression créé à partir de cette fonction anonyme soit éligible pour garbage collection.

Dans le cadre de la vérification de l'attribution définie, un paramètre de valeur est considéré comme initialement assigné.

Paramètres de référence

Un paramètre déclaré avec un `ref` modificateur est un ***paramètre de référence***.

Un paramètre de référence ne crée pas un nouvel emplacement de stockage. Au lieu de cela, un paramètre de référence représente le même emplacement de stockage que la variable donnée comme argument dans le membre de fonction ou l'appel de fonction anonyme. Ainsi, la valeur d'un paramètre de référence est toujours la même que la variable sous-jacente.

Les règles d'assignation définie suivantes s'appliquent aux paramètres de référence. Notez les différentes règles pour les paramètres de sortie décrits dans paramètres de sortie.

- Une variable doit être assignée de manière définitive (assignation définie) avant de pouvoir être passée en tant que paramètre de référence dans un membre de fonction ou un appel de délégué.
- Dans un membre de fonction ou une fonction anonyme, un paramètre de référence est considéré comme initialement assigné.

Dans une méthode d'instance ou un accesseur d'instance d'un type struct, le `this` mot clé se comporte exactement comme un paramètre de référence du type struct (cet accès).

Paramètres de sortie

Un paramètre déclaré avec un `out` modificateur est un ***paramètre de sortie***.

Un paramètre de sortie ne crée pas un nouvel emplacement de stockage. Au lieu de cela, un paramètre de sortie représente le même emplacement de stockage que la variable donnée comme argument dans le membre de fonction ou l'appel de délégué. Ainsi, la valeur d'un paramètre de sortie est toujours la même que la variable sous-jacente.

Les règles d'assignation définie suivantes s'appliquent aux paramètres de sortie. Notez les différentes règles pour les paramètres de référence décrites dans paramètres de référence.

- Une variable n'a pas besoin d'être définitivement assignée avant de pouvoir être passée en tant que paramètre de sortie dans un membre de fonction ou un appel de délégué.
- Après l'achèvement normal d'un membre de fonction ou d'un appel de délégué, chaque variable qui a été passée comme paramètre de sortie est considérée comme affectée dans ce chemin d'exécution.
- Dans un membre de fonction ou une fonction anonyme, un paramètre de sortie est considéré comme initialement non assigné.
- Chaque paramètre de sortie d'un membre de fonction ou d'une fonction anonyme doit être assignée définitivement (assignation définie) avant que la fonction membre ou la fonction anonyme retourne normalement.

Dans un constructeur d'instance d'un type struct, le `this` mot clé se comporte exactement comme un paramètre de sortie du type struct (cet accès).

Variables locales

Une **variable locale** _ * est déclarée par un _local_variable_declaration *, qui peut se produire dans un *bloc*, un *for_Statement*, un *switch_Statement* ou un *using_statement*; ou par un *foreach_statement* ou un *specific_catch_clause* pour un *try_statement*.

La durée de vie d'une variable locale correspond à la partie de l'exécution du programme pendant laquelle le stockage est garanti comme étant réservé. Cette durée de vie étend au moins l'entrée dans le *bloc*, *for_Statement*, *switch_Statement*, *using_statement*, *foreach_statement* ou *specific_catch_clause* auquel elle est associée, jusqu'à ce que l'exécution de ce *bloc*, *for_Statement* , *switch_Statement* , *using_statement*, *foreach_statement* ou *specific_catch_clause* se termine de quelque manière que ce soit. (L'entrée d'un *bloc* délimité ou l'appel d'une méthode interrompt, mais ne termine pas, l'exécution du *bloc* en cours, *for_Statement*, *switch_Statement*, *using_statement*, *foreach_statement* ou *specific_catch_clause*.) Si la variable locale est capturée par une fonction anonyme (variables externes capturées), sa durée de vie s'étend au moins jusqu'à ce que le délégué ou l'arborescence d'expressions créé à partir de la fonction anonyme, ainsi que tous les autres objets qui font référence à la variable capturée, soient éligibles pour garbage collection.

Si le *bloc* parent, *for_Statement*, *switch_Statement*, *using_statement*, *foreach_statement* ou *specific_catch_clause* est entré de manière récursive, une nouvelle instance de la variable locale est créée à chaque fois, et son *local_variable_initializer*, le cas échéant, est évaluée à chaque fois.

Une variable locale introduite par un *local_variable_declaration* n'est pas initialisée automatiquement et n'a donc pas de valeur par défaut. Dans le cadre de la vérification de l'attribution définie, une variable locale introduite par un *local_variable_declaration* est considérée comme initialement non assignée. Une *local_variable_declaration* peut inclure une *local_variable_initializer*, auquel cas la variable est considérée comme définitivement assignée uniquement après l'expression d'initialisation (instructions de déclaration).

Dans l'étendue d'une variable locale introduite par un *local_variable_declaration*, il s'agit d'une erreur au moment de la compilation pour faire référence à cette variable locale dans une position textuelle qui précède ses *local_variable_declarator*. Si la déclaration de variable locale est implicite (déclarations de variables locales), il s'agit également d'une erreur qui fait référence à la variable dans son *local_variable_declarator*.

Une variable locale introduite par un *foreach_statement* ou un *specific_catch_clause* est considérée comme assignée de manière définitive dans sa portée entière.

La durée de vie réelle d'une variable locale dépend de l'implémentation. Par exemple, un compilateur peut déterminer statiquement qu'une variable locale dans un bloc est utilisée uniquement pour une petite partie de ce bloc. À l'aide de cette analyse, le compilateur peut générer du code qui donne au stockage de la variable une durée de vie plus faible que le bloc qui le contient.

Le stockage référencé par une variable de référence locale est récupéré indépendamment de la durée de vie de cette variable de référence locale (gestion automatiquede la mémoire).

Valeurs par défaut

Les catégories de variables suivantes sont automatiquement initialisées à leurs valeurs par défaut :

- variables statiques
- Variables d'instance des instances de classe.
- Éléments du tableau.

La valeur par défaut d'une variable dépend du type de la variable et est déterminée comme suit :

- Pour une variable d'un *Value_type*, la valeur par défaut est la même que la valeur calculée par le constructeur par défaut de *Value_type*(constructeurs par défaut).
- Pour une variable d'un *reference_type*, la valeur par défaut est `null` .

L'initialisation aux valeurs par défaut est généralement effectuée en faisant en sorte que le gestionnaire de mémoire ou le garbage collector initialise la mémoire à tous les bits-zéro avant qu'il ne soit alloué pour utilisation. Pour cette raison, il est pratique d'utiliser All-bits-Zero pour représenter la référence null.

Assignation définie

À un emplacement donné dans le code exécutable d'un membre de fonction, une variable est dite ***assignée*** de manière définitive si le compilateur peut prouver, par une analyse de workflow statique particulière (règles précises pour déterminer l'assignation définie), que la variable a été initialisée automatiquement ou a été la cible d'au moins une assignation. Comme indiqué de façon informelle, les règles d'attribution définie sont les suivantes :

- Une variable initialement assignée (variables initialement affectées) est toujours considérée comme assignée définitivement.

- Une variable initialement non assignée (variables initialement non assignées) est considérée comme définitivement assignée à un emplacement donné si tous les chemins d'exécution possibles conduisant à cet emplacement contiennent au moins l'un des éléments suivants :
  - Assignation simple (assignation simple) dans laquelle la variable est l'opérande de gauche.
  - Une expression d'appel (expressions d'appel) ou une expression de création d'objet (expressions de création d'objet) qui passe la variable en tant que paramètre de sortie.
  - Pour une variable locale, déclaration de variable locale (déclarations de variable locale) qui comprend un initialiseur de variable.

La spécification formelle sous-jacente aux règles informelles ci-dessus est décrite dans variables initialement attribuées, variables initialement non assignéeset règles précises pour déterminer l'assignation définie.

Les États d'assignation définie des variables d'instance d'une variable *struct_type* sont suivis individuellement et collectivement. Outre les règles ci-dessus, les règles suivantes s'appliquent à *struct_type* variables et à leurs variables d'instance :

- Une variable d'instance est considérée comme assignée définitivement si sa variable *struct_type* conteneur est considérée comme assignée de manière définitive.
- Une variable *struct_type* est considérée comme assignée de manière définitive si chacune de ses variables d'instance est considérée comme assignée de manière définitive.

L'assignation définie est une exigence dans les contextes suivants :

- Une variable doit être assignée de manière définitive à chaque emplacement où sa valeur est obtenue. Cela permet de s'assurer que les valeurs non définies ne se produisent jamais. L'occurrence d'une variable dans une expression est considérée pour obtenir la valeur de la variable, sauf lorsque
  - la variable est l'opérande gauche d'une assignation simple.
  - la variable est transmise en tant que paramètre de sortie, ou
  - la variable est une variable *struct_type* et se produit comme opérande gauche d'un accès au membre.
- Une variable doit être assignée de manière définitive à chaque emplacement où elle est passée en tant que paramètre de référence. Cela garantit que le membre de fonction appelé peut prendre en compte le paramètre de référence initialement assigné.

- Tous les paramètres de sortie d'un membre de fonction doivent être assignés définitivement à chaque emplacement où le membre de fonction retourne (via une `return` instruction ou l'exécution atteint la fin du corps du membre de la fonction). Cela garantit que les fonctions membres ne retournent pas de valeurs non définies dans les paramètres de sortie, ce qui permet au compilateur de considérer un appel de membre de fonction qui prend une variable comme paramètre de sortie équivalent à une assignation à la variable.
- La `this` variable d'un constructeur d'instance *struct_type* doit être assignée de manière définitive à chaque emplacement retourné par le constructeur d'instance.

Variables initialement attribuées

Les catégories de variables suivantes sont classées comme initialement attribuées :

- variables statiques
- Variables d'instance des instances de classe.
- Variables d'instance de variables de struct initialement attribuées.
- Éléments du tableau.
- Paramètres de valeur.
- Paramètres de référence.
- Variables déclarées dans une `catch` clause ou une `foreach` instruction.

Variables initialement non assignées

Les catégories de variables suivantes sont classées comme étant initialement non assignées :

- Variables d'instance de variables de struct initialement non assignées.
- Paramètres de sortie, y compris la `this` variable des constructeurs d'instance de struct.
- Les variables locales, à l'exception de celles déclarées dans une `catch` clause ou une `foreach` instruction.

Règles précises pour déterminer l'assignation définie

Pour déterminer que chaque variable utilisée est assignée de manière définitive, le compilateur doit utiliser un processus qui est équivalent à celui décrit dans cette section.

Le compilateur traite le corps de chaque membre de fonction qui a une ou plusieurs variables initialement non assignées. Pour chaque variable *v* non assignée initialement,

le compilateur détermine un **État d'assignation définie** à * pour _V * à chacun des points suivants dans le membre de fonction :

- Au début de chaque instruction
- Au point de terminaison (points de terminaison et accessibilité) de chaque instruction
- Sur chaque arc qui transfère le contrôle à une autre instruction ou jusqu'au point de terminaison d'une instruction
- Au début de chaque expression
- À la fin de chaque expression

L'état d'assignation définie de *v* peut être :

- Assigné de manière définitive. Cela indique qu'une valeur a été assignée à *v* pour tous les flux de contrôle possibles à ce stade.
- Pas définitivement assigné. Pour l'état d'une variable à la fin d'une expression de type `bool` , l'état d'une variable qui n'est pas définitivement assignée peut (mais n'est pas nécessairement) appartenir à l'un des sous-États suivants :
  - Assigné définitivement après une expression true. Cet état indique que *v* est définitivement affecté si l'expression booléenne est évaluée comme true, mais n'est pas nécessairement affectée si l'expression booléenne est évaluée comme false.
  - Assigné définitivement après une expression false. Cet état indique que *v* est définitivement affecté si l'expression booléenne est évaluée comme false, mais n'est pas nécessairement affectée si l'expression booléenne est évaluée comme true.

Les règles suivantes régissent la façon dont l'état d'une variable *v* est déterminé à chaque emplacement.

Règles générales pour les instructions

- *v* n'est pas définitivement assignée au début d'un corps de membre de fonction.
- *v* est définitivement assignée au début de toute instruction inaccessible.
- L'état d'assignation définie de *v* au début d'une autre instruction est déterminé en vérifiant l'état d'assignation définie de *v* sur tous les transferts de workflow de contrôle qui ciblent le début de cette instruction. Si (et seulement si) *v* est définitivement assigné sur tous les transferts de ce type de contrôle, *v* est définitivement assigné au début de l'instruction. L'ensemble des transferts de workflow possibles est déterminé de la même façon que pour la vérification de l'accessibilité des instructions (points de terminaison et accessibilité).

- L'état d'assignation définie de *v* au point de terminaison d'un bloc,,,,,,,, ou de l' `checked` `unchecked` `if` `while` `do` `for` `foreach` `lock` `using` `switch` instruction est déterminé en vérifiant l'état d'assignation définie de *v* sur tous les transferts de workflow qui ciblent le point de terminaison de cette instruction. Si *v* est définitivement assignée à tous les transferts de ce type de contrôle, *v* est définitivement assigné au point de terminaison de l'instruction. Dispose *v* n'est pas définitivement assigné au point de terminaison de l'instruction. L'ensemble des transferts de workflow possibles est déterminé de la même façon que pour la vérification de l'accessibilité des instructions (points de terminaison et accessibilité).

Instructions de bloc, instructions vérifiées et non vérifiées

L'état d'assignation définie de *v* sur le transfert de contrôle vers la première instruction de la liste d'instructions dans le bloc (ou jusqu'au point de terminaison du bloc, si la liste d'instructions est vide) est le même que l'instruction d'assignation définie de *v* avant l'instruction de bloc, `checked` ou `unchecked` .

Instructions d'expression

Pour une instruction d'expression *stmt* qui se compose de l'expression *expr*:

- *v* a le même état d'assignation définie au début de *expr* qu'au début de *stmt*.
- Si *v* est assignée de manière définitive à la fin de *expr*, elle est définitivement assignée au point de terminaison de *stmt*; dispose elle n'est pas définitivement assignée au point de terminaison de *stmt*.

Instructions de déclaration

- Si *stmt* est une instruction de déclaration sans initialiseurs, alors *v* a le même état d'assignation définie au point de terminaison de *stmt* qu'au début de *stmt*.
- Si *stmt* est une instruction de déclaration avec des initialiseurs, l'état d'assignation définie pour *v* est déterminé comme si *stmt* était une liste d'instructions, avec une instruction d'assignation pour chaque déclaration avec un initialiseur (dans l'ordre de déclaration).

Instructions If

Pour une `if` instruction *stmt* de la forme :

```
C#
```

```
if ( expr ) then_stmt else else_stmt
```

- *v* a le même état d'assignation définie au début de *expr* qu'au début de *stmt*.
- Si *v* est définitivement assignée à la fin de *expr*, elle est définitivement assignée au transfert du workflow de contrôle à *then_stmt* et à *else_stmt* ou au point de terminaison de *stmt* s'il n'y a aucune clause Else.
- Si *v* a l'état « définitivement assigné après une expression true » à la fin de *expr*, il est définitivement affecté sur le transfert du workflow de contrôle à *then_stmt* et n'est pas définitivement assigné sur le transfert de workflow à *else_stmt* ou à l'extrémité de *stmt* s'il n'y a pas de clause Else.
- Si *v* a l'état « définitivement assigné après une expression false » à la fin de *expr*, il est définitivement affecté sur le transfert du workflow de contrôle à *else_stmt* et n'est pas définitivement assigné sur le transfert du workflow de contrôle à *then_stmt*. Elle est définitivement assignée au point de terminaison de *stmt* si et seulement si elle est définitivement assignée au point de terminaison de *then_stmt*.
- Dans le cas contraire, *v* est considéré comme n'étant pas définitivement affecté sur le transfert du workflow de contrôle vers le *then_stmt* ou *else_stmt*, ou vers le point de terminaison de *stmt* s'il n'y a aucune clause Else.

Instructions switch

Dans une `switch` instruction *stmt* avec une expression de contrôle *expr*:

- L'état d'assignation définie de *v* au début de *expr* est identique à l'état *v* au début de *stmt*.
- L'état d'assignation définie de *v* sur le transfert du workflow de contrôle vers une liste d'instructions de bloc switch accessible est le même que l'état d'assignation définie de *v* à la fin de *expr*.

Instructions while

Pour une `while` instruction *stmt* de la forme :

C#

```
while ( expr ) while_body
```

- *v* a le même état d'assignation définie au début de *expr* qu'au début de *stmt*.
- Si *v* est définitivement assignée à la fin de *expr*, elle est définitivement assignée au transfert du workflow de contrôle à *while_body* et au point de terminaison de *stmt*.

- Si *v* a l'état « définitivement assigné après une expression true » à la fin de *expr*, il est définitivement affecté au transfert du workflow de contrôle à *while_body*, mais il n'est pas définitivement assigné au point de terminaison de *stmt*.
- Si *v* a l'état « définitivement assigné après une expression false » à la fin de *expr*, il est définitivement affecté sur le transfert du workflow de contrôle au point de terminaison de *stmt*, mais n'est pas définitivement assigné sur le transfert du workflow de contrôle à *while_body*.

Instructions do

Pour une `do` instruction *stmt* de la forme :

```
C#

do do_body while ( expr ) ;
```

- *v* a le même état d'assignation définie sur le transfert du workflow de contrôle entre le début de *stmt* et *do_body* qu'au début de *stmt*.
- *v* a le même état d'assignation définie au début de *expr* qu'au point de terminaison de *do_body*.
- Si *v* est définitivement assignée à la fin de *expr*, elle est définitivement assignée sur le transfert du workflow de contrôle vers le point de terminaison de *stmt*.
- Si *v* a l'état « définitivement assigné après une expression false » à la fin de *expr*, il est définitivement affecté sur le transfert du workflow de contrôle au point de terminaison de *stmt*.

Pour les instructions

Vérification de l'assignation définie pour une `for` instruction de la forme :

```
C#

for ( for_initializer ; for_condition ; for_iterator ) embedded_statement
```

est effectué comme si l'instruction était écrite :

```
C#

{
    for_initializer ;
    while ( for_condition ) {
        embedded_statement ;
        for_iterator ;
```

```
    }
}
```

Si le *for_condition* est omis de l' `for` instruction, l'évaluation de l'assignation définie se poursuit comme si *for_condition* était remplacé par `true` dans l'expansion ci-dessus.

Instructions break, continue et GOTO

L'état d'assignation définie de *v* sur le transfert de workflow de contrôle provoqué par une `break` `continue` instruction, ou `goto` est le même que l'état d'assignation définie de *v* au début de l'instruction.

Instructions throw

Pour une instruction *stmt* au format

```
C#

throw expr ;
```

L'état d'assignation définie de *v* au début de *expr* est le même que l'état d'assignation définie de *v* au début de *stmt*.

Instructions Return

Pour une instruction *stmt* au format

```
C#

return expr ;
```

- L'état d'assignation définie de *v* au début de *expr* est le même que l'état d'assignation définie de *v* au début de *stmt*.
- Si *v* est un paramètre de sortie, il doit être affecté définitivement :
  - après *expr*
  - ou à la fin du `finally` bloc d'un `try` - `finally` ou `try` - `catch` - `finally` qui englobe l' `return` instruction.

Pour une instruction stmt de la forme :

```
C#
```

```
return ;
```

- Si *v* est un paramètre de sortie, il doit être affecté définitivement :
  - avant *stmt*
  - ou à la fin du `finally` bloc d'un `try` - `finally` ou `try` - `catch` - `finally` qui englobe l' `return` instruction.

Instructions Try-Catch

Pour une instruction *stmt* de la forme :

```
C#

try try_block
catch(...) catch_block_1
...
catch(...) catch_block_n
```

- L'état d'assignation définie de *v* au début de *try_block* est identique à l'état d'assignation définie de *v* au début de *stmt*.
- L'état d'assignation définie de *v* au début de *catch_block_i* (pour Any *i*) est identique à l'état d'assignation définie de *v* au début de *stmt*.
- L'état d'assignation définie de *v* à l'extrémité de *stmt* est définitivement affecté si (et seulement si) *v* est définitivement assigné au point de terminaison de *try_block* et chaque *catch_block_i* (pour chaque *i* de 1 à *n*).

Instructions try-finally

Pour une `try` instruction *stmt* de la forme :

```
C#

try try_block finally finally_block
```

- L'état d'assignation définie de *v* au début de *try_block* est identique à l'état d'assignation définie de *v* au début de *stmt*.
- L'état d'assignation définie de *v* au début de *finally_block* est identique à l'état d'assignation définie de *v* au début de *stmt*.
- L'état d'assignation définie de *v* à l'extrémité de *stmt* est définitivement affecté si (et seulement si) au moins l'une des conditions suivantes est vraie :
  - *v* est définitivement assignée au point de terminaison de *try_block*

- *v* est définitivement assignée au point de terminaison de *finally_block*

Si un transfert de workflow de contrôle (par exemple, une `goto` instruction) commence dans *try_block*, et se termine en dehors de *try_block*, *v* est également considéré comme définitivement assigné sur ce transfert de workflow si *v* est définitivement assigné au point de terminaison de *finally_block*. (Il ne s'agit pas d'un seul si, si *v* est définitivement affecté pour une autre raison sur ce transfert de workflow, il est toujours considéré comme définitivement affecté.)

Instructions try-catch-finally

Analyse d'assignation définie pour une `try` - `catch` - `finally` instruction de la forme :

C#

```
try try_block
catch(...) catch_block_1
...
catch(...) catch_block_n
finally *finally_block*
```

est effectué comme si l'instruction était une `try` - `finally` instruction englobant une `try` - `catch` instruction :

C#

```
try {
    try try_block
    catch(...) catch_block_1
    ...
    catch(...) catch_block_n
}
finally finally_block
```

L'exemple suivant montre comment les différents blocs d'une `try` instruction (instruction try) affectent l'assignation définie.

C#

```
class A
{
    static void F() {
        int i, j;
        try {
            goto LABEL;
            // neither i nor j definitely assigned
```

```
            i = 1;
            // i definitely assigned
        }

        catch {
            // neither i nor j definitely assigned
            i = 3;
            // i definitely assigned
        }

        finally {
            // neither i nor j definitely assigned
            j = 5;
            // j definitely assigned
            }
        // i and j definitely assigned
        LABEL:;
        // j definitely assigned

    }
}
```

Instructions foreach

Pour une `foreach` instruction *stmt* de la forme :

```
C#

foreach ( type identifier in expr ) embedded_statement
```

- L'état d'assignation définie de *v* au début de *expr* est identique à l'état *v* au début de *stmt*.
- L'état d'assignation définie de *v* sur le transfert du workflow de contrôle vers *embedded_statement* ou vers le point de terminaison de *stmt* est identique à l'état *v* à la fin de *expr*.

Utilisation d'instructions

Pour une `using` instruction *stmt* de la forme :

```
C#

using ( resource_acquisition ) embedded_statement
```

- L'état d'assignation définie de *v* au début de *resource_acquisition* est identique à l'état *v* au début de *stmt*.

- L'état d'assignation définie de *v* sur le transfert du workflow de contrôle vers *embedded_statement* est identique à l'état *v* à la fin de *resource_acquisition*.

Instructions Lock

Pour une `lock` instruction *stmt* de la forme :

```
C#

lock ( expr ) embedded_statement
```

- L'état d'assignation définie de *v* au début de *expr* est identique à l'état *v* au début de *stmt*.
- L'état d'assignation définie de *v* sur le transfert du workflow de contrôle vers *embedded_statement* est identique à l'état *v* à la fin de *expr*.

Instructions yield

Pour une `yield return` instruction *stmt* de la forme :

```
C#

yield return expr ;
```

- L'état d'assignation définie de *v* au début de *expr* est identique à l'état *v* au début de *stmt*.
- L'état d'assignation définie de *v* à la fin de *stmt* est identique à l'état *v* à la fin de *expr*.
- Une `yield break` instruction n'a aucun effet sur l'état d'assignation définie.

Règles générales pour les expressions simples

La règle suivante s'applique à ces types d'expressions : littéraux (littéraux), noms simples (noms simples), expressions d'accès de membre (accès aux membres), expressions d'accès de base non indexées (accès de base), `typeof` expressions (opérateur typeof), expressions de valeur par défaut (expressions de valeur par défaut) et `nameof` expressions (expressions Nameof).

- L'état d'assignation définie de *v* à la fin d'une telle expression est le même que l'état d'assignation définie de *v* au début de l'expression.

Règles générales pour les expressions avec des expressions incorporées

Les règles suivantes s'appliquent à ces types d'expressions : les expressions entre parenthèses (expressions entre parenthèses), les expressions d'accès aux éléments (accès aux éléments), les expressions d'accès de base avec indexation (accès de base), les expressions de conversion d'incrémentationet de décrémentation(opérateurs d'incrémentationetde décrémentation de préfixeunaire `+` , `-` , `~` , `*` expressions, binaires,,,,, `+` `-` `*` `/` `%` `<<` , `>>` , `<` , `<=` , `>` , `>=` ,,, `==` `!=` `is` , `as` , `&` , `|` , `^` expressions (opérateurs arithmétiques, opérateurs de décalage, opérateurs relationnels et de test de type, opérateurs logiques), expressions d'assignation composée (assignation composée) `checked` et `unchecked` expressions (opérateurs activés et désactivés), ainsi quelesexpressions de création de tableau et de

Chacune de ces expressions a une ou plusieurs sous-expressions qui sont évaluées de manière non conditionnelle dans un ordre fixe. Par exemple, l' `%` opérateur binaire évalue le côté gauche de l'opérateur, puis le côté droit. Une opération d'indexation évalue l'expression indexée, puis évalue chacune des expressions d'index, dans l'ordre, de gauche à droite. Pour une expression *expr*, qui contient des sous-expressions *E1, E2,...,* en, évaluées dans cet ordre :

- L'état d'assignation définie de *v* au début de *E1* est le même que l'état d'assignation définie au début de *expr*.
- L'état d'assignation définie de *v* au début de l' *AE* (*i* supérieur à un) est identique à l'état d'assignation définie à la fin de la sous-expression précédente.
- L'état d'assignation définie de *v* à la fin de *expr* est identique à l'état d'assignation définie à la *fin de la*

Expressions d'appel et expressions de création d'objet

Pour une expression d'appel *expr* de la forme :

```
C#

primary_expression ( arg1 , arg2 , ... , argN )
```

ou une expression de création d'objet de la forme suivante :

```
C#

new type ( arg1 , arg2 , ... , argN )
```

- Pour une expression d'appel, l'état d'assignation définie de *v* avant *primary_expression* est identique à l'état *v* avant *expr*.
- Pour une expression d'appel, l'état d'assignation définie de *v* avant *Arg1* est le même que l'état de *v* après *primary_expression*.
- Pour une expression de création d'objet, l'état d'assignation définie de *v* avant *Arg1* est le même que l'état de *v* avant *expr*.
- Pour chaque argument *Argi*, l'état d'assignation définie de *v* après *Argi* est déterminé par les règles d'expression normales, en ignorant `ref` les `out` modificateurs ou.
- Pour chaque argument *Argi* pour un *i* supérieur à un, l'état d'assignation définie de *v* avant *Argi* est identique à l'état *v* après l' *argument précédent*.
- Si la variable *v* est passée comme `out` argument (c'est-à-dire un argument du formulaire `out v`) dans l'un des arguments, l'état *v* après *expr* est définitivement assigné. Dispose l'état de *v* après *expr* est identique à l'état *v* après *argN*.
- Pour les initialiseurs de tableau (expressions de création de tableau), les initialiseurs d'objets (initialiseurs d'objets), les initialiseurs de collection (initialiseurs de collection) et les initialiseurs d'objets anonymes (expressions de création d'objets anonymes), l'état d'assignation définie est déterminé par le développement défini en termes de.

Expressions d'assignation simples

Pour une expression *expr* de la forme `w = expr_rhs` :

- L'état d'assignation définie de *v* avant *expr_rhs* est identique à l'état d'assignation définie de *v* avant *expr*.
- L'état d'assignation définie de *v* après *expr* est déterminé par :
  - Si *w* est la même variable que *v*, l'état d'assignation définie de *v* après *expr* est définitivement assigné.
  - Sinon, si l'assignation se produit dans le constructeur d'instance d'un type struct, si *w* est un accès à la propriété désignant une propriété implémentée automatiquement *p* sur l'instance en cours de construction et que *v* est le champ de stockage masqué de *P*, l'état d'assignation définie de *v* après *expr* est définitivement assigné.
  - Sinon, l'état d'assignation définie de *v* après *expr* est le même que l'état d'assignation définie de *v* après *expr_rhs*.

Expressions && (conditionnelles et)

Pour une expression *expr* de la forme `expr_first && expr_second` :

- L'état d'assignation définie de *v* avant *expr_first* est identique à l'état d'assignation définie de *v* avant *expr*.
- L'état d'assignation définie de *v* avant *expr_second* est définitivement assigné si l'état de *v* après *expr_first* est définitivement assigné ou « définitivement assigné après l'expression true ». Dans le cas contraire, elle n'est pas définitivement assignée.
- L'état d'assignation définie de *v* après *expr* est déterminé par :
  - Si *expr_first* est une expression constante avec la valeur `false` , l'état d'assignation définie de *v* après *expr* est le même que l'état d'assignation définie de *v* après *expr_first*.
  - Dans le cas contraire, si l'état de *v* après *expr_first* est définitivement affecté, l'état de *v* après *expr* est définitivement assigné.
  - Dans le cas contraire, si l'état de *v* après *expr_second* est assignée définitivement et que l'état de *v* après *expr_first* est « définitivement assigné après false expression », alors l'état de *v* après *expr* est définitivement assigné.
  - Dans le cas contraire, si l'état de *v* après *expr_second* est définitivement assigné ou « définitivement assigné après une expression true », l'état *v* après *expr* est « définitivement affecté après une expression true ».
  - Sinon, si l'état de *v* après *expr_first* est « définitivement assigné après une expression false » et que l'état de *v* après *expr_second* est « définitivement assigné après une expression false », l'état de *v* après *expr* est « définitivement assigné après false expression ».
  - Dans le cas contraire, l'état de *v* après *expr* n'est pas définitivement assigné.

Dans l'exemple

C#

```
class A
{
    static void F(int x, int y) {
        int i;
        if (x >= 0 && (i = y) >= 0) {
            // i definitely assigned
        }
        else {
            // i not definitely assigned
        }
        // i not definitely assigned
    }
}
```

la variable `i` est considérée comme définitivement assignée dans l'une des instructions incorporées d'une `if` instruction, mais pas dans l'autre. Dans l' `if` instruction de la

méthode `F` , la variable `i` est définitivement assignée dans la première instruction incorporée, car l'exécution de l'expression `(i = y)` précède toujours l'exécution de cette instruction incorporée. En revanche, la variable `i` n'est pas définitivement assignée dans la deuxième instruction incorporée, car `x >= 0` peut avoir testé false, ce qui entraîne la non-attribution de la variable `i` .

|| expressions (conditionnelles ou)

Pour une expression *expr* de la forme `expr_first || expr_second` :

- L'état d'assignation définie de *v* avant *expr_first* est identique à l'état d'assignation définie de *v* avant *expr*.
- L'état d'assignation définie de *v* avant *expr_second* est définitivement assigné si l'état de *v* après *expr_first* est définitivement assigné ou « définitivement assigné après une expression false ». Dans le cas contraire, elle n'est pas définitivement assignée.
- L'instruction d'assignation définie de *v* après *expr* est déterminée par :
  - Si *expr_first* est une expression constante avec la valeur `true` , l'état d'assignation définie de *v* après *expr* est le même que l'état d'assignation définie de *v* après *expr_first*.
  - Dans le cas contraire, si l'état de *v* après *expr_first* est définitivement affecté, l'état de *v* après *expr* est définitivement assigné.
  - Dans le cas contraire, si l'état de *v* après *expr_second* est assignée définitivement, et que l'état de *v* après *expr_first* est « définitivement assigné après une expression true », l'état de *v* après *expr* est définitivement assigné.
  - Dans le cas contraire, si l'état de *v* après *expr_second* est définitivement assigné ou « définitivement assigné après une expression false », l'état *v* après *expr* est « définitivement affecté après une expression false ».
  - Sinon, si l'état de *v* après *expr_first* est « définitivement assigné après une expression true » et que l'état de *v* après *expr_second* est « définitivement assigné après une expression true », l'état de *v* après *expr* est « définitivement assigné après l'expression true ».
  - Dans le cas contraire, l'état de *v* après *expr* n'est pas définitivement assigné.

Dans l'exemple

C#

```
class A
{
    static void G(int x, int y) {
        int i;
```

```
        if (x >= 0 || (i = y) >= 0) {
            // i not definitely assigned
        }
        else {
            // i definitely assigned
        }
        // i not definitely assigned
    }
}
```

la variable `i` est considérée comme définitivement assignée dans l'une des instructions incorporées d'une `if` instruction, mais pas dans l'autre. Dans l' `if` instruction de la méthode `G` , la variable `i` est définitivement assignée dans la deuxième instruction incorporée, car l'exécution de l'expression `(i = y)` précède toujours l'exécution de cette instruction incorporée. En revanche, la variable `i` n'est pas définitivement assignée dans la première instruction incorporée, car `x >= 0` peut avoir testé la valeur true, ce qui a pour effet que la variable n'est pas `i` assignée.

! expressions (négation logique)

Pour une expression *expr* de la forme `! expr_operand` :

- L'état d'assignation définie de *v* avant *expr_operand* est identique à l'état d'assignation définie de *v* avant *expr*.
- L'état d'assignation définie de *v* après *expr* est déterminé par :
  - Si l'état *v* après * expr_operand * est définitivement affecté, l'état *v* après *expr* est définitivement assigné.
  - Si l'état de *v* après * expr_operand * n'est pas définitivement assigné, l'état de *v* après *expr* n'est pas définitivement assigné.
  - Si l'état *v* après * expr_operand * est « définitivement assigné après une expression false », alors l'état de *v* après *expr* est « définitivement assigné après l'expression true ».
  - Si l'état de *v* après * expr_operand * est « définitivement assigné après l'expression true », alors l'état de *v* après *expr* est « définitivement assigné après une expression false ».

?? expressions (fusion de valeurs null)

Pour une expression *expr* de la forme `expr_first ?? expr_second` :

- L'état d'assignation définie de *v* avant *expr_first* est identique à l'état d'assignation définie de *v* avant *expr*.

- L'état d'assignation définie de *v* avant *expr_second* est identique à l'état d'assignation définie de *v* après *expr_first*.
- L'instruction d'assignation définie de *v* après *expr* est déterminée par :
  - Si *expr_first* est une expression constante (expressions constantes) avec la valeur null, l'état *v* après *expr* est identique à l'état *v* après *expr_second*.
- Dans le cas contraire, l'état de *v* après *expr* est le même que l'état d'assignation définie de *v* après *expr_first*.

?: (conditionnel), expressions

Pour une expression *expr* de la forme `expr_cond ? expr_true : expr_false` :

- L'état d'assignation définie de *v* avant *expr_cond* est identique à l'état *v* avant *expr*.
- L'état d'assignation définie de *v* avant *expr_true* n'est jamais assigné si et seulement si l'un des éléments suivants est présent :
  - *expr_cond* est une expression constante avec la valeur `false`
  - l'état de *v* après *expr_cond* est définitivement assigné ou « définitivement assigné après l'expression true ».
- L'état d'assignation définie de *v* avant *expr_false* n'est jamais assigné si et seulement si l'un des éléments suivants est présent :
  - *expr_cond* est une expression constante avec la valeur `true`
  - l'état de *v* après *expr_cond* est définitivement assigné ou « définitivement assigné après une expression false ».
- L'état d'assignation définie de *v* après *expr* est déterminé par :
  - Si *expr_cond* est une expression constante (expressions constantes) avec la valeur `true` , l'état *v* après *expr* est identique à l'état *v* après *expr_true*.
  - Sinon, si *expr_cond* est une expression constante (expressions constantes) avec la valeur, `false` l'état de *v* après *expr* est identique à l'état *v* après *expr_false*.
  - Sinon, si l'état de *v* après *expr_true* est assignée définitivement et que l'état de *v* après *expr_false* est définitivement assigné, l'état *v* après *expr* est définitivement assigné.
  - Dans le cas contraire, l'état de *v* après *expr* n'est pas définitivement assigné.

Fonctions anonymes

Pour un *lambda_expression* ou *anonymous_method_expression expr* avec *un corps (* de type *bloc* ou *expression*) :

- L'état d'assignation définie d'une variable externe *v* avant le *corps* est identique à l'état *v* avant *expr*. Autrement dit, l'état d'assignation définie des variables externes est hérité du contexte de la fonction anonyme.

- L'état d'assignation définie d'une variable externe *v* après *expr* est identique à l'état *v* avant *expr*.

L'exemple

```
C#

delegate bool Filter(int i);

void F() {
    int max;

    // Error, max is not definitely assigned
    Filter f = (int n) => n < max;

    max = 5;
    DoWork(f);
}
```

génère une erreur au moment de la compilation, car `max` n'est pas définitivement assignée où la fonction anonyme est déclarée. L'exemple

```
C#

delegate void D();

void F() {
    int n;
    D d = () => { n = 1; };

    d();

    // Error, n is not definitely assigned
    Console.WriteLine(n);
}
```

génère également une erreur au moment de la compilation, car l'assignation à `n` dans la fonction anonyme n'a aucun effet sur l'état d'assignation définie de `n` l'extérieur de la fonction anonyme.

Références de variables

Une *variable_reference* est une *expression* classée en tant que variable. Un *variable_reference* désigne un emplacement de stockage qui est accessible à la fois pour extraire la valeur actuelle et pour stocker une nouvelle valeur.

```
```

antlr

```
variable_reference
    : expression
    ;
```

En C et C++, une *variable_reference* est appelée *lvalue*.

Atomicité des références de variable

Les lectures et les écritures des types de données suivants sont des types de référence Atomic : `bool` ,,,, `char` `byte` `sbyte` `short` , `ushort` ,,, `uint` `int` `float` et. En outre, les lectures et écritures de types ENUM avec un type sous-jacent dans la liste précédente sont également atomiques. Les lectures et écritures d'autres types, notamment,,, `long` `ulong` `double` et `decimal` , ainsi que les types définis par l'utilisateur, ne sont pas nécessairement atomiques. Outre les fonctions de bibliothèque conçues à cet effet, il n'existe aucune garantie de lecture-modification/écriture atomique, comme dans le cas de l'incrémentation ou de la décrémentation.

Conversions

Article • 13/05/2021

Une *__conversion__ _ permet à une expression d'être traitée comme étant d'un type particulier. Une conversion peut entraîner le traitement d'une expression d'un type donné comme ayant un type différent, ou elle peut provoquer l'utilisation d'une expression sans type pour obtenir un type. Les conversions peuvent être *implicites* ou _ *Explicit* *, ce qui détermine si un cast explicite est nécessaire. Par exemple, la conversion du type `int` en type `long` est implicite, de sorte que les expressions de type `int` peuvent implicitement être traitées comme type `long` . La conversion inverse, du type `long` en type `int` , est explicite et un cast explicite est donc nécessaire.

C#

```
int a = 123;
long b = a;          // implicit conversion from int to long
int c = (int) b;     // explicit conversion from long to int
```

Certaines conversions sont définies par le langage. Les programmes peuvent également définir leurs propres conversions (conversions définies par l'utilisateur).

Conversions implicites

Les conversions suivantes sont classées comme conversions implicites :

- Conversions d'identité
- Conversions numériques implicites
- Conversions d'énumération implicites
- Conversions de chaînes interpolées implicites
- Conversions Nullable implicites
- Conversions de littéral null
- Conversions de références implicites
- Conversions boxing
- Conversions dynamiques implicites
- Conversions d'expressions constantes implicites
- Conversions implicites définies par l'utilisateur
- Conversions de fonctions anonymes
- Conversions de groupes de méthodes

Les conversions implicites peuvent se produire dans diverses situations, y compris les appels de membresde fonctions (vérification au moment de la compilation de la

résolution de surcharge dynamique), les expressions de cast (expressions de cast) et les assignations (opérateurs d'assignation).

Les conversions implicites prédéfinies aboutissent toujours et n'entraînent jamais la levée d'exceptions. Les conversions implicites définies par l'utilisateur doivent également présenter ces caractéristiques.

À des fins de conversion, les types `object` et `dynamic` sont considérés comme équivalents.

Toutefois, les conversions dynamiques (conversionsdynamiques implicites et conversions dynamiques explicites) s'appliquent uniquement aux expressions de type `dynamic` (type dynamique).

Conversion d'identité

Une conversion d'identité est convertie de n'importe quel type vers le même type. Cette conversion existe de telle sorte qu'une entité qui a déjà un type requis peut être considérée comme convertible en ce type.

- Étant donné que `object` et `dynamic` sont considérés comme équivalents, il existe une conversion d'identité entre `object` et `dynamic` , et entre les types construits qui sont identiques lors du remplacement de toutes les occurrences de `dynamic` par `object` .

Conversions numériques implicites

Les conversions numériques implicites sont les suivantes :

- De `sbyte` à `short` ,,,,, `int` `long` `float` `double` ou `decimal` .
- De `byte` à `short` ,,,,, `ushort` `int` `uint` `long` `ulong` , `float` , `double` , ou `decimal` .
- De `short` à `int` , `long` ,,, `float` `double` ou `decimal` .
- De `ushort` à `int` ,,,,, `uint` `long` `ulong` `float` `double` , ou `decimal` .
- De `int` à `long` , `float` , `double` ou `decimal` .
- De `uint` à `long` , `ulong` ,,, `float` `double` ou `decimal` .
- De `long` à `float` , `double` ou `decimal` .
- De `ulong` à `float` , `double` ou `decimal` .
- De `char` à,,,,,, `ushort` `int` `uint` `long` `ulong` `float` `double` , ou `decimal` .
- De `float` à `double`

Les conversions de `int` ,, `uint` `long` ou `ulong` vers `float` et depuis `long` ou `ulong` vers `double` peuvent entraîner une perte de précision, mais n'entraînent jamais de perte de magnitude. Les autres conversions numériques implicites ne perdent jamais d'informations.

Il n'y a pas de conversion implicite vers le `char` type, donc les valeurs des autres types intégraux ne sont pas automatiquement converties vers le `char` type.

Conversions d'énumération implicites

Une conversion d'énumération implicite permet à l' *decimal_integer_literal* `0` d'être convertie en n'importe quel *enum_type* et en n'importe quel *nullable_type* dont le type sous-jacent est un *enum_type*. Dans ce dernier cas, la conversion est évaluée en convertissant les *enum_type* sous-jacentes et en encapsulant le résultat (types Nullable).

Conversions de chaînes interpolées implicites

Une conversion de chaîne interpolée implicite permet à une *interpolated_string_expression* (chaînes interpolées) d'être converties en `System.IFormattable` ou `System.FormattableString` (qui implémente `System.IFormattable`).

Lorsque cette conversion est appliquée, une valeur de chaîne n'est pas composée de la chaîne interpolée. Au lieu de cela, une instance de `System.FormattableString` est créée, comme décrit plus loin dans chaînes interpolées.

Conversions Nullable implicites

Les conversions implicites prédéfinies qui fonctionnent sur des types valeur non Nullable peuvent également être utilisées avec des formes Nullable de ces types. Pour chacune des conversions de valeurs et d'identités implicites prédéfinies qui convertissent un type valeur n'acceptant pas les valeurs null `S` en type valeur n'acceptant pas les valeurs null `T` , les conversions Nullable implicites suivantes existent :

- Conversion implicite de `S?` en `T?` .
- Conversion implicite de `S` en `T?` .

L'évaluation d'une conversion Nullable implicite basée sur une conversion sous-jacente de `S` en `T` continue comme suit :

- Si la conversion de valeurs NULL est comprise entre `S?` et `T?` :
  - Si la valeur source est null (la `HasValue` propriété a la valeur false), le résultat est la valeur null de type `T?` .
  - Sinon, la conversion est évaluée en tant que désencapsulation de `S?` à `S` , suivi de la conversion sous-jacente de `S` en `T` , suivie d'un encapsulation (types Nullable) de `T` à `T?` .

- Si la conversion Nullable est comprise entre et `S` `T?` , la conversion est évaluée comme la conversion sous-jacente de `S` en, suivie d' `T` une encapsulation de `T` à `T?` .

Conversions de littéral null

Une conversion implicite existe du `null` littéral en n'importe quel type Nullable. Cette conversion produit la valeur null (types Nullable) du type Nullable donné.

Conversions de références implicites

Les conversions de référence implicites sont les suivantes :

- Depuis n'importe quel *reference_type* vers `object` et `dynamic` .
- De tout *class_type* `S` à n'importe quel *class_type* `T` , fourni `S` est dérivé de `T` .
- À partir de n'importe quel *class_type* `S` à n'importe quel *INTERFACE_TYPE* `T` , `S` implémente `T` .
- De tout *INTERFACE_TYPE* `S` à n'importe quel *INTERFACE_TYPE* `T` , fourni `S` est dérivé de `T` .
- D'un *array_type* `S` avec un type `SE` d'élément à un *array_type* `T` avec un type d'élément `TE` , à condition que toutes les conditions suivantes soient vraies :
  - `S` et `T` diffèrent uniquement par le type d'élément. En d'autres termes, `S` et `T` ont le même nombre de dimensions.
  - `SE`Et `TE` sont *reference_type* s.
  - Une conversion de référence implicite existe entre `SE` et `TE` .
- Depuis n'importe quel *array_type* vers `System.Array` et les interfaces qu'il implémente.
- D'un type tableau unidimensionnel `S[]` vers `System.Collections.Generic.IList<T>` et ses interfaces de base, à condition qu'il existe une conversion d'identité ou de référence implicite de `S` en `T` .
- Depuis n'importe quel *delegate_type* vers `System.Delegate` et les interfaces qu'il implémente.

- Du littéral null à n'importe quel *reference_type*.
- De tout *reference_type* à un *reference_type* `T` s'il a une conversion d'identité ou de référence implicite en *reference_type* `T0` et `T0` a une conversion d'identité en `T` .
- De tout *reference_type* à un type d'interface ou de délégué `T` s'il a une conversion d'identité ou de référence implicite en une interface ou un type délégué `T0` et que la `T0` variance est convertible (conversion de variance) en `T` .
- Conversions implicites impliquant des paramètres de type connus comme des types référence. Pour plus d'informations sur les conversions implicites impliquant des paramètres de type, consultez conversions implicites impliquant des paramètres de type.

Les conversions de référence implicites sont les conversions entre *reference_type* s qui peuvent être prouvées de toujours s'exécuter correctement, et ne nécessitent donc pas de vérification au moment de l'exécution.

Les conversions de référence, implicites ou explicites, ne modifient jamais l'identité référentielle de l'objet en cours de conversion. En d'autres termes, si une conversion de référence peut modifier le type de la référence, elle ne modifie jamais le type ou la valeur de l'objet auquel il est fait référence.

Conversions boxing

Une conversion boxing permet à un *Value_type* d'être implicitement converti en un type référence. Une conversion boxing existe de tout *non_nullable_value_type* vers `object` et `dynamic` , vers `System.ValueType` et vers n'importe quel *INTERFACE_TYPE* implémenté par le *non_nullable_value_type*. En outre, un *enum_type* peut être converti en type `System.Enum` .

Une conversion boxing existe d'un *nullable_type* à un type référence, si et seulement s'il existe une conversion boxing du *non_nullable_value_type* sous-jacent au type référence.

Un type valeur a une conversion boxing en un type interface `I` s'il a une conversion boxing en un type interface `I0` et `I0` une conversion d'identité en `I` .

Un type valeur a une conversion boxing en un type interface `I` s'il a une conversion boxing en type interface ou délégué `I0` et que la `I0` variance est convertible (conversion d'écart) en `I` .

Le boxing d'une valeur d'une *non_nullable_value_type* consiste à allouer une instance d'objet et à copier la valeur *Value_type* dans cette instance. Une structure peut être convertie dans le type `System.ValueType` , car il s'agit d'une classe de base pour tous les structs (héritage).

Le boxing d'une valeur d'un *nullable_type* se déroule comme suit :

- Si la valeur source est null (la `HasValue` propriété a la valeur false), le résultat est une référence null du type cible.
- Dans le cas contraire, le résultat est une référence à un boxed `T` produit en désencapsulant et en Boxing la valeur source.

Les conversions boxing sont décrites plus en détail dans conversions boxing.

Conversions dynamiques implicites

Une conversion dynamique implicite existe à partir d'une expression de type `dynamic` vers n'importe quel type `T` . La conversion est liée dynamiquement (liaison dynamique), ce qui signifie qu'une conversion implicite est recherchée au moment de l'exécution à partir du type d'exécution de l'expression à `T` . Si aucune conversion n'est trouvée, une exception Runtime est levée.

Notez que cette conversion implicite n'enfreint apparemment pas les conseils au début des conversions implicites qu'une conversion implicite ne doit jamais provoquer une exception. Toutefois, il ne s'agit pas de la conversion elle-même, mais de la *détection* de la conversion à l'origine de l'exception. Le risque d'exceptions au moment de l'exécution est inhérent à l'utilisation de la liaison dynamique. Si la liaison dynamique de la conversion n'est pas souhaitée, l'expression peut être d'abord convertie en `object` , puis en type souhaité.

L'exemple suivant illustre des conversions dynamiques implicites :

C#

```
object o  = "object"
dynamic d = "dynamic";

string s1 = o; // Fails at compile-time -- no conversion exists
string s2 = d; // Compiles and succeeds at run-time
int i     = d; // Compiles but fails at run-time -- no conversion exists
```

Les assignations à `s2` et `i` utilisent les conversions dynamiques implicites, où la liaison des opérations est suspendue jusqu'au moment de l'exécution. Au moment de l'exécution, les conversions implicites sont recherchées à partir du type au moment de l'exécution de `d` -- `string` --vers le type cible. Une conversion a été détectée à `string` , mais pas à `int` .

Conversions d'expressions constantes implicites

Une conversion d'expression constante implicite autorise les conversions suivantes :

- Une *constant_expression* (expressions constantes) de type `int` peut être convertie en type `sbyte` ,,,, `byte` `short` `ushort` `uint` ou, à `ulong` condition que la valeur de la *constant_expression* soit comprise dans la plage du type de destination.
- Une *constant_expression* de type `long` peut être convertie en type `ulong` , à condition que la valeur de la *constant_expression* ne soit pas négative.

Conversions implicites impliquant des paramètres de type

Les conversions implicites suivantes existent pour un paramètre de type donné `T` :

- De à `T` sa classe de base effective `C` , de `T` à toute classe de base de `C` , et de `T` à toute interface implémentée par `C` . Au moment de l'exécution, si `T` est un type valeur, la conversion est exécutée en tant que conversion boxing. Dans le cas contraire, la conversion est exécutée en tant que conversion de référence implicite ou conversion d'identité.
- De `T` à un type d'interface `I` dans `T` l'ensemble d'interfaces effectif de et de `T` à n'importe quelle interface de base de `I` . Au moment de l'exécution, si `T` est un type valeur, la conversion est exécutée en tant que conversion boxing. Dans le cas contraire, la conversion est exécutée en tant que conversion de référence implicite ou conversion d'identité.
- De `T` à un paramètre de type `U` , fourni `T` dépend de `U` (contraintes de paramètre detype). Au moment de l'exécution, si `U` est un type valeur, `T` et `U` sont nécessairement du même type et aucune conversion n'est effectuée. Sinon, si `T` est un type valeur, la conversion est exécutée en tant que conversion boxing. Dans le cas contraire, la conversion est exécutée en tant que conversion de référence implicite ou conversion d'identité.
- Du littéral null à `T` , fourni `T` est un type référence.
- De `T` vers un type référence `I` s'il a une conversion implicite en un type référence `S0` et `S0` a une conversion d'identité en `S` . Au moment de l'exécution, la conversion est exécutée de la même façon que la conversion en `S0` .
- De `T` à un type d'interface `I` s'il a une conversion implicite en une interface ou un type délégué `I0` et que la `I0` variance est convertible en `I` (conversion d'écart). Au moment de l'exécution, si `T` est un type valeur, la conversion est exécutée en tant que conversion boxing. Dans le cas contraire, la conversion est exécutée en tant que conversion de référence implicite ou conversion d'identité.

Si `T` est connu comme un type référence (contraintes de paramètre de type), les conversions ci-dessus sont toutes classées comme conversions de référence implicites

(conversions deréférence implicites). Si `T` le type de référence n'est pas connu, les conversions ci-dessus sont classées comme conversions boxing (conversionsboxing).

Conversions implicites définies par l'utilisateur

Une conversion implicite définie par l'utilisateur se compose d'une conversion implicite standard facultative, suivie de l'exécution d'un opérateur de conversion implicite défini par l'utilisateur, suivie d'une autre conversion implicite standard facultative. Les règles exactes d'évaluation des conversions implicites définies par l'utilisateur sont décrites dans traitement des conversions implicites définies par l'utilisateur.

Conversions de fonctions anonymes et conversions de groupes de méthodes

Les fonctions anonymes et les groupes de méthodes n'ont pas de types dans et à eux-mêmes, mais peuvent être implicitement convertis en types délégués ou en types d'arborescences d'expressions. Les conversions de fonctions anonymes sont décrites plus en détail dans conversions de fonctions anonymes et conversions de groupes de méthodes dans les conversions de groupe de méthodes.

Conversions explicites

Les conversions suivantes sont classées comme des conversions explicites :

- Toutes les conversions implicites.
- Conversions numériques explicites.
- Conversions d'énumération explicites.
- Conversions Nullable explicites.
- Conversions de référence explicites.
- Conversions d'interface explicites.
- Conversions unboxing.
- Conversions dynamiques explicites
- Conversions explicites définies par l'utilisateur.

Les conversions explicites peuvent se produire dans les expressions de cast (expressions de cast).

L'ensemble de conversions explicites comprend toutes les conversions implicites. Cela signifie que les expressions de cast redondantes sont autorisées.

Les conversions explicites qui ne sont pas des conversions implicites sont des conversions qui ne peuvent pas être prouvées comme ayant échoué, des conversions qui sont susceptibles de perdre des informations et des conversions entre des domaines de types suffisamment différentes pour mériter une notation explicite.

Conversions numériques explicites

Les conversions numériques explicites sont les conversions d'un *numeric_type* à un autre *numeric_type* pour lequel une conversion numérique implicite (conversions numériques implicites) n'existe pas encore :

- De `sbyte` à `byte` , `ushort` ,,, `uint` `ulong` ou `char` .
- De `byte` à `sbyte` et `char` .
- De `short` à `sbyte` ,,,,, `byte` `ushort` `uint` `ulong` ou `char` .
- De `ushort` à `sbyte` , `byte` , `short` ou `char` .
- De `int` à `sbyte` ,,,,, `byte` `short` `ushort` `uint` `ulong` , ou `char` .
- De `uint` à `sbyte` ,,,,, `byte` `short` `ushort` `int` ou `char` .
- De `long` à,,,,,, `sbyte` `byte` `short` `ushort` `int` `uint` `ulong` , ou `char` .
- De `ulong` à,,,,,, `sbyte` `byte` `short` `ushort` `int` `uint` `long` , ou `char` .
- De `char` à `sbyte` , `byte` ou `short` .
- De `float` à `sbyte` ,,,,, `byte` `short` `ushort` `int` `uint` , `long` , `ulong` , `char` , ou `decimal` .
- De `double` à `sbyte` ,,,,, `byte` `short` `ushort` `int` `uint` , `long` , `ulong` , `char` , `float` , ou `decimal` .
- De `decimal` à `sbyte` ,,,,, `byte` `short` `ushort` `int` `uint` , `long` , `ulong` , `char` , `float` , ou `double` .

Étant donné que les conversions explicites incluent toutes les conversions numériques implicites et explicites, il est toujours possible de convertir des *numeric_type* en tout autre *numeric_type* à l'aide d'une expression de cast (expressions de cast).

Les conversions numériques explicites peuvent éventuellement perdre des informations ou provoquer des exceptions. Une conversion numérique explicite est traitée comme suit :

- Pour une conversion d'un type intégral vers un autre type intégral, le traitement dépend du contexte de vérification de dépassement de capacité (opérateurs activés et désactivés) dans lequel la conversion a lieu :
  - Dans un `checked` contexte, la conversion s'effectue correctement si la valeur de l'opérande source est comprise dans la plage du type de destination, mais lève

une `System.OverflowException` si la valeur de l'opérande source est en dehors de la plage du type de destination.
    - Dans un `unchecked` contexte, la conversion est toujours réussie et continue comme suit.
        - Si le type source est plus grand que le type de destination, la valeur source est tronquée via l'abandon de ses bits « supplémentaires » les plus significatifs. Le résultat est ensuite traité en tant que valeur du type de destination.
        - Si le type source est plus petit que le type de destination, la valeur source fait l'objet d'une extension de signe ou d'une extension de zéros, pour avoir la même taille que le type de destination. L'extension de signe est utilisée si le type source est signé. L'extension de zéro est utilisée si le type source est non signé. Le résultat est ensuite traité en tant que valeur du type de destination.
        - Si le type source a la même taille que le type de destination, la valeur source est traitée comme une valeur du type de destination.
- Pour une conversion de `decimal` en type intégral, la valeur source est arrondie vers zéro à la valeur intégrale la plus proche, et cette valeur intégrale devient le résultat de la conversion. Si la valeur intégrale obtenue est en dehors de la plage du type de destination, une `System.OverflowException` est levée.
- Pour une conversion à partir de `float` ou `double` en un type intégral, le traitement dépend du contexte de vérification de dépassement de capacité (opérateurs activés et désactivés) dans lequel la conversion a lieu :
    - Dans un `checked` contexte, la conversion se poursuit comme suit :
        - Si la valeur de l'opérande est NaN ou infinie, une `System.OverflowException` exception est levée.
        - Dans le cas contraire, l'opérande source est arrondi vers la valeur intégrale la plus proche de zéro. Si cette valeur intégrale est comprise dans la plage du type de destination, cette valeur est le résultat de la conversion.
        - Sinon, une exception `System.OverflowException` est levée.
    - Dans un `unchecked` contexte, la conversion est toujours réussie et continue comme suit.
        - Si la valeur de l'opérande est NaN ou infinie, le résultat de la conversion est une valeur non spécifiée du type de destination.
        - Dans le cas contraire, l'opérande source est arrondi vers la valeur intégrale la plus proche de zéro. Si cette valeur intégrale est comprise dans la plage du type de destination, cette valeur est le résultat de la conversion.
        - Dans le cas contraire, le résultat de la conversion est une valeur non spécifiée du type de destination.

- Pour une conversion de `double` en `float` , la `double` valeur est arrondie à la valeur la plus proche `float` . Si la `double` valeur est trop petite pour être représentée en tant que `float` , le résultat devient un zéro positif ou un zéro négatif. Si la `double` valeur est trop grande pour être représentée en tant que `float` , le résultat devient une infini positif ou négatif. Si la `double` valeur est NaN, le résultat est également Nan.
- Pour une conversion à partir de `float` ou `double` vers `decimal` , la valeur source est convertie en `decimal` représentation et arrondie au nombre le plus proche après le 28 décimale si nécessaire (type décimal). Si la valeur source est trop petite pour être représentée en tant que `decimal` , le résultat devient zéro. Si la valeur source est NaN, Infinity ou trop grande pour être représentée en tant que `decimal` , une `System.OverflowException` est levée.
- Pour une conversion de `decimal` en `float` ou `double` , la `decimal` valeur est arrondie à la valeur ou la plus proche `double` `float` . Bien que cette conversion puisse perdre la précision, elle ne provoque jamais la levée d'une exception.

Conversions d'énumération explicites

Les conversions d'énumération explicites sont les suivantes :

- De `sbyte` ,,,,, `byte` `short` `ushort` `int` `uint` , `long` , `ulong` , `char` , `float` , `double` ou `decimal` à n'importe quel *enum_type*.
- De tout *enum_type* à `sbyte` , `byte` , `short` , `ushort` , `int` , `uint` , `long` , `ulong` , `char` , `float` , `double` ou `decimal` .
- De tout *enum_type* à n'importe quel autre *enum_type*.

Une conversion d'énumération explicite entre deux types est traitée en traitant tout *enum_type* participant en tant que type sous-jacent de ce *enum_type*, puis en effectuant une conversion numérique implicite ou explicite entre les types résultants. Par exemple, étant donné un *enum_type* `E` avec et le type sous-jacent de `int` , une conversion de `E` en `byte` est traitée comme une conversion numérique explicite (conversions numériques explicites) de `int` en `byte` , et une conversion de `byte` en `E` est traitée comme une conversion numérique implicite (conversions numériques implicites) de en `byte` `int` .

Conversions Nullable explicites

Les ***conversions Nullable explicites*** autorisent les conversions explicites prédéfinies qui fonctionnent sur les types valeur non Nullable pour être également utilisées avec les formes Nullable de ces types. Pour chacune des conversions explicites prédéfinies qui

convertissent un type valeur n'acceptant pas les valeurs null `S` en type valeur non Nullable (conversion d'identité, conversions `T` numériques implicites, conversions d' énumération implicites, conversions numériques expliciteset conversions d'énumération explicites), les conversions Nullable suivantes existent :

- Conversion explicite de `S?` en `T?` .
- Conversion explicite de `S` en `T?` .
- Conversion explicite de `S?` en `T` .

L'évaluation d'une conversion Nullable basée sur une conversion sous-jacente de `S` en `T` continue comme suit :

- Si la conversion de valeurs NULL est comprise entre `S?` et `T?` :
  - Si la valeur source est null (la `HasValue` propriété a la valeur false), le résultat est la valeur null de type `T?` .
  - Dans le cas contraire, la conversion est évaluée en tant que désencapsulation de `S?` à `S` , suivi de la conversion sous-jacente de `S` en `T` , suivie d'une encapsulation de `T` à `T?` .
- Si la conversion Nullable est comprise entre et `S` `T?` , la conversion est évaluée comme la conversion sous-jacente de `S` en, suivie d' `T` une encapsulation de `T` à `T?` .
- Si la conversion Nullable est comprise entre et `S?` `T` , la conversion est évaluée comme un désencapsulation de `S?` à, suivie de `S` la conversion sous-jacente de `S` en `T` .

Notez qu'une tentative de désencapsulation d'une valeur Nullable lève une exception si la valeur est `null` .

Conversions de références explicites

Les conversions de référence explicites sont les suivantes :

- De `object` et `dynamic` vers tout autre *reference_type*.
- De tout *class_type* `S` à n'importe quel *class_type* `T` , fourni `S` est une classe de base de `T` .
- De tout *class_type* `S` à n'importe quel *INTERFACE_TYPE* `T` , fourni `S` n'est pas sealed et fourni `S` n'implémente pas `T` .
- De tout *INTERFACE_TYPE* `S` à n'importe quel *class_type* `T` , fourni `T` n'est pas scellé ou n'est pas `T` implémenté `S` .

- De tout *INTERFACE_TYPE* `S` à n'importe quel *INTERFACE_TYPE* `T` , fourni `S` n'est pas dérivé de `T` .
- D'un *array_type* `S` avec un type `SE` d'élément à un *array_type* `T` avec un type d'élément `TE` , à condition que toutes les conditions suivantes soient vraies :
  - `S` et `T` diffèrent uniquement par le type d'élément. En d'autres termes, `S` et `T` ont le même nombre de dimensions.
  - `SE`Et `TE` sont *reference_type* s.
  - Une conversion de référence explicite existe entre `SE` et `TE` .
- À partir de `System.Array` et des interfaces qu'il implémente à n'importe quel *array_type*.
- D'un type tableau unidimensionnel `S[]` vers `System.Collections.Generic.IList<T>` et ses interfaces de base, à condition qu'il existe une conversion de référence explicite de `S` en `T` .
- Depuis `System.Collections.Generic.IList<S>` et ses interfaces de base vers un type de tableau unidimensionnel `T[]` , à condition qu'il existe une conversion d'identité ou de référence explicite de `S` en `T` .
- À partir de `System.Delegate` et des interfaces qu'il implémente à n'importe quel *delegate_type*.
- D'un type référence à un type référence `T` s'il a une conversion de référence explicite en un type référence `T0` et `T0` a une conversion d'identité `T` .
- D'un type référence à un type d'interface ou de délégué `T` s'il a une conversion de référence explicite en type interface ou délégué `T0` et `T0` est convertible en valeur `T` ou `T` est convertible en valeur `T0` (conversion d'écart).
- De `D<S1...Sn>` à `D<T1...Tn>` où `D<X1...Xn>` est un type délégué générique, `D<S1...Sn>` n'est pas compatible avec ou identique à `D<T1...Tn>` , et pour chaque paramètre `Xi` de type des `D` éléments suivants :
  - Si `Xi` est indifférent, `Si` est identique à `Ti` .
  - Si `Xi` est covariant, il existe une conversion d'identité ou de référence implicite ou explicite de `Si` en `Ti` .
  - Si `Xi` est contravariant, `Si` et `Ti` sont identiques ou les deux types de référence.
- Les conversions explicites impliquant des paramètres de type connus comme des types référence. Pour plus d'informations sur les conversions explicites impliquant des paramètres de type, consultez conversions explicites impliquant des paramètres de type.

Les conversions de référence explicites sont les conversions entre les types référence qui requièrent des vérifications à l'exécution pour s'assurer qu'elles sont correctes.

Pour qu'une conversion de référence explicite aboutisse au moment de l'exécution, la valeur de l'opérande source doit être `null` , ou le type réel de l'objet référencé par l'opérande source doit être un type qui peut être converti vers le type de destination par une conversion de référence implicite (conversions de référence implicites) ou une conversion boxing (conversions boxing). Si une conversion de référence explicite échoue, une `System.InvalidCastException` exception est levée.

Les conversions de référence, implicites ou explicites, ne modifient jamais l'identité référentielle de l'objet en cours de conversion. En d'autres termes, si une conversion de référence peut modifier le type de la référence, elle ne modifie jamais le type ou la valeur de l'objet auquel il est fait référence.

Conversions unboxing

Une conversion unboxing permet à un type de référence d'être converti explicitement en *Value_type*. Une conversion unboxing existe à partir des `object` types, `dynamic` et `System.ValueType` à n'importe quel *non_nullable_value_type*, et de n'importe quel *INTERFACE_TYPE* à n'importe quel *non_nullable_value_type* qui implémente le *INTERFACE_TYPE*. En outre `System.Enum` , le type peut être unboxed en tout *enum_type*.

Une conversion unboxing existe d'un type référence à un *nullable_type* s'il existe une conversion unboxing du type référence au *non_nullable_value_type* sous-jacent de l' *nullable_type*.

Un type valeur `S` a une conversion unboxing à partir d'un type interface `I` s'il a une conversion unboxing d'un type interface `I0` et `I0` a une conversion d'identité en `I` .

Un type valeur `S` a une conversion unboxing à partir d'un type interface `I` s'il a une conversion unboxing à partir d'une interface ou d'un type délégué `I0` , et `I0` est convertible en variance en `I` ou `I` est convertible en valeur `I0` (conversion d'écart).

Une opération d'unboxing consiste à vérifier en premier lieu que l'instance d'objet est une valeur boxed du *Value_type* donné, puis à copier la valeur hors de l'instance. La conversion unboxing d'une référence null en *nullable_type* produit la valeur null du *nullable_type*. Un struct peut être unboxed du type `System.ValueType` , car il s'agit d'une classe de base pour tous les structs (héritage).

Les conversions unboxing sont décrites plus en détail dans conversions unboxing.

Conversions dynamiques explicites

Une conversion dynamique explicite existe à partir d'une expression de type `dynamic` vers n'importe quel type `T` . La conversion est liée dynamiquement (liaison dynamique), ce qui signifie qu'une conversion explicite sera recherchée au moment de l'exécution à partir du type d'exécution de l'expression à `T` . Si aucune conversion n'est trouvée, une exception Runtime est levée.

Si la liaison dynamique de la conversion n'est pas souhaitée, l'expression peut être d'abord convertie en `object` , puis en type souhaité.

Supposons que la classe suivante soit définie :

C#

```
class C
{
    int i;

    public C(int i) { this.i = i; }

    public static explicit operator C(string s)
    {
        return new C(int.Parse(s));
    }
}
```

L'exemple suivant illustre des conversions dynamiques explicites :

C#

```
object o  = "1";
dynamic d = "2";

var c1 = (C)o; // Compiles, but explicit reference conversion fails
var c2 = (C)d; // Compiles and user defined conversion succeeds
```

La meilleure conversion de `o` en `C` est trouvée au moment de la compilation pour être une conversion de référence explicite. Cela échoue au moment de l'exécution, car `"1"` n'est pas en fait un `C` . La conversion de `d` en `C` Toutefois, en tant que conversion dynamique explicite, est suspendue au moment de l'exécution, où une conversion définie par l'utilisateur à partir du type au moment de l'exécution de `d` -- `string` --vers `C` est trouvée, et elle est effectuée.

Conversions explicites impliquant des paramètres de type

Les conversions explicites suivantes existent pour un paramètre de type donné `T` :

- De la classe de base effective `C` de `T` à `T` et de toute classe de base de `C` à `T` . Au moment de l'exécution, si `T` est un type valeur, la conversion est exécutée en tant que conversion unboxing. Dans le cas contraire, la conversion est exécutée en tant que conversion de référence explicite ou conversion d'identité.
- De n'importe quel type d'interface à `T` . Au moment de l'exécution, si `T` est un type valeur, la conversion est exécutée en tant que conversion unboxing. Dans le cas contraire, la conversion est exécutée en tant que conversion de référence explicite ou conversion d'identité.
- De `T` à n'importe quel *INTERFACE_TYPE* à `I` condition qu'il n'y ait pas encore de conversion implicite de `T` en `I` . Au moment de l'exécution, si `T` est un type valeur, la conversion est exécutée comme une conversion boxing suivie d'une conversion de référence explicite. Dans le cas contraire, la conversion est exécutée en tant que conversion de référence explicite ou conversion d'identité.
- D'un paramètre `U` de type à `T` , fourni `T` dépend de `U` (contraintes de paramètre detype). Au moment de l'exécution, si `U` est un type valeur, `T` et `U` sont nécessairement du même type et aucune conversion n'est effectuée. Sinon, si `T` est un type valeur, la conversion est exécutée en tant que conversion unboxing. Dans le cas contraire, la conversion est exécutée en tant que conversion de référence explicite ou conversion d'identité.

Si `T` est connu comme un type référence, les conversions ci-dessus sont toutes classées comme des conversions de référence explicites (conversions de référence explicites). Si `T` le type de référence n'est pas connu, les conversions ci-dessus sont classées comme conversions unboxing (conversionsunboxing).

Les règles ci-dessus n'autorisent pas une conversion explicite directe d'un paramètre de type sans contrainte en un type qui n'est pas une interface, ce qui peut être surprenant. La raison de cette règle est d'éviter toute confusion et de rendre la sémantique de ces conversions claire. Observez par exemple la déclaration suivante :

C#

```
class X<T>
{
    public static long F(T t) {
        return (long)t;               // Error
    }
}
```

Si la conversion directe explicite de `t` en `int` était autorisée, vous pouvez vous attendre à ce qu'elle `X<int>.F(7)` retourne `7L` . Toutefois, ce n'est pas le cas, car les conversions numériques standard sont uniquement prises en compte lorsque les types sont

identifiés comme étant numériques au moment de la liaison. Pour que la sémantique soit claire, l'exemple ci-dessus doit plutôt être écrit :

```C#
class X<T>
{
    public static long F(T t) {
        return (long)(object)t;        // Ok, but will only work when T is
long
    }
}
```

Ce code va maintenant être compilé, mais l'exécution `X<int>.F(7)` lèvera une exception au moment de l'exécution, car un boxed `int` ne peut pas être converti directement en `long` .

Conversions explicites définies par l'utilisateur

Une conversion explicite définie par l'utilisateur se compose d'une conversion explicite standard facultative, suivie de l'exécution d'un opérateur de conversion implicite ou explicite défini par l'utilisateur, puis d'une autre conversion explicite standard facultative. Les règles exactes d'évaluation des conversions explicites définies par l'utilisateur sont décrites dans traitement des conversions explicites définies par l'utilisateur.

Conversions standard

Les conversions standard sont les conversions prédéfinies qui peuvent se produire dans le cadre d'une conversion définie par l'utilisateur.

Conversions implicites standard

Les conversions implicites suivantes sont classées comme conversions implicites standard :

- Conversions d'identité (conversion d'identité)
- Conversions numériques implicites (conversions numériques implicites)
- Conversions Nullable implicites (conversions Nullable implicites)
- Conversions de référence implicites (conversions de référence implicites)
- Conversions boxing (conversions boxing)
- Conversions d'expressions constantes implicites (conversions d'expressions constantes implicites)

- Conversions implicites impliquant des paramètres de type (conversions implicites impliquant des paramètres de type)

Les conversions implicites standard excluent spécifiquement les conversions implicites définies par l'utilisateur.

Conversions explicites standard

Les conversions explicites standard sont toutes des conversions implicites standard, ainsi que le sous-ensemble des conversions explicites pour lesquelles il existe une conversion implicite standard opposée. En d'autres termes, si une conversion implicite standard existe d'un type `A` vers un type `B` , une conversion explicite standard existe du type `A` vers le type `B` et du type `B` vers le type `A` .

Conversions définies par l'utilisateur

C# permet aux conversions implicites et explicites prédéfinies d'être complétées par les ***conversions définies*** par l'utilisateur. Les conversions définies par l'utilisateur sont introduites par la déclaration d'opérateurs de conversion (opérateurs de conversion) dans les types classe et struct.

Conversions définies par l'utilisateur autorisées

C# autorise uniquement la déclaration de certaines conversions définies par l'utilisateur. En particulier, il n'est pas possible de redéfinir une conversion implicite ou explicite existante.

Pour un type de source `S` et un type de cible donnés `T` , si `S` ou `T` sont des types Nullable, Let `S0` et `T0` font référence à leurs types sous-jacents, sinon `S0` et `T0` sont égaux à `S` et `T` respectivement. Une classe ou un struct est autorisé à déclarer une conversion d'un type source `S` en type cible `T` uniquement si toutes les conditions suivantes sont vraies :

- `S0` et `T0` sont des types différents.
- `S0`Ou `T0` est le type de classe ou de struct dans lequel la déclaration d'opérateur a lieu.
- Ni `S0` ni `T0` n'est un *INTERFACE_TYPE*.
- En excluant les conversions définies par l'utilisateur, il n'existe pas de conversion de `S` vers `T` ou de à `T` `S` .

Les restrictions qui s'appliquent aux conversions définies par l'utilisateur sont abordées plus en détail dans opérateurs de conversion.

Opérateurs de conversion levés

Dans le cas d'un opérateur de conversion défini par l'utilisateur qui convertit un type valeur n'acceptant pas les valeurs null `S` en un type valeur qui n'autorise pas les valeurs null `T` , il existe un ***opérateur de conversion levé*** qui convertit de `S?` en `T?` . Cet opérateur de conversion levé effectue un désencapsulation de `S?` vers `S` , suivi de la conversion définie par l'utilisateur de `S` à `T` , suivie d'une encapsulation de `T` à `T?` , sauf qu'une valeur null `S?` convertit directement en valeur null `T?` .

Un opérateur de conversion levé a la même classification implicite ou explicite que son opérateur de conversion défini par l'utilisateur sous-jacent. Le terme « conversion définie par l'utilisateur » s'applique à l'utilisation des opérateurs de conversion levés et définis par l'utilisateur.

Évaluation des conversions définies par l'utilisateur

Une conversion définie par l'utilisateur convertit une valeur de son type, appelée «**source type** _ », en un autre type, appelé le *type cible*. Évaluation d'un centre de conversion défini par l'utilisateur lors de la recherche de l'opérateur de conversion le *plus spécifique** défini par l'utilisateur pour les types source et cible particuliers. Cette détermination est divisée en plusieurs étapes :

- Recherche de l'ensemble de classes et de structures à partir desquelles les opérateurs de conversion définis par l'utilisateur seront pris en compte. Cet ensemble se compose du type de source et de ses classes de base, ainsi que du type cible et de ses classes de base (avec les hypothèses implicites que seules les classes et les structs peuvent déclarer des opérateurs définis par l'utilisateur, et que les types non-classes n'ont pas de classes de base). Dans le cadre de cette étape, si le type source ou cible est un *nullable_type*, le type sous-jacent est utilisé à la place.
- À partir de cet ensemble de types, déterminer quels opérateurs de conversion définis par l'utilisateur et levés sont applicables. Pour qu'un opérateur de conversion soit applicable, il doit être possible d'effectuer une conversion standard (conversions standard) du type source en type d'opérande de l'opérateur, et il doit être possible d'effectuer une conversion standard du type de résultat de l'opérateur vers le type cible.
- À partir de l'ensemble des opérateurs définis par l'utilisateur applicables, en déterminant quel opérateur est sans ambiguïté le plus spécifique. En règle

générale, l'opérateur le plus spécifique est l'opérateur dont le type d'opérande est « le plus proche » du type source et dont le type de résultat est « le plus proche » du type cible. Les opérateurs de conversion définis par l'utilisateur sont préférés aux opérateurs de conversion levés. Les règles exactes pour l'établissement de l'opérateur de conversion défini par l'utilisateur le plus spécifique sont définies dans les sections suivantes.

Une fois qu'un opérateur de conversion défini par l'utilisateur le plus spécifique a été identifié, l'exécution réelle de la conversion définie par l'utilisateur implique jusqu'à trois étapes :

- Tout d'abord, si nécessaire, en effectuant une conversion standard du type source vers le type d'opérande de l'opérateur de conversion défini par l'utilisateur ou levé.
- Ensuite, appelez l'opérateur de conversion défini par l'utilisateur ou levé pour effectuer la conversion.
- Enfin, si nécessaire, effectuer une conversion standard du type de résultat de l'opérateur de conversion défini par l'utilisateur ou levé vers le type cible.

L'évaluation d'une conversion définie par l'utilisateur n'implique jamais plus d'un opérateur de conversion défini par l'utilisateur ou levé. En d'autres termes, une conversion de type `S` en type `T` n'exécute jamais d'abord une conversion définie par l'utilisateur de `S` vers `X` , puis exécute une conversion définie par l'utilisateur de `X` vers `T` .

Les définitions exactes de l'évaluation des conversions implicites ou explicites définies par l'utilisateur sont indiquées dans les sections suivantes. Les définitions utilisent les termes suivants :

- Si une conversion implicite standard (conversions implicites standard) existe d'un type `A` vers un type `B` et si ni `A` ni `B` n'est *INTERFACE_TYPE* s, est dit qu'il `A` s'agit d'un ***englobé par** _ `B` et `B` est dit _ *englobe* * `A` .
- Le ***type le plus englobant*** dans un ensemble de types est celui qui englobe tous les autres types dans le jeu. Si aucun type unique n'englobe tous les autres types, le jeu n'a pas de type englobant le plus. En termes plus intuitifs, le type le plus englobant est le plus grand type dans l'ensemble, à savoir le type dans lequel chacun des autres types peut être converti implicitement.
- Le ***type le plus englobé*** dans un ensemble de types est celui qui est englobé par tous les autres types dans le jeu. Si aucun type unique n'est englobé par tous les autres types, le jeu n'a pas de type le plus englobé. En termes plus intuitifs, le type le plus englobé est le type le plus petit dans le jeu, le type qui peut être converti implicitement en chacun des autres types.

Traitement des conversions implicites définies par l'utilisateur

Une conversion implicite définie par l'utilisateur du type `S` en type `T` est traitée comme suit :

- Déterminez les types `S0` et `T0` . Si `S` ou `T` sont des types Nullable `S0` et `T0` sont leurs types sous-jacents, sinon `S0` et `T0` sont égaux à `S` et `T` respectivement.
- Recherchez l'ensemble de types, `D` , à partir duquel les opérateurs de conversion définis par l'utilisateur seront pris en compte. Cet ensemble se compose de `S0` (si `S0` est une classe ou un struct), des classes de base de `S0` (si `S0` est une classe) et `T0` (si `T0` est une classe ou un struct).
- Recherchez l'ensemble des opérateurs de conversion démontés et définis par l'utilisateur applicables `U` . Cet ensemble se compose des opérateurs de conversion implicite définis par l'utilisateur et levés déclarés par les classes ou les structs dans `D` qui convertissent un type englobant en `S` un type englobé par `T` . Si `U` est vide, la conversion n'est pas définie et une erreur de compilation se produit.
- Recherchez le type de source le plus spécifique, `SX` , des opérateurs dans `U` :
  - Si l'un des opérateurs dans `U` Convert from `S` , `SX` est `S` .
  - Sinon, `SX` est le type le plus englobé dans l'ensemble combiné des types de sources des opérateurs dans `U` . Si exactement un type le plus englobé est introuvable, la conversion est ambiguë et une erreur de compilation se produit.
- Recherchez le type cible le plus spécifique, `TX` , des opérateurs dans `U` :
  - Si l'un des opérateurs dans `U` convertir en `T` , `TX` est `T` .
  - Sinon, `TX` est le type le plus englobant dans l'ensemble combiné des types cibles des opérateurs dans `U` . Si un seul type englobant est introuvable, la conversion est ambiguë et une erreur de compilation se produit.
- Recherchez l'opérateur de conversion le plus spécifique :
  - Si `U` contient exactement un opérateur de conversion défini par l'utilisateur qui convertit de `SX` en `TX` , il s'agit de l'opérateur de conversion le plus spécifique.
  - Sinon, si `U` contient exactement un opérateur de conversion levé qui convertit de `SX` en `TX` , il s'agit de l'opérateur de conversion le plus spécifique.
  - Dans le cas contraire, la conversion est ambiguë et une erreur de compilation se produit.
- Enfin, appliquez la conversion :
  - Si `S` n'a pas `SX` la valeur, une conversion implicite standard de `S` en `SX` est effectuée.

- L'opérateur de conversion le plus spécifique est appelé pour convertir de `SX` en `TX` .
- Si `TX` n'a pas `T` la valeur, une conversion implicite standard de `TX` en `T` est effectuée.

Traitement des conversions explicites définies par l'utilisateur

Une conversion explicite définie par l'utilisateur du type `S` en type `T` est traitée comme suit :

- Déterminez les types `S0` et `T0` . Si `S` ou `T` sont des types Nullable `S0` et `T0` sont leurs types sous-jacents, sinon `S0` et `T0` sont égaux à `S` et `T` respectivement.
- Recherchez l'ensemble de types, `D` , à partir duquel les opérateurs de conversion définis par l'utilisateur seront pris en compte. Cet ensemble se compose de `S0` (si `S0` est une classe ou un struct), des classes de base de `S0` (si est `S0` une classe), `T0` (si `T0` est une classe ou un struct) et des classes de base de `T0` (si `T0` est une classe).
- Recherchez l'ensemble des opérateurs de conversion démontés et définis par l'utilisateur applicables `U` . Cet ensemble se compose des opérateurs de conversion implicite ou explicite définis par l'utilisateur déclarés par les classes ou les structs dans `D` qui convertissent un type englobant ou englobé par en `S` un type englobant ou englobé par `T` . Si `U` est vide, la conversion n'est pas définie et une erreur de compilation se produit.
- Recherchez le type de source le plus spécifique, `SX` , des opérateurs dans `U` :
  - Si l'un des opérateurs dans `U` Convert from `S` , `SX` est `S` .
  - Sinon, si l'un des opérateurs en cours de `U` conversion à partir de types qui englobe `S` , `SX` est le type le plus englobé dans l'ensemble combiné des types de source de ces opérateurs. Si aucun type englobant le plus n'est trouvé, la conversion est ambiguë et une erreur de compilation se produit.
  - Sinon, `SX` est le type le plus englobant dans l'ensemble combiné des types de sources des opérateurs dans `U` . Si un seul type englobant est introuvable, la conversion est ambiguë et une erreur de compilation se produit.
- Recherchez le type cible le plus spécifique, `TX` , des opérateurs dans `U` :
  - Si l'un des opérateurs dans `U` convertir en `T` , `TX` est `T` .
  - Sinon, si l'un des opérateurs en cours de `U` conversion vers des types qui sont inclus par `T` , `TX` est le type le plus englobant dans l'ensemble combiné de types cibles de ces opérateurs. Si un seul type englobant est introuvable, la conversion est ambiguë et une erreur de compilation se produit.

    - Sinon, `TX` est le type le plus englobé dans l'ensemble combiné de types cibles des opérateurs dans `U` . Si aucun type englobant le plus n'est trouvé, la conversion est ambiguë et une erreur de compilation se produit.
- Recherchez l'opérateur de conversion le plus spécifique :
    - Si `U` contient exactement un opérateur de conversion défini par l'utilisateur qui convertit de `SX` en `TX` , il s'agit de l'opérateur de conversion le plus spécifique.
    - Sinon, si `U` contient exactement un opérateur de conversion levé qui convertit de `SX` en `TX` , il s'agit de l'opérateur de conversion le plus spécifique.
    - Dans le cas contraire, la conversion est ambiguë et une erreur de compilation se produit.
- Enfin, appliquez la conversion :
    - Si `S` n'a pas `SX` la valeur, une conversion explicite standard de `S` en `SX` est effectuée.
    - L'opérateur de conversion défini par l'utilisateur le plus spécifique est appelé pour convertir de `SX` en `TX` .
    - Si `TX` n'a pas `T` la valeur, une conversion explicite standard de `TX` en `T` est effectuée.

Conversions de fonctions anonymes

Un *anonymous_method_expression* ou *lambda_expression* est classé comme une fonction anonyme (expressions de fonction anonymes). L'expression n'a pas de type mais peut être convertie implicitement en un type délégué compatible ou un type d'arborescence d'expression. Plus précisément, une fonction anonyme `F` est compatible avec un type délégué `D` fourni :

- Si `F` contient un *anonymous_function_signature*, alors `D` et `F` ont le même nombre de paramètres.
- Si `F` ne contient pas d' *anonymous_function_signature*, `D` peut avoir zéro, un ou plusieurs paramètres de tout type, à condition qu'aucun paramètre de n' `D` ait le `out` modificateur de paramètre.
- Si `F` a une liste de paramètres explicitement typée, chaque paramètre dans `D` a le même type et les mêmes modificateurs que le paramètre correspondant dans `F` .
- Si `F` a une liste de paramètres typée implicitement, n' `D` a aucun paramètre `ref` ou `out` .
- Si le corps de `F` est une expression et `D` a un `void` type de retour ou `F` est Async et `D` a le type de retour `Task` , lorsque chaque paramètre de reçoit `F` le type du paramètre correspondant dans `D` , le corps de `F` est une expression valide (

expressionsWRT) qui serait autorisée en tant que *statement_expression* (instructions d'expression).

- Si le corps de `F` est un bloc d'instructions et `D` a un `void` type de retour ou `F` est Async et `D` a le type `Task` de retour, lorsque chaque paramètre de reçoit `F` le type du paramètre correspondant dans `D` , le corps de `F` est un bloc d'instructions valide (blocsWRT) dans lequel aucune `return` instruction ne spécifie une expression.
- Si le corps de `F` est une expression et que `F` est non asynchrone et `D` a un type de retour non void `T` , *ou* `F` est Async et `D` a un type `Task<T>` de retour, lorsque chaque paramètre de reçoit `F` le type du paramètre correspondant dans `D` , le corps de `F` est une expression valide (les expressionsWRT) qui est implicitement convertible en `T` .
- Si le corps de `F` est un bloc d'instructions, et `F` est non asynchrone et `D` a un type de retour non void `T` , *ou* `F` est asynchrone et `D` a un type de retour `Task<T>` , alors quand chaque paramètre de reçoit `F` le type du paramètre correspondant dans `D` , le corps de `F` est un bloc d'instructions valide (blocsWRT) avec un point de terminaison non accessible dans lequel chaque `return` instruction spécifie une expression qui est implicitement convertible en `T` .

À des fins de concision, cette section utilise la forme abrégée pour les types de tâches `Task` et `Task<T>` (fonctions Async).

Une expression lambda `F` est compatible avec un type d'arborescence d'expression `Expression<D>` si `F` est compatible avec le type délégué `D` . Notez que cela ne s'applique pas aux méthodes anonymes, mais uniquement aux expressions lambda.

Certaines expressions lambda ne peuvent pas être converties en types d'arborescence d'expression : bien que la conversion *existe*, elle échoue au moment de la compilation. C'est le cas si l'expression lambda :

- A un corps de *bloc*
- Contient des opérateurs d'assignation simples ou composés
- Contient une expression liée dynamiquement
- Est asynchrone

Les exemples qui suivent utilisent un type délégué générique `Func<A,R>` qui représente une fonction qui accepte un argument de type `A` et retourne une valeur de type `R` :

```
C#

delegate R Func<A,R>(A arg);
```

Dans les affectations

```C#
Func<int,int> f1 = x => x + 1;                // Ok

Func<int,double> f2 = x => x + 1;             // Ok

Func<double,int> f3 = x => x + 1;             // Error

Func<int, Task<int>> f4 = async x => x + 1;    // Ok
```

le paramètre et les types de retour de chaque fonction anonyme sont déterminés à partir du type de la variable à laquelle la fonction anonyme est assignée.

La première assignation convertit correctement la fonction anonyme en type délégué, `Func<int,int>` car, lorsque `x` est donné, le type `int` `x+1` est une expression valide qui est implicitement convertible en type `int` .

De même, la deuxième assignation convertit correctement la fonction anonyme en type délégué `Func<int,double>` , car le résultat de `x+1` (de type `int`) est implicitement convertible en type `double` .

Toutefois, la troisième assignation est une erreur au moment de la compilation, car, lorsque `x` est donné, `double` le résultat de `x+1` (de type `double`) n'est pas implicitement convertible en type `int` .

La quatrième assignation convertit correctement la fonction Async anonyme en type délégué `Func<int, Task<int>>` , car le résultat de `x+1` (de type `int`) est implicitement convertible en type `int` de résultat du type de tâche `Task<int>` .

Les fonctions anonymes peuvent influencer la résolution de surcharge et participer à l'inférence de type. Pour plus d'informations, consultez membres de fonction .

Évaluation de conversions de fonctions anonymes en types délégués

La conversion d'une fonction anonyme en type délégué produit une instance de délégué qui fait référence à la fonction anonyme et au jeu (éventuellement vide) des variables externes capturées qui sont actives au moment de l'évaluation. Lorsque le délégué est appelé, le corps de la fonction anonyme est exécuté. Le code du corps est exécuté à l'aide de l'ensemble des variables externes capturées référencées par le délégué.

La liste d'appel d'un délégué produit à partir d'une fonction anonyme contient une seule entrée. L'objet cible et la méthode cible exacts du délégué ne sont pas spécifiés. En particulier, il n'est pas spécifié si l'objet cible du délégué est `null`, la `this` valeur du membre de fonction englobant ou un autre objet.

Les conversions de fonctions anonymes sémantiquement identiques avec le même ensemble (éventuellement vide) d'instances de variable externe capturées vers les mêmes types délégués sont autorisées (mais pas obligatoires) pour retourner la même instance de délégué. Le terme sémantiquement identique est utilisé ici pour signifier que l'exécution des fonctions anonymes produira, dans tous les cas, les mêmes effets en raison des mêmes arguments. Cette règle permet d'optimiser le code tel que le suivant.

C#

```
delegate double Function(double x);

class Test
{
    static double[] Apply(double[] a, Function f) {
        double[] result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }

    static void F(double[] a, double[] b) {
        a = Apply(a, (double x) => Math.Sin(x));
        b = Apply(b, (double y) => Math.Sin(y));
        ...
    }
}
```

Étant donné que les deux délégués de fonction anonymes ont le même ensemble (vide) de variables externes capturées, et puisque les fonctions anonymes sont sémantiquement identiques, le compilateur est autorisé à faire en sorte que les délégués fassent référence à la même méthode cible. En effet, le compilateur est autorisé à retourner la même instance de délégué à partir des deux expressions de fonction anonymes.

Évaluation de conversions de fonctions anonymes en types d'arborescences d'expression

La conversion d'une fonction anonyme en un type d'arborescence d'expression produit une arborescence d'expression (types d'arborescence d'expression). Plus précisément, l'évaluation de la conversion de fonction anonyme provoque la construction d'une structure d'objet qui représente la structure de la fonction anonyme elle-même. La

structure précise de l'arborescence de l'expression, ainsi que le processus exact de création de celle-ci, sont définis par l'implémentation.

Exemple d'implémentation

Cette section décrit une implémentation possible de conversions de fonctions anonymes en termes d'autres constructions C#. L'implémentation décrite ici est basée sur les mêmes principes que ceux utilisés par le compilateur Microsoft C#, mais il ne s'agit pas d'une implémentation imposée, ni de la seule possibilité. Il mentionne brièvement les conversions vers les arborescences d'expressions, car leur sémantique exacte ne se situe pas dans le cadre de cette spécification.

Le reste de cette section fournit plusieurs exemples de code qui contient des fonctions anonymes avec différentes caractéristiques. Pour chaque exemple, une traduction correspondante vers du code qui utilise uniquement d'autres constructions C# est fournie. Dans les exemples, l'identificateur `D` est supposé par représenter le type délégué suivant :

C#

```
public delegate void D();
```

La forme la plus simple d'une fonction anonyme est celle qui ne capture aucune variable externe :

C#

```
class Test
{
    static void F() {
        D d = () => { Console.WriteLine("test"); };
    }
}
```

Il peut être traduit en une instanciation de délégué qui fait référence à une méthode statique générée par le compilateur dans laquelle le code de la fonction anonyme est placé :

C#

```
class Test
{
    static void F() {
        D d = new D(__Method1);
    }
```

```
    static void __Method1() {
        Console.WriteLine("test");
    }
}
```

Dans l'exemple suivant, la fonction anonyme fait référence aux membres d'instance de `this` :

C#

```
class Test
{
    int x;

    void F() {
        D d = () => { Console.WriteLine(x); };
    }
}
```

Il peut être traduit en une méthode d'instance générée par le compilateur contenant le code de la fonction anonyme :

C#

```
class Test
{
    int x;

    void F() {
        D d = new D(__Method1);
    }

    void __Method1() {
        Console.WriteLine(x);
    }
}
```

Dans cet exemple, la fonction anonyme capture une variable locale :

C#

```
class Test
{
    void F() {
        int y = 123;
        D d = () => { Console.WriteLine(y); };
    }
}
```

La durée de vie de la variable locale doit maintenant être étendue à au moins la durée de vie du délégué de fonction anonyme. Cela peut être obtenu en « délevage » la variable locale dans un champ d'une classe générée par le compilateur. L'instanciation de la variable locale (instanciation de variables locales) correspond ensuite à la création d'une instance de la classe générée par le compilateur, et l'accès à la variable locale correspond à l'accès à un champ dans l'instance de la classe générée par le compilateur. En outre, la fonction anonyme devient une méthode d'instance de la classe générée par le compilateur :

C#

```
class Test
{
    void F() {
        __Locals1 __locals1 = new __Locals1();
        __locals1.y = 123;
        D d = new D(__locals1.__Method1);
    }

    class __Locals1
    {
        public int y;

        public void __Method1() {
            Console.WriteLine(y);
        }
    }
}
```

Enfin, la fonction anonyme suivante capture, ainsi `this` que deux variables locales avec des durées de vie différentes :

C#

```
class Test
{
    int x;

    void F() {
        int y = 123;
        for (int i = 0; i < 10; i++) {
            int z = i * 2;
            D d = () => { Console.WriteLine(x + y + z); };
        }
    }
}
```

Ici, une classe générée par le compilateur est créée pour chaque bloc d'instructions dans lequel les variables locales sont capturées, de sorte que les variables locales dans les différents blocs peuvent avoir des durées de vie indépendantes. Une instance de `__Locals2` , la classe générée par le compilateur pour le bloc d'instructions interne, contient la variable locale `z` et un champ qui fait référence à une instance de `__Locals1` . Une instance de `__Locals1` , la classe générée par le compilateur pour le bloc d'instructions externe, contient la variable locale `y` et un champ qui référence `this` le membre de fonction englobant. Avec ces structures de données, il est possible d'atteindre toutes les variables externes capturées par le biais d'une instance de `__Local2` , et le code de la fonction anonyme peut donc être implémenté comme une méthode d'instance de cette classe.

C#

```
class Test
{
    void F() {
        __Locals1 __locals1 = new __Locals1();
        __locals1.__this = this;
        __locals1.y = 123;
        for (int i = 0; i < 10; i++) {
            __Locals2 __locals2 = new __Locals2();
            __locals2.__locals1 = __locals1;
            __locals2.z = i * 2;
            D d = new D(__locals2.__Method1);
        }
    }

    class __Locals1
    {
        public Test __this;
        public int y;
    }

    class __Locals2
    {
        public __Locals1 __locals1;
        public int z;

        public void __Method1() {
            Console.WriteLine(__locals1.__this.x + __locals1.y + z);
        }
    }
}
```

La même technique appliquée ici pour capturer les variables locales peut également être utilisée lors de la conversion de fonctions anonymes en arborescences d'expressions : les références aux objets générés par le compilateur peuvent être stockées dans

l'arborescence de l'expression, et l'accès aux variables locales peut être représenté en tant qu'accès au champ sur ces objets. L'avantage de cette approche est qu'elle permet aux variables locales « levées » d'être partagées entre les délégués et les arborescences d'expressions.

Conversions de groupes de méthodes

Une conversion implicite (conversions implicites) existe d'un groupe de méthodes (classifications d'expressions) en un type délégué compatible. Pour un type délégué `D` et une expression `E` classés en tant que groupe de méthodes, une conversion implicite existe de `E` en `D` si `E` contient au moins une méthode applicable dans sa forme normale (fonction membre applicable) à une liste d'arguments construite à l'aide des types de paramètres et des modificateurs de `D` , comme décrit dans l'exemple suivant.

L'application de compilation d'une conversion d'un groupe de méthodes `E` en un type délégué `D` est décrite dans ce qui suit. Notez que l'existence d'une conversion implicite de `E` en `D` ne garantit pas que l'application de compilation de la conversion échouera sans erreur.

- Une méthode unique `M` est sélectionnée correspondant à un appel de méthode (appels de méthode) du formulaire `E(A)` , avec les modifications suivantes :
  - La liste `A` d'arguments est une liste d'expressions, chacune classée comme une variable et avec le type et le modificateur (`ref` ou `out`) du paramètre correspondant dans le *formal_parameter_list* de `D` .
  - Les méthodes candidates prises en compte sont uniquement celles qui sont applicables dans leur forme normale (fonction membre applicable), et non celles qui s'appliquent uniquement dans leur forme développée.
- Si l'algorithme d' appels de méthode génère une erreur, une erreur de compilation se produit. Dans le cas contraire, l'algorithme produit une seule meilleure méthode qui `M` a le même nombre de paramètres que `D` et la conversion est considérée comme existante.
- La méthode sélectionnée `M` doit être compatible (déléguer la compatibilité) avec le type délégué `D` , ou sinon, une erreur de compilation se produit.
- Si la méthode sélectionnée `M` est une méthode d'instance, l'expression d'instance associée à `E` détermine l'objet cible du délégué.
- Si la méthode sélectionnée M est une méthode d'extension qui est indiquée au moyen d'un accès de membre sur une expression d'instance, cette expression de l'instance détermine l'objet cible du délégué.

- Le résultat de la conversion est une valeur de type `D`, à savoir un délégué nouvellement créé qui fait référence à la méthode et à l'objet cible sélectionnés.
- Notez que ce processus peut mener à la création d'un délégué à une méthode d'extension, si l'algorithme d' appels de méthode ne parvient pas à trouver une méthode d'instance mais réussit à traiter l'appel de `E(A)` comme un appel de méthode d'extension (appels de méthode d'extension). Un délégué ainsi créé capture la méthode d'extension ainsi que son premier argument.

L'exemple suivant illustre les conversions de groupe de méthodes :

C#

```
delegate string D1(object o);

delegate object D2(string s);

delegate object D3();

delegate string D4(object o, params object[] a);

delegate string D5(int i);

class Test
{
    static string F(object o) {...}

    static void G() {
        D1 d1 = F;            // Ok
        D2 d2 = F;            // Ok
        D3 d3 = F;            // Error -- not applicable
        D4 d4 = F;            // Error -- not applicable in normal form
        D5 d5 = F;            // Error -- applicable but not compatible

    }
}
```

Assignation qui `d1` convertit implicitement le groupe `F` de méthodes en une valeur de type `D1`.

L'assignation pour `d2` montrer comment il est possible de créer un délégué vers une méthode qui a des types de paramètres moins dérivés (contravariant) et un type de retour plus dérivé (covariant).

L'assignation pour `d3` montrer comment aucune conversion n'existe si la méthode n'est pas applicable.

L'assignation pour `d4` montrer comment la méthode doit être applicable dans sa forme normale.

L'assignation de `d5` montre comment les types de paramètres et de retour du délégué et de la méthode sont autorisés à différer uniquement pour les types référence.

Comme avec toutes les autres conversions implicites et explicites, l'opérateur de cast peut être utilisé pour effectuer explicitement une conversion de groupe de méthodes. Ainsi, l'exemple

```
C#

object obj = new EventHandler(myDialog.OkClick);
```

à la place, il est possible d'écrire

```
C#

object obj = (EventHandler)myDialog.OkClick;
```

Les groupes de méthodes peuvent influencer la résolution de surcharge et participer à l'inférence de type. Pour plus d'informations, consultez membres de fonction .

L'évaluation au moment de l'exécution d'une conversion de groupe de méthodes se déroule comme suit :

- Si la méthode sélectionnée au moment de la compilation est une méthode d'instance, ou s'il s'agit d'une méthode d'extension accessible en tant que méthode d'instance, l'objet cible du délégué est déterminé à partir de l'expression d'instance associée à `E` :
  - L'expression d'instance est évaluée. Si cette évaluation provoque une exception, aucune autre étape n'est exécutée.
  - Si l'expression d'instance est d'un *reference_type*, la valeur calculée par l'expression d'instance devient l'objet cible. Si la méthode sélectionnée est une méthode d'instance et que l'objet cible est `null` , une `System.NullReferenceException` est levée et aucune autre étape n'est exécutée.
  - Si l'expression d'instance est d'un *Value_type*, une opération boxing (conversions boxing) est effectuée pour convertir la valeur en objet, et cet objet devient l'objet cible.
- Sinon, la méthode sélectionnée fait partie d'un appel de méthode statique, et l'objet cible du délégué est `null` .
- Une nouvelle instance du type délégué `D` est allouée. Si la mémoire disponible est insuffisante pour allouer la nouvelle instance, une `System.OutOfMemoryException` est levée et aucune autre étape n'est exécutée.

- La nouvelle instance de délégué est initialisée avec une référence à la méthode qui a été déterminée au moment de la compilation et une référence à l'objet cible calculé ci-dessus.

Expressions

Article • 13/05/2021

Une expression est une séquence d'opérateurs et d'opérandes. Ce chapitre définit la syntaxe, l'ordre d'évaluation des opérandes et des opérateurs, ainsi que la signification des expressions.

Classifications des expressions

Une expression est classée comme l'un des éléments suivants :

- Valeur. Chaque valeur possède un type associé.
- Variable. Chaque variable a un type associé, à savoir le type déclaré de la variable.
- Espace de noms. Une expression avec cette classification ne peut apparaître qu'en tant que partie gauche d'un *member_access* (accès aux membres). Dans tout autre contexte, une expression classée en tant qu'espace de noms provoque une erreur de compilation.
- Type. Une expression avec cette classification ne peut apparaître qu'en tant que partie gauche d'un *member_access* (accès aux membres), ou en tant qu'opérande pour l' `as` opérateur (opérateur As), l' `is` opérateur (opérateur is) ou l' `typeof` opérateur (opérateur typeof). Dans tout autre contexte, une expression classifiée comme un type provoque une erreur au moment de la compilation.
- Un groupe de méthodes, qui est un ensemble de méthodes surchargées résultant d'une recherche de membre (recherche de membre). Un groupe de méthodes peut avoir une expression d'instance associée et une liste d'arguments de type associée. Quand une méthode d'instance est appelée, le résultat de l'évaluation de l'expression d'instance devient l'instance représentée par `this` (cet accès). Un groupe de méthodes est autorisé dans un *invocation_expression* (expressions d'appel), un *delegate_creation_expression* (expressions de création de délégué) et en tant que côté gauche d'un opérateur is et peut être converti implicitement en un type délégué compatible (conversions de groupe de méthodes). Dans tout autre contexte, une expression classifiée comme un groupe de méthodes provoque une erreur au moment de la compilation.
- Littéral null. Une expression avec cette classification peut être implicitement convertie en un type référence ou un type Nullable.
- Fonction anonyme. Une expression avec cette classification peut être convertie implicitement en un type délégué compatible ou un type d'arborescence d'expression.

- Accès à une propriété. Chaque accès à une propriété a un type associé, à savoir le type de la propriété. En outre, un accès à une propriété peut avoir une expression d'instance associée. Lorsqu'un accesseur (`get` le `set` bloc ou) d'un accès à une propriété d'instance est appelé, le résultat de l'évaluation de l'expression d'instance devient l'instance représentée par `this` (cet accès).
- Accès à un événement. Chaque accès à un événement a un type associé, à savoir le type de l'événement. En outre, un accès aux événements peut avoir une expression d'instance associée. Un accès aux événements peut apparaître en tant qu'opérande gauche des `+=` `-=` opérateurs et (assignation d'événement). Dans tout autre contexte, une expression classifiée comme un accès à un événement provoque une erreur de compilation.
- Un accès à un indexeur. Chaque accès à l'indexeur a un type associé, à savoir le type d'élément de l'indexeur. En outre, un accès à un indexeur a une expression d'instance associée et une liste d'arguments associée. Lorsqu'un accesseur (`get` le `set` bloc ou) d'un accès à un indexeur est appelé, le résultat de l'évaluation de l'expression d'instance devient l'instance représentée par `this` (cet accès) et le résultat de l'évaluation de la liste d'arguments devient la liste de paramètres de l'appel.
- Nothing. Cela se produit lorsque l'expression est un appel d'une méthode avec un type de retour `void` . Une expression classifiée comme Nothing n'est valide que dans le contexte d'une *statement_expression* (instructions d'expression).

Le résultat final d'une expression n'est jamais un espace de noms, un type, un groupe de méthodes ou un accès à un événement. Au lieu de cela, comme indiqué ci-dessus, ces catégories d'expressions sont des constructions intermédiaires qui sont uniquement autorisées dans certains contextes.

Un accès à une propriété ou un accès à un indexeur est toujours reclassifié comme valeur en effectuant un appel de l' *accesseur Get* ou de l' *accesseur Set*. L'accesseur particulier est déterminé par le contexte de l'accès à la propriété ou à l'indexeur : si l'accès est la cible d'une assignation, l' *accesseur Set* est appelé pour assigner une nouvelle valeur (assignation simple). Dans le cas contraire, l' *accesseur Get* est appelé pour obtenir la valeur actuelle (valeurs des expressions).

Valeurs des expressions

La plupart des constructions qui impliquent une expression requièrent finalement que l'expression désigne une ***valeur***. Dans ce cas, si l'expression réelle désigne un espace de noms, un type, un groupe de méthodes ou rien, une erreur de compilation se produit. Toutefois, si l'expression désigne un accès à une propriété, un indexeur ou une variable, la valeur de la propriété, de l'indexeur ou de la variable est implicitement remplacée :

- La valeur d'une variable est simplement la valeur actuellement stockée dans l'emplacement de stockage identifié par la variable. Une variable doit être considérée comme assignée de manière définitive (assignation définie) avant que sa valeur puisse être obtenue, sinon une erreur de compilation se produit.
- La valeur d'une expression d'accès à la propriété est obtenue en appelant l' *accesseur Get* de la propriété. Si la propriété n'a pas d' *accesseur Get*, une erreur de compilation se produit. Sinon, un appel de membre de fonction (vérification au moment de la compilation de la résolution de surcharge dynamique) est effectué et le résultat de l'appel devient la valeur de l'expression d'accès à la propriété.
- La valeur d'une expression d'accès à l'indexeur est obtenue en appelant l' *accesseur Get* de l'indexeur. Si l'indexeur n'a pas d' *accesseur Get*, une erreur de compilation se produit. Sinon, un appel de membre de fonction (vérification au moment de la compilation de la résolution de surcharge dynamique) est effectué avec la liste d'arguments associée à l'expression d'accès de l'indexeur, et le résultat de l'appel devient la valeur de l'expression d'accès de l'indexeur.

Liaison statique et dynamique

Le processus de détermination de la signification d'une opération en fonction du type ou de la valeur des expressions constitutives (arguments, opérandes, récepteurs) est souvent appelé ***liaison***. Par exemple, la signification d'un appel de méthode est déterminée en fonction du type du récepteur et des arguments. La signification d'un opérateur est déterminée en fonction du type de ses opérandes.

En C#, le sens d'une opération est généralement déterminé au moment de la compilation, en fonction du type au moment de la compilation des expressions qui le composent. De même, si une expression contient une erreur, l'erreur est détectée et signalée par le compilateur. Cette approche est connue sous le nom de ***liaison statique***.

Toutefois, si une expression est une expression dynamique (par exemple, a le type `dynamic`), cela indique que toute liaison à laquelle elle participe doit être basée sur son type au moment de l'exécution (par exemple, le type réel de l'objet qu'elle désigne au moment de l'exécution) plutôt que le type qu'elle a au moment de la compilation. La liaison d'une telle opération est donc différée jusqu'au moment où l'opération doit être exécutée pendant l'exécution du programme. C'est ce que l'on appelle la ***liaison dynamique***.

Lorsqu'une opération est liée de manière dynamique, le compilateur n'effectue que peu ou pas de vérification. Au lieu de cela, si la liaison au moment de l'exécution échoue, les erreurs sont signalées en tant qu'exceptions au moment de l'exécution.

Les opérations suivantes en C# sont sujettes à la liaison :

- Accès au membre : `e.M`
- Appel de méthode : `e.M(e1, ..., eN)`
- Appel de délégué : `e(e1, ..., eN)`
- Accès à l'élément : `e[e1, ..., eN]`
- Création d'objets : `new C(e1, ..., eN)`
- Opérateurs unaires surchargés : `+` , `-` , `!` , `~` , `++` , `--` , `true` , `false`
- Opérateurs binaires surchargés : `+` , `-` ,, `*` `/` , `%` , `&` , `&&` , `|` , `||` , `??` , `^` , `<<` , `>>` , `==` ,,,,,,,,, `!=` , `>` , `<` , `>=` , `<=`
- Opérateurs d'assignation : `=` , `+=` ,, `-=` `*=` , `/=` , `%=` , `&=` , `|=` , `^=` , `<<=` , `>>=`
- Conversions implicites et explicites

Quand aucune expression dynamique n'est impliquée, C# utilise par défaut la liaison statique, ce qui signifie que les types de compilation des expressions constitutives sont utilisés dans le processus de sélection. Toutefois, lorsque l'une des expressions constitutives des opérations listées ci-dessus est une expression dynamique, l'opération est liée dynamiquement.

Liaison-heure

La liaison statique a lieu au moment de la compilation, tandis que la liaison dynamique a lieu au moment de l'exécution. Dans les sections suivantes, le terme « ***Binding-Time*** » fait référence à la compilation ou au moment de l'exécution, selon le moment où la liaison a lieu.

L'exemple suivant illustre les notions de liaison statique et dynamique et de l'heure de liaison :

```
C#

object  o = 5;
dynamic d = 5;

Console.WriteLine(5);  // static  binding to Console.WriteLine(int)
Console.WriteLine(o);  // static  binding to Console.WriteLine(object)
Console.WriteLine(d);  // dynamic binding to Console.WriteLine(int)
```

Les deux premiers appels sont liés de manière statique : la surcharge de `Console.WriteLine` est choisie en fonction du type au moment de la compilation de son argument. Ainsi, la durée de la liaison est au moment de la compilation.

Le troisième appel est lié dynamiquement : la surcharge de `Console.WriteLine` est choisie en fonction du type au moment de l'exécution de son argument. Cela se produit

parce que l'argument est une expression dynamique--son type au moment de la compilation est `dynamic` . Ainsi, le temps de liaison pour le troisième appel est au moment de l'exécution.

Liaison dynamique

L'objectif de la liaison dynamique est de permettre aux programmes C# d'interagir avec les ***objets dynamiques***, c'est-à-dire les objets qui ne respectent pas les règles normales du système de type C#. Les objets dynamiques peuvent être des objets provenant d'autres langages de programmation avec des systèmes de types différents, ou des objets qui sont configurés par programme pour implémenter leur propre sémantique de liaison pour différentes opérations.

Le mécanisme par lequel un objet dynamique implémente sa propre sémantique est défini par l'implémentation. Une interface donnée, à nouveau définie, est implémentée par les objets dynamiques pour signaler à la Runtime C# qu'ils ont une sémantique spéciale. Ainsi, chaque fois que les opérations sur un objet dynamique sont liées de manière dynamique, leur propre sémantique de liaison, plutôt que celles de C# comme spécifié dans ce document, sont reportées.

Bien que l'objectif de la liaison dynamique soit d'autoriser l'interopérabilité avec les objets dynamiques, C# autorise la liaison dynamique sur tous les objets, qu'ils soient dynamiques ou non. Cela permet une intégration plus fluide des objets dynamiques, car les résultats des opérations sur ceux-ci peuvent ne pas être eux-mêmes des objets dynamiques, mais ils sont toujours d'un type inconnu pour le programmeur au moment de la compilation. En outre, la liaison dynamique permet d'éliminer le code basé sur la réflexion, susceptible d'engendrer des erreurs, même quand aucun objet impliqué n'est un objet dynamique.

Les sections suivantes décrivent pour chaque construction dans le langage exactement quand la liaison dynamique est appliquée, la vérification du temps de compilation, le cas échéant, et le résultat de la classification des expressions et des résultats au moment de la compilation.

Types d'expressions constitutives

Lorsqu'une opération est liée de manière statique, le type d'une expression de composant (par exemple, un récepteur, un argument, un index ou un opérande) est toujours considéré comme le type au moment de la compilation de cette expression.

Lorsqu'une opération est liée de manière dynamique, le type d'une expression constitutive est déterminé de différentes manières selon le type de compilation de

l'expression constitutive :

- Une expression constitutive du type au moment de `dynamic` la compilation est considérée comme ayant le type de la valeur réelle évaluée par l'expression au moment de l'exécution
- Une expression de composant dont le type au moment de la compilation est un paramètre de type est considérée comme ayant le type auquel le paramètre de type est lié au moment de l'exécution
- Dans le cas contraire, l'expression constitutive est considérée comme ayant le type au moment de la compilation.

Opérateurs

Les expressions sont construites à partir des *opérateurs****opérandes** et. Les opérateurs d'une expression indiquent les opérations à appliquer aux opérandes. Parmi les exemples d'opérateurs figurent `+`, `-`, `_`, `/` et `new`. Les littéraux, les champs, les variables locales et les expressions sont des exemples d'opérandes.

Il existe trois types d'opérateurs :

- Les opérateurs unaires. Les opérateurs unaires acceptent un opérande et utilisent soit la notation de préfixe (telle que), soit la `--x` notation suffixée (telle que `x++`).
- Opérateurs binaires. Les opérateurs binaires prennent deux opérandes et utilisent tous la notation infixe (telle que `x + y`).
- Opérateur ternaire. Un seul opérateur ternaire, `?:` , existe ; il prend trois opérandes et utilise la notation infixe (`c ? x : y`).

L'ordre d'évaluation des opérateurs dans une expression est déterminé par les ***précédence*** _ *et* _ *associativité* des opérateurs (priorité et associativitédes opérateurs).

Les opérandes d'une expression sont évalués de gauche à droite. Par exemple, dans `F(i) + G(i++) * H(i)` , `F` la méthode est appelée à l'aide de l'ancienne valeur de `i` , puis la méthode `G` est appelée avec l'ancienne valeur de `i` et, enfin, `H` la méthode est appelée avec la nouvelle valeur de `i` . Cela est indépendant de et non lié à la priorité des opérateurs.

Certains opérateurs peuvent être ***surchargés***. La surcharge d'opérateur permet de spécifier des implémentations d'opérateur définies par l'utilisateur pour les opérations où l'un des opérandes ou les deux sont d'un type de classe ou de struct défini par l'utilisateur (surcharge d'opérateur).

Priorité des opérateurs et associativité

Lorsqu'une expression contient plusieurs opérateurs, le *__précédence__ _ des opérateurs contrôle l'ordre dans lequel les opérateurs individuels sont évalués. Par exemple, l'expression `x + y _ z` est évaluée en tant que `x + (y * z)`, car l'opérateur `*` a une priorité plus élevée que l'opérateur `+` binaire. La priorité d'un opérateur est établie par la définition de sa production grammaticale associée. Par exemple, un *additive_expression* se compose d'une séquence de *multiplicative_expression* s séparés par des `+` `-` opérateurs Or, donnant ainsi aux `+` opérateurs et une priorité plus `-` faible que les `*` `/` opérateurs, et `%` .

Le tableau suivant récapitule tous les opérateurs par ordre de priorité, du plus élevé au plus bas :

Section	Catégorie	Opérateurs
Expressions primaires	Principal	`x.y` `f(x)` `a[x]` `x++` `x--` `new` `typeof` `default` `checked` `unchecked` `delegate`
Opérateurs unaires	Unaire	`+` `-` `!` `~` `++x` `--x` `(T)x`
Opérateurs arithmétiques	Multiplicatif	`*` `/` `%`
Opérateurs arithmétiques	Additive	`+` `-`
Opérateurs de décalage	Shift	`<<` `>>`
Opérateurs relationnels et de test de type	Relations et test de type	`<` `>` `<=` `>=` `is` `as`
Opérateurs relationnels et de test de type	Égalité	`==` `!=`
Opérateurs logiques	ET logique	`&`
Opérateurs logiques	XOR logique	`^`
Opérateurs logiques	OU logique	`|`
Opérateurs logiques conditionnels	AND conditionnel	`&&`
Opérateurs logiques conditionnels	OR conditionnel	`||`
Opérateur de fusion de Null	Fusion de Null	`??`

Section	Catégorie	Opérateurs
Opérateur conditionnel	Logique conditionnelle	`?:`
Opérateurs d'assignation, expressions de fonction anonymes	Assignation et expression lambda	`=` `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `^=` `|=` `=>`

Lorsqu'un opérande se produit entre deux opérateurs de même priorité, l'associativité des opérateurs détermine l'ordre dans lequel les opérations sont effectuées :

- À l'exception des opérateurs d'assignation et de l'opérateur de fusion Null, tous les opérateurs binaires sont ***associatifs à gauche***, ce qui signifie que les opérations sont effectuées de gauche à droite. Par exemple, `x + y + z` est évalué comme étant `(x + y) + z`.
- Les opérateurs d'assignation, l'opérateur de fusion Null et l'opérateur conditionnel (`?:`) sont ***associatifs à droite***, ce qui signifie que les opérations sont exécutées de droite à gauche. Par exemple, `x = y = z` est évalué comme étant `x = (y = z)`.

La priorité et l'associativité peuvent être contrôlées à l'aide de parenthèses. Par exemple, `x + y * z` multiplie d'abord `y` par `z`, puis ajoute le résultat à `x`, mais `(x + y) * z` ajoute d'abord `x` et `y`, puis multiplie le résultat par `z`.

Surcharge d'opérateur

Tous les opérateurs unaires et binaires ont des implémentations prédéfinies qui sont automatiquement disponibles dans toute expression. Outre les implémentations prédéfinies, les implémentations définies par l'utilisateur peuvent être introduites en incluant `operator` des déclarations dans des classes et des structs (opérateurs). Les implémentations d'opérateur définies par l'utilisateur ont toujours priorité sur les implémentations d'opérateur prédéfinies : uniquement lorsqu'il n'existe aucune implémentation d'opérateur définie par l'utilisateur applicable, les implémentations d'opérateur prédéfinies sont prises en compte, comme décrit dans résolution de surcharge d'opérateur unaire et résolution de surcharge d'opérateur binaire.

Les ***opérateurs unaires surchargeables*** sont :

C#

```
+   -   !   ~   ++   --   true   false
```

Bien que `true` et ne `false` soient pas utilisés explicitement dans les expressions (et, par conséquent, ne sont pas inclus dans la table de précédence dans la priorité d' opérateur et l'associativité), ils sont considérés comme des opérateurs, car ils sont appelés dans plusieurs contextes d'expression : des expressions booléennes (expressions booléennes) et des expressions impliquant le conditionnel (opérateur conditionnel) et des opérateurs logique

Les ***opérateurs binaires surchargeables*** sont les suivants :

```
C#

+   -   *   /   %   &   |   ^   <<   >>   ==   !=   >   <   >=   <=
```

Seuls les opérateurs listés ci-dessus peuvent être surchargés. En particulier, il n'est pas possible de surcharger l'accès aux membres, l'appel de méthode, ou les `=` `&&` opérateurs,, `||` , `??` ,, `?:` `=>` ,,,,, `checked` `unchecked` `new` `typeof` `default` , `as` et `is` .

Quand un opérateur binaire est surchargé, l'opérateur d'assignation correspondant, le cas échéant, est aussi implicitement surchargé. Par exemple, une surcharge de l'opérateur `*` est également une surcharge de l'opérateur `*=` . Cela est décrit plus en détail dans assignation composée. Notez que l'opérateur d'assignation lui-même (`=`) ne peut pas être surchargé. Une assignation effectue toujours une simple copie bit-wise d'une valeur dans une variable.

Les opérations de cast, telles que `(T)x` , sont surchargées en fournissant des conversions définies par l'utilisateur (conversions définies par l'utilisateur).

L'accès à l'élément, tel que `a[x]` , n'est pas considéré comme un opérateur surchargeable. À la place, l'indexation définie par l'utilisateur est prise en charge par le biais d'indexeurs (indexeurs).

Dans les expressions, les opérateurs sont référencés à l'aide de la notation opérateur, et dans les déclarations, les opérateurs sont référencés à l'aide de la notation fonctionnelle. Le tableau suivant montre la relation entre les notations opérateur et opérateur pour les opérateurs unaires et binaires. Dans la première entrée, *op* dénote n'importe quel opérateur de préfixe unaire surchargeable. Dans la deuxième entrée, *op* dénote le suffixe et les opérateurs unaires `++` `--` . Dans la troisième entrée, *op* dénote tout opérateur binaire surchargeable.

Notation d'opérateur	Notation fonctionnelle
`op x`	`operator op(x)`

Notation d'opérateur	Notation fonctionnelle
`x op`	`operator op(x)`
`x op y`	`operator op(x,y)`

Les déclarations d'opérateur définies par l'utilisateur requièrent toujours qu'au moins un des paramètres soit du type de classe ou de struct qui contient la déclaration d'opérateur. Par conséquent, il n'est pas possible pour un opérateur défini par l'utilisateur d'avoir la même signature qu'un opérateur prédéfini.

Les déclarations d'opérateur définies par l'utilisateur ne peuvent pas modifier la syntaxe, la priorité ou l'associativité d'un opérateur. Par exemple, l' `/` opérateur est toujours un opérateur binaire, a toujours le niveau de précédence spécifié dans priorité et associativité des opérateurs, et est toujours associatif à gauche.

Bien qu'il soit possible pour un opérateur défini par l'utilisateur d'effectuer n'importe quel calcul, les implémentations qui produisent des résultats autres que ceux qui sont intuitivement attendus sont fortement déconseillées. Par exemple, une implémentation de `operator ==` doit comparer les deux opérandes pour déterminer leur égalité et retourner un `bool` résultat approprié.

Les descriptions des opérateurs individuels dans les expressions primaires par le biais d' opérateurs logiques conditionnels spécifient les implémentations prédéfinies des opérateurs et toutes les règles supplémentaires qui s'appliquent à chaque opérateur. Les descriptions utilisent les termes * résolution de **surcharge d'opérateur unaire** , *résolution de surcharge d'opérateur binaire et _ promotion numérique *, définitions de qui se trouvent dans les sections suivantes.*

Résolution de surcharge d'opérateur unaire

Une opération de la forme `op x` ou `x op` , où `op` est un opérateur unaire surchargeable, et `x` est une expression de type `X` , est traitée comme suit :

- Le jeu d'opérateurs candidats définis par l'utilisateur fourni par `X` pour l'opération `operator op(x)` est déterminé à l'aide des règles des opérateurs candidats définispar l'utilisateur.
- Si le jeu d'opérateurs candidats définis par l'utilisateur n'est pas vide, il devient alors l'ensemble des opérateurs candidats pour l'opération. Dans le cas contraire, les implémentations unaires prédéfinies `operator op` , y compris leurs formulaires levés, deviennent l'ensemble des opérateurs candidats pour l'opération. Les

implémentations prédéfinies d'un opérateur donné sont spécifiées dans la description de l'opérateur (expressions primaires et opérateurs unaires).
- Les règles de résolution de surcharge de la résolution de surcharge sont appliquées au jeu d'opérateurs candidats pour sélectionner le meilleur opérateur par rapport à la liste d'arguments `(x)`, et cet opérateur devient le résultat du processus de résolution de surcharge. Si la résolution de surcharge ne peut pas sélectionner un seul opérateur Best, une erreur de liaison est générée.

Résolution de surcharge d'opérateur binaire

Une opération de la forme `x op y`, où `op` est un opérateur binaire surchargeable, `x` est une expression de type `X` et `y` est une expression de type `Y`, est traitée comme suit :

- L'ensemble des opérateurs candidats définis par l'utilisateur fournis par `X` et `Y` pour l'opération `operator op(x,y)` est déterminé. L'ensemble se compose de l'Union des opérateurs candidats fournis par `X` et des opérateurs candidats fournis par `Y`, chacun déterminé à l'aide des règles des opérateurs candidats définispar l'utilisateur. Si `X` et `Y` sont du même type, ou si `X` et `Y` sont dérivés d'un type de base commun, les opérateurs candidats partagés ne se produisent que dans l'ensemble combiné une fois.
- Si le jeu d'opérateurs candidats définis par l'utilisateur n'est pas vide, il devient alors l'ensemble des opérateurs candidats pour l'opération. Dans le cas contraire, les implémentations binaires prédéfinies `operator op`, y compris leurs formulaires levés, deviennent l'ensemble des opérateurs candidats pour l'opération. Les implémentations prédéfinies d'un opérateur donné sont spécifiées dans la description de l'opérateur (opérateurs arithmétiques à l'aide d' opérateurs logiques conditionnels). Pour les opérateurs enum et Delegate prédéfinis, les seuls opérateurs pris en compte sont ceux définis par un type enum ou Delegate qui est le type de temps de liaison de l'un des opérandes.
- Les règles de résolution de surcharge de la résolution de surcharge sont appliquées au jeu d'opérateurs candidats pour sélectionner le meilleur opérateur par rapport à la liste d'arguments `(x,y)`, et cet opérateur devient le résultat du processus de résolution de surcharge. Si la résolution de surcharge ne peut pas sélectionner un seul opérateur Best, une erreur de liaison est générée.

Opérateurs candidats définis par l'utilisateur

Étant donné un type `T` et une opération `operator op(A)`, où `op` est un opérateur surchargeable et `A` est une liste d'arguments, l'ensemble des opérateurs candidats définis par l'utilisateur fournis par `T` pour `operator op(A)` est déterminé comme suit :

- Déterminez le type `T0` . Si `T` est un type Nullable, `T0` est son type sous-jacent ; sinon, `T0` est égal à `T` .
- Pour toutes les `operator op` déclarations dans `T0` et toutes les formes levées de tels opérateurs, si au moins un opérateur est applicable (membre de fonction applicable) par rapport à la liste d'arguments `A` , alors l'ensemble des opérateurs candidats se compose de tous les opérateurs applicables dans `T0` .
- Sinon, si `T0` a `object` la valeur, l'ensemble des opérateurs candidats est vide.
- Sinon, le jeu d'opérateurs candidats fourni par `T0` est l'ensemble des opérateurs candidats fournis par la classe de base directe de `T0` ou la classe de base effective de `T0` si `T0` est un paramètre de type.

Promotions numériques

La promotion numérique consiste à effectuer automatiquement certaines conversions implicites des opérandes des opérateurs numériques unaires et binaires prédéfinis. La promotion numérique n'est pas un mécanisme distinct, mais plutôt un effet de l'application de la résolution de surcharge aux opérateurs prédéfinis. La promotion numérique n'affecte pas spécifiquement l'évaluation des opérateurs définis par l'utilisateur, bien que les opérateurs définis par l'utilisateur puissent être implémentés pour présenter des effets similaires.

À titre d'exemple de promotion numérique, prenez en compte les implémentations prédéfinies de l' `*` opérateur binaire :

C#

```
int operator *(int x, int y);
uint operator *(uint x, uint y);
long operator *(long x, long y);
ulong operator *(ulong x, ulong y);
float operator *(float x, float y);
double operator *(double x, double y);
decimal operator *(decimal x, decimal y);
```

Lorsque des règles de résolution de surcharge (résolution de surcharge) sont appliquées à cet ensemble d'opérateurs, l'effet est de sélectionner le premier des opérateurs pour lesquels des conversions implicites existent à partir des types d'opérandes. Par exemple, pour l'opération `b * s` , où `b` est un `byte` et `s` est un `short` , la résolution de surcharge sélectionne `operator *(int,int)` comme meilleur opérateur. Ainsi, l'effet est que `b` et `s` sont convertis en `int` , et le type du résultat est `int` . De même, pour l'opération `i * d` , où `i` est un `int` et `d` est un `double` , la résolution de surcharge sélectionne `operator *(double,double)` comme meilleur opérateur.

Promotions numériques unaires

La promotion numérique unaire se produit pour les opérandes des `+` `-` opérateurs unaires, et prédéfinis `~` . La promotion numérique unaire consiste simplement à convertir des opérandes de type `sbyte` , `byte` ,, `short` `ushort` ou `char` en type `int` . En outre, pour l'opérateur unaire `-` , la promotion numérique unaire convertit les opérandes de type `uint` en type `long` .

Promotions numériques binaires

La promotion numérique binaire se produit pour les opérandes des `+` `-` `*` `/` `%` `&` `|` `^` `==` `!=` `>` `<` `>=` `<=` opérateurs binaires prédéfinis,,,,,,,,,, et. La promotion numérique binaire convertit implicitement les deux opérandes en un type commun qui, dans le cas des opérateurs non relationnels, devient également le type de résultat de l'opération. La promotion numérique binaire consiste à appliquer les règles suivantes, dans l'ordre où elles apparaissent ici :

- Si l'un des opérandes est de type `decimal` , l'autre opérande est converti en type `decimal` , ou une erreur de liaison au moment de la liaison se produit si l'autre opérande est de type `float` ou `double` .
- Sinon, si l'un des opérandes est de type `double` , l'autre opérande est converti en type `double` .
- Sinon, si l'un des opérandes est de type `float` , l'autre opérande est converti en type `float` .
- Sinon, si l'un des opérandes est de type `ulong` , l'autre opérande est converti en type `ulong` , ou une erreur de liaison au moment de la liaison se produit si l'autre opérande est de type `sbyte` ,, `short` `int` ou `long` .
- Sinon, si l'un des opérandes est de type `long` , l'autre opérande est converti en type `long` .
- Sinon, si l'un des opérandes est de type `uint` et que l'autre opérande est de type `sbyte` , `short` ou, `int` les deux opérandes sont convertis en type `long` .
- Sinon, si l'un des opérandes est de type `uint` , l'autre opérande est converti en type `uint` .
- Sinon, les deux opérandes sont convertis en type `int` .

Notez que la première règle interdit toutes les opérations qui combinent le `decimal` type aux `double` types et `float` . La règle suit du fait qu'il n'y a pas de conversion implicite entre le `decimal` type et `double` les `float` types et.

Notez également qu'il n'est pas possible qu'un opérande soit de type `ulong` lorsque l'autre opérande est d'un type intégral signé. La raison est qu'il n'existe aucun type intégral qui peut représenter la plage complète de `ulong` ainsi que les types intégraux signés.

Dans les deux cas ci-dessus, une expression de cast peut être utilisée pour convertir explicitement un opérande en un type compatible avec l'autre opérande.

Dans l'exemple

```
C#

decimal AddPercent(decimal x, double percent) {
    return x * (1.0 + percent / 100.0);
}
```

une erreur de liaison s'est produite, car un `decimal` ne peut pas être multiplié par un `double` . L'erreur est résolue par la conversion explicite du second opérande en `decimal` , comme suit :

```
C#

decimal AddPercent(decimal x, double percent) {
    return x * (decimal)(1.0 + percent / 100.0);
}
```

Opérateurs levés

Les ***opérateurs levés*** permettent aux opérateurs prédéfinis et définis par l'utilisateur qui fonctionnent sur des types valeur non Nullable d'être également utilisés avec les formulaires Nullable de ces types. Les opérateurs levés sont construits à partir d'opérateurs prédéfinis et définis par l'utilisateur qui répondent à certaines exigences, comme décrit dans les rubriques suivantes :

- Pour les opérateurs unaires

  ```
  C#

  +  ++  -  --  !  ~
  ```

  une forme levée d'un opérateur existe si l'opérande et les types de résultats sont des types valeur non Nullable. Le formulaire levé est construit en ajoutant un `?` modificateur unique à l'opérande et aux types de résultats. L'opérateur levé

produit une valeur null si l'opérande a la valeur null. Dans le cas contraire, l'opérateur Levé désencapsule l'opérande, applique l'opérateur sous-jacent et encapsule le résultat.

- Pour les opérateurs binaires

```
C#

+  -  *  /  %  &  |  ^  <<  >>
```

une forme levée d'un opérateur existe si l'opérande et les types de résultats sont tous des types valeur non Nullable. Le formulaire levé est construit en ajoutant un `?` modificateur unique à chaque opérande et chaque type de résultat. L'opérateur levé produit une valeur null si un ou les deux opérandes ont la valeur null (une exception étant les `&` `|` opérateurs et du `bool?` type, comme décrit dans opérateurs logiques booléens). Dans le cas contraire, l'opérateur Levé désencapsule les opérandes, applique l'opérateur sous-jacent et encapsule le résultat.

- Pour les opérateurs d'égalité

```
C#

==  !=
```

une forme levée d'un opérateur existe si les types d'opérandes sont des types valeur non Nullable et si le type de résultat est `bool` . Le formulaire levé est construit en ajoutant un `?` modificateur unique à chaque type d'opérande. L'opérateur levé considère deux valeurs NULL comme étant égales et une valeur NULL n'est pas égale à une valeur non null. Si les deux opérandes n'ont pas la valeur null, l'opérateur Levé désencapsule les opérandes et applique l'opérateur sous-jacent pour produire le `bool` résultat.

- Pour les opérateurs relationnels

```
C#

<  >  <=  >=
```

une forme levée d'un opérateur existe si les types d'opérandes sont des types valeur non Nullable et si le type de résultat est `bool` . Le formulaire levé est construit en ajoutant un `?` modificateur unique à chaque type d'opérande.

L'opérateur levé produit la valeur `false` si un ou les deux opérandes ont la valeur null. Dans le cas contraire, l'opérateur Levé désencapsule les opérandes et applique l'opérateur sous-jacent pour produire le `bool` résultat.

Recherche de membres

Une recherche de membre est le processus par lequel la signification d'un nom dans le contexte d'un type est déterminée. Une recherche de membre peut se produire dans le cadre de l'évaluation d'un *simple_name* (noms simples) ou d'un *member_access* (accès aux membres) dans une expression. Si le *simple_name* ou *member_access* se produit comme *primary_expression* d'un *invocation_expression* (appels de méthode), le membre est dit appelé.

Si un membre est une méthode ou un événement, ou s'il s'agit d'une constante, d'un champ ou d'une propriété d'un type délégué (délégués) ou du type `dynamic` (type dynamique), le membre est dit *invocable*.

La recherche de membres prend en compte non seulement le nom d'un membre, mais également le nombre de paramètres de type que le membre possède et si le membre est accessible. Dans le cadre de la recherche de membres, les méthodes génériques et les types génériques imbriqués ont le nombre de paramètres de type indiqué dans leurs déclarations respectives et tous les autres membres ont des paramètres de type zéro.

Une recherche de membre d'un nom `N` avec des `K` paramètres de type dans un type `T` est traitée comme suit :

- Tout d'abord, un ensemble de membres accessibles nommés `N` est déterminé :
  - Si `T` est un paramètre de type, le jeu est l'Union des ensembles de membres accessibles nommés `N` dans chacun des types spécifiés en tant que contrainte principale ou contrainte secondaire (contraintes de paramètre de type) pour `T`, ainsi que l'ensemble des membres accessibles nommés `N` dans `object`.
  - Dans le cas contraire, l'ensemble est constitué de tous les membres accessibles (accès aux membres) nommés `N` dans `T`, y compris les membres hérités et les membres accessibles nommés `N` dans `object`. Si `T` est un type construit, l'ensemble des membres est obtenu en remplaçant les arguments de type comme décrit dans membres des types construits. Les membres qui incluent un `override` modificateur sont exclus de l'ensemble.
- Ensuite, si `K` est égal à zéro, tous les types imbriqués dont les déclarations incluent des paramètres de type sont supprimés. Si `K` n'est pas égal à zéro, tous les membres avec un nombre différent de paramètres de type sont supprimés. Notez

que lorsque `K` est égal à zéro, les méthodes ayant des paramètres de type ne sont pas supprimées, car le processus d'inférence de type (inférence de type) peut être en mesure de déduire les arguments de type.

- Ensuite, si le membre est *appelé*, tous les membres qui ne sont pas *invocables* sont supprimés du jeu.
- Ensuite, les membres qui sont masqués par d'autres membres sont supprimés du jeu. Pour chaque membre du `S.M` jeu, où `S` est le type dans lequel le membre `M` est déclaré, les règles suivantes sont appliquées :
  - Si `M` est une constante, un champ, une propriété, un événement ou un membre de l'énumération, tous les membres déclarés dans un type de base de `S` sont supprimés du jeu.
  - Si `M` est une déclaration de type, tous les types non déclarés dans un type de base de `S` sont supprimés du jeu, et toutes les déclarations de type avec le même nombre de paramètres de type `M` déclarés dans un type de base de `S` sont supprimées de l'ensemble.
  - Si `M` est une méthode, tous les membres qui ne sont pas des méthodes déclarés dans un type de base de `S` sont supprimés du jeu.
- Ensuite, les membres d'interface qui sont masqués par les membres de classe sont supprimés du jeu. Cette étape n'a d'effet que si `T` est un paramètre de type et `T` a à la fois une classe de base effective autre que `object` et un ensemble d'interfaces effectif non vide (contraintes de paramètre de type). Pour chaque membre du `S.M` jeu, où `S` est le type dans lequel le membre `M` est déclaré, les règles suivantes sont appliquées si `S` est une déclaration de classe autre que `object` :
  - Si `M` est une constante, un champ, une propriété, un événement, un membre de l'énumération ou une déclaration de type, tous les membres déclarés dans une déclaration d'interface sont supprimés du jeu.
  - Si `M` est une méthode, tous les membres qui ne sont pas des méthodes déclarés dans une déclaration d'interface sont supprimés du jeu, et toutes les méthodes ayant la même signature que celle `M` déclarée dans une déclaration d'interface sont supprimées de l'ensemble.
- Enfin, si vous avez supprimé des membres masqués, le résultat de la recherche est déterminé :
  - Si l'ensemble se compose d'un seul membre qui n'est pas une méthode, ce membre est le résultat de la recherche.
  - Sinon, si l'ensemble contient uniquement des méthodes, ce groupe de méthodes est le résultat de la recherche.
  - Dans le cas contraire, la recherche est ambiguë et une erreur de liaison s'est produite.

Pour les recherches de membres dans les types autres que les paramètres de type et les interfaces, et les recherches de membres dans les interfaces qui sont strictement à héritage simple (chaque interface dans la chaîne d'héritage a exactement zéro ou une interface de base directe), l'effet des règles de recherche est simplement que les membres dérivés masquent les membres de base avec le même nom ou la même signature. Ces recherches à héritage unique ne sont jamais ambiguës. Les ambiguïtés qui peuvent se produire à partir de recherches de membres dans des interfaces à héritage multiple sont décrites dans accès aux membres d'interface.

Types de base

Dans le cadre de la recherche de membres, un type `T` est considéré comme ayant les types de base suivants :

- Si `T` est `object` , n' `T` a aucun type de base.
- Si `T` est un *enum_type*, les types de base de `T` sont les types de classe `System.Enum` , `System.ValueType` et `object` .
- Si `T` est un *struct_type*, les types de base de `T` sont les types de classe `System.ValueType` et `object` .
- Si `T` est un *class_type*, les types de base de `T` sont les classes de base de `T` , y compris le type de classe `object` .
- Si `T` est un *INTERFACE_TYPE*, les types de base de `T` sont les interfaces de base de `T` et le type de classe `object` .
- Si `T` est un *array_type*, les types de base de `T` sont les types de classe `System.Array` et `object` .
- Si `T` est un *delegate_type*, les types de base de `T` sont les types de classe `System.Delegate` et `object` .

Fonctions membres

Les fonctions membres sont des membres qui contiennent des instructions exécutables. Les membres de fonction sont toujours membres de types et ne peuvent pas être membres d'espaces de noms. C# définit les catégories de fonctions membres suivantes :

- Méthodes
- Propriétés
- Événements
- Indexeurs
- Opérateurs définis par l'utilisateur
- Constructeurs d'instance

- Constructeurs statiques
- Destructeurs

À l'exception des destructeurs et des constructeurs statiques (qui ne peuvent pas être appelés explicitement), les instructions contenues dans les fonctions membres sont exécutées par le biais d'appels de membres de fonction. La syntaxe réelle pour l'écriture d'un appel de membre de fonction dépend de la catégorie de membre de fonction particulière.

La liste d'arguments (listes d'arguments) d'un appel de membre de fonction fournit des valeurs réelles ou des références de variables pour les paramètres de la fonction membre.

Les appels de méthodes génériques peuvent utiliser l'inférence de type pour déterminer le jeu d'arguments de type à passer à la méthode. Ce processus est décrit dans inférence de type.

Les appels de méthodes, d'indexeurs, d'opérateurs et de constructeurs d'instance utilisent la résolution de surcharge pour déterminer l'ensemble de candidats des membres de fonction à appeler. Ce processus est décrit dans résolution de surcharge.

Une fois qu'un membre de fonction particulier a été identifié au moment de la liaison, éventuellement par la résolution de surcharge, le processus d'exécution réel de l'appel de la fonction membre est décrit dans vérification de la résolution de surcharge dynamique au momentde la compilation.

Le tableau suivant résume le traitement qui a lieu dans les constructions impliquant les six catégories de fonctions membres qui peuvent être explicitement appelées. Dans le tableau, `e` , `x` , `y` et `value` indiquent des expressions classifiées comme variables ou valeurs, `T` indique une expression classifiée comme un type, `F` est le nom simple d'une méthode et `P` est le nom simple d'une propriété.

Composer	Exemple	Description
Appel de méthode	`F(x,y)`	La résolution de surcharge est appliquée pour sélectionner la meilleure méthode `F` dans la classe ou le struct conteneur. La méthode est appelée avec la liste d'arguments `(x,y)` . Si la méthode n'est pas `static` , l'expression d'instance est `this` .
	`T.F(x,y)`	La résolution de surcharge est appliquée pour sélectionner la meilleure méthode `F` dans la classe ou le struct `T` . Une erreur de liaison au moment de la liaison se produit si la méthode n'est pas `static` . La méthode est appelée avec la liste d'arguments `(x,y)` .

Composer	Exemple	Description
	`e.F(x,y)`	La résolution de surcharge est appliquée pour sélectionner la meilleure méthode F dans la classe, la structure ou l'interface donnée par le type de `e` . Une erreur de liaison au moment de la liaison se produit si la méthode est `static` . La méthode est appelée avec l'expression d'instance `e` et la liste d'arguments `(x,y)` .
Accès à la propriété	`P`	L' `get` accesseur de la propriété `P` dans la classe ou le struct conteneur est appelé. Une erreur de compilation se produit si `P` est en écriture seule. Si `P` n'est pas `static` , l'expression d'instance est `this` .
	`P = value`	L' `set` accesseur de la propriété `P` dans la classe ou le struct conteneur est appelé avec la liste d'arguments `(value)` . Une erreur de compilation se produit si `P` est en lecture seule. Si `P` n'est pas `static` , l'expression d'instance est `this` .
	`T.P`	L' `get` accesseur de la propriété `P` dans la classe ou le struct `T` est appelé. Une erreur de compilation se produit si `P` n'est pas `static` ou si est `P` en écriture seule.
	`T.P = value`	L' `set` accesseur de la propriété `P` dans la classe ou le struct `T` est appelé avec la liste d'arguments `(value)` . Une erreur de compilation se produit si `P` n'est pas `static` ou si `P` est en lecture seule.
	`e.P`	L' `get` accesseur de la propriété `P` dans la classe, la structure ou l'interface donnée par le type de `e` est appelé avec l'expression d'instance `e` . Une erreur de liaison au moment de la liaison se produit si `P` est `static` ou si `P` est en écriture seule.
	`e.P = value`	L' `set` accesseur de la propriété `P` dans la classe, la structure ou l'interface donnée par le type de `e` est appelé avec l'expression d'instance `e` et la liste d'arguments `(value)` . Une erreur de liaison au moment de la liaison se produit si `P` est `static` ou si `P` est en lecture seule.
Accès aux événements	`E += value`	L' `add` accesseur de l'événement `E` dans la classe ou le struct conteneur est appelé. Si `E` n'est pas statique, l'expression d'instance est `this` .
	`E -= value`	L' `remove` accesseur de l'événement `E` dans la classe ou le struct conteneur est appelé. Si `E` n'est pas statique, l'expression d'instance est `this` .
	`T.E += value`	L' `add` accesseur de l'événement `E` dans la classe ou le struct `T` est appelé. Une erreur de liaison au moment de la liaison se produit si `E` n'est pas statique.

Composer	Exemple	Description
	`T.E -= value`	L' `remove` accesseur de l'événement `E` dans la classe ou le struct `T` est appelé. Une erreur de liaison au moment de la liaison se produit si `E` n'est pas statique.
	`e.E += value`	L' `add` accesseur de l'événement `E` dans la classe, la structure ou l'interface donnée par le type de `e` est appelé avec l'expression d'instance `e` . Une erreur de liaison au moment de la liaison se produit si `E` est statique.
	`e.E -= value`	L' `remove` accesseur de l'événement `E` dans la classe, la structure ou l'interface donnée par le type de `e` est appelé avec l'expression d'instance `e` . Une erreur de liaison au moment de la liaison se produit si `E` est statique.
Accès aux indexeurs	`e[x,y]`	La résolution de surcharge est appliquée pour sélectionner le meilleur indexeur de la classe, du struct ou de l'interface donné par le type de e. L' `get` accesseur de l'indexeur est appelé avec l'expression d'instance `e` et la liste d'arguments `(x,y)` . Une erreur de liaison au moment de la liaison se produit si l'indexeur est en écriture seule.
	`e[x,y] = value`	La résolution de surcharge est appliquée pour sélectionner le meilleur indexeur dans la classe, la structure ou l'interface donnée par le type de `e` . L' `set` accesseur de l'indexeur est appelé avec l'expression d'instance `e` et la liste d'arguments `(x,y,value)` . Une erreur de liaison s'est produite si l'indexeur est en lecture seule.
Appel d'opérateur	`-x`	La résolution de surcharge est appliquée pour sélectionner le meilleur opérateur unaire dans la classe ou le struct donné par le type de `x` . L'opérateur sélectionné est appelé avec la liste d'arguments `(x)` .
	`x + y`	La résolution de surcharge est appliquée pour sélectionner le meilleur opérateur binaire dans les classes ou les structs spécifiés par les types de `x` et `y` . L'opérateur sélectionné est appelé avec la liste d'arguments `(x,y)` .
Appel du constructeur d'instance	`new T(x,y)`	La résolution de surcharge est appliquée pour sélectionner le constructeur d'instance le mieux adapté à la classe ou au struct `T` . Le constructeur d'instance est appelé avec la liste d'arguments `(x,y)` .

Listes d'arguments

Chaque membre de fonction et appel de délégué inclut une liste d'arguments qui fournit des valeurs réelles ou des références de variables pour les paramètres de la fonction membre. La syntaxe permettant de spécifier la liste d'arguments d'un appel de membre de fonction dépend de la catégorie de membre de fonction :

- Pour les constructeurs d'instance, les méthodes, les indexeurs et les délégués, les arguments sont spécifiés en tant que *argument_list*, comme décrit ci-dessous. Pour les indexeurs, lors de l'appel de l' `set` accesseur, la liste d'arguments comprend également l'expression spécifiée comme opérande droit de l'opérateur d'assignation.
- Pour les propriétés, la liste d'arguments est vide lors de l'appel de l' `get` accesseur, et se compose de l'expression spécifiée comme opérande droit de l'opérateur d'assignation lors de l'appel de l' `set` accesseur.
- Pour les événements, la liste d'arguments se compose de l'expression spécifiée comme opérande droit de l' `+=` `-=` opérateur or.
- Pour les opérateurs définis par l'utilisateur, la liste d'arguments se compose du seul opérande de l'opérateur unaire ou des deux opérandes de l'opérateur binaire.

Les arguments des propriétés (Propriétés), événements (événements) et opérateurs définis par l'utilisateur (opérateurs) sont toujours passés en tant que paramètres de valeur (paramètres de valeur). Les arguments des indexeurs (indexeurs) sont toujours passés en tant que paramètres de valeur (paramètres de valeur) ou tableaux de paramètres (tableaux de paramètres). Les paramètres de référence et de sortie ne sont pas pris en charge pour ces catégories de membres de fonction.

Les arguments d'un constructeur d'instance, d'une méthode, d'un indexeur ou d'un appel de délégué sont spécifiés en tant que *argument_list*:

antlr

```
argument_list
    : argument (',' argument)*
    ;

argument
    : argument_name? argument_value
    ;

argument_name
    : identifier ':'
    ;

argument_value
    : expression
    | 'ref' variable_reference
    | 'out' variable_reference
    ;
```

Un *argument_list* se compose d'un ou plusieurs *arguments*, séparés par des virgules. Chaque argument se compose d'un *argument_name* facultatif suivi d'un

argument_value. Un *argument* avec un *argument_name* est appelé un ***argument nommé** , *alors qu'un argument * sans argument_name est un **argument * positionnel*.*. L'apparition d'un argument positionnel après un argument nommé dans un _argument_list * constitue une erreur.

La *argument_value* peut prendre l'une des formes suivantes :

- *Expression* indiquant que l'argument est passé en tant que paramètre de valeur (paramètres de valeur).
- Mot clé `ref` suivi d'un *variable_reference* (références de variable), indiquant que l'argument est passé en tant que paramètre de référence (paramètres de référence). Une variable doit être assignée de manière définitive (assignation définie) avant de pouvoir être passée en tant que paramètre de référence. Mot clé `out` suivi d'un *variable_reference* (références de variable), indiquant que l'argument est passé en tant que paramètre de sortie (paramètres de sortie). Une variable est considérée comme assignée définitivement (assignation définie) à la suite d'un appel de membre de fonction dans lequel la variable est passée comme paramètre de sortie.

Paramètres correspondants

Pour chaque argument dans une liste d'arguments, il doit y avoir un paramètre correspondant dans le membre de fonction ou le délégué appelé.

La liste de paramètres utilisée dans les éléments suivants est déterminée comme suit :

- Pour les méthodes virtuelles et les indexeurs définis dans les classes, la liste de paramètres est choisie à partir de la déclaration ou de la substitution la plus spécifique, en commençant par le type statique du récepteur, et en recherchant dans ses classes de base.
- Pour les méthodes d'interface et les indexeurs, la liste de paramètres est choisie en fonction de la définition la plus spécifique du membre, en commençant par le type d'interface et en parcourant les interfaces de base. Si aucune liste de paramètres unique n'est trouvée, une liste de paramètres avec des noms inaccessibles et aucun paramètre facultatif n'est construite, de sorte que les appels ne peuvent pas utiliser de paramètres nommés ou omettre des arguments facultatifs.
- Pour les méthodes partielles, la liste de paramètres de la déclaration de méthode partielle de définition est utilisée.
- Pour tous les autres membres de fonction et délégués, il n'existe qu'une seule liste de paramètres, qui est celle utilisée.

La position d'un argument ou d'un paramètre est définie en tant que nombre d'arguments ou de paramètres qui la précèdent dans la liste d'arguments ou la liste de paramètres.

Les paramètres correspondants pour les arguments de membre de fonction sont établis comme suit :

- Arguments dans le *argument_list* de constructeurs d'instance, de méthodes, d'indexeurs et de délégués :
  - Un argument positionnel dans lequel un paramètre fixe se produit à la même position dans la liste de paramètres correspond à ce paramètre.
  - Un argument positionnel d'un membre de fonction avec un tableau de paramètres appelé dans sa forme normale correspond au tableau de paramètres, qui doit se trouver à la même position dans la liste de paramètres.
  - Argument positionnel d'un membre de fonction avec un tableau de paramètres appelé dans sa forme développée, où aucun paramètre fixe ne se trouve à la même position dans la liste de paramètres, correspond à un élément dans le tableau de paramètres.
  - Un argument nommé correspond au paramètre du même nom dans la liste de paramètres.
  - Pour les indexeurs, lors de l'appel de l' `set` accesseur, l'expression spécifiée comme opérande droit de l'opérateur d'assignation correspond au paramètre implicite `value` de la `set` déclaration d'accesseur.
- Pour les propriétés, lors de l'appel de l' `get` accesseur, il n'y a pas d'arguments. Lors de l'appel de l' `set` accesseur, l'expression spécifiée comme opérande droit de l'opérateur d'assignation correspond au paramètre implicite `value` de la `set` déclaration d'accesseur.
- Pour les opérateurs unaires définis par l'utilisateur (y compris les conversions), l'opérande unique correspond au paramètre unique de la déclaration d'opérateur.
- Pour les opérateurs binaires définis par l'utilisateur, l'opérande de gauche correspond au premier paramètre, tandis que l'opérande droit correspond au deuxième paramètre de la déclaration de l'opérateur.

Évaluation au moment de l'exécution des listes d'arguments

Pendant le traitement au moment de l'exécution d'un appel de membre de fonction (vérification au moment de la compilation de la résolution de surcharge dynamique), les expressions ou les références de variable d'une liste d'arguments sont évaluées dans l'ordre, de gauche à droite, comme suit :

- Pour un paramètre de valeur, l'expression d'argument est évaluée et une conversion implicite (conversions implicites) en type de paramètre correspondant est effectuée. La valeur résultante devient la valeur initiale du paramètre de valeur dans l'appel de membre de fonction.
- Pour un paramètre de référence ou de sortie, la référence de variable est évaluée et l'emplacement de stockage résultant devient l'emplacement de stockage représenté par le paramètre dans l'appel de membre de fonction. Si la référence variable donnée en tant que paramètre de référence ou de sortie est un élément de tableau d'un *reference_type*, un contrôle au moment de l'exécution est effectué pour garantir que le type d'élément du tableau est identique au type du paramètre. Si cette vérification échoue, une `System.ArrayTypeMismatchException` exception est levée.

Les méthodes, les indexeurs et les constructeurs d'instance peuvent déclarer leur paramètre le plus à droite comme un tableau de paramètres (tableaux de paramètres). Ces fonctions sont appelées soit sous leur forme normale, soit sous leur forme développée en fonction de ce qui est applicable (fonction membre applicable) :

- Lorsqu'un membre de fonction avec un tableau de paramètres est appelé dans sa forme normale, l'argument donné pour le tableau de paramètres doit être une expression unique qui est implicitement convertible (conversions implicites) en type de tableau de paramètres. Dans ce cas, le tableau de paramètres agit précisément comme un paramètre de valeur.
- Lorsqu'un membre de fonction avec un tableau de paramètres est appelé dans sa forme développée, l'appel doit spécifier zéro, un ou plusieurs arguments positionnels pour le tableau de paramètres, où chaque argument est une expression implicitement convertible (conversions implicites) en type d'élément du tableau de paramètres. Dans ce cas, l'appel crée une instance du type de tableau de paramètres avec une longueur correspondant au nombre d'arguments, initialise les éléments de l'instance de tableau avec les valeurs d'argument données et utilise l'instance de tableau nouvellement créée comme argument réel.

Les expressions d'une liste d'arguments sont toujours évaluées dans l'ordre dans lequel elles sont écrites. Ainsi, l'exemple

C#

```
class Test
{
    static void F(int x, int y = -1, int z = -2) {
        System.Console.WriteLine("x = {0}, y = {1}, z = {2}", x, y, z);
    }

    static void Main() {
```

```
        int i = 0;
        F(i++, i++, i++);
        F(z: i++, x: i++);
    }
}
```

génère la sortie

```
Console

x = 0, y = 1, z = 2
x = 4, y = -1, z = 3
```

Les règles de covariance de tableau (covariance de tableau) permettent à une valeur d'un type de tableau `A[]` d'être une référence à une instance d'un type de tableau `B[]` , à condition qu'il existe une conversion de référence implicite de `B` en `A` . En raison de ces règles, lorsqu'un élément de tableau d'un *reference_type* est passé en tant que paramètre de référence ou de sortie, un contrôle au moment de l'exécution est nécessaire pour garantir que le type d'élément réel du tableau est identique à celui du paramètre. Dans l'exemple

```
C#

class Test
{
    static void F(ref object x) {...}

    static void Main() {
        object[] a = new object[10];
        object[] b = new string[10];
        F(ref a[0]);        // Ok
        F(ref b[1]);        // ArrayTypeMismatchException
    }
}
```

le deuxième appel de `F` lève une `System.ArrayTypeMismatchException` exception, car le type d'élément réel de `b` est `string` et non `object` .

Lorsqu'un membre de fonction avec un tableau de paramètres est appelé dans sa forme développée, l'appel est traité exactement comme si une expression de création de tableau avec un initialiseur de tableau (expressions de création de tableau) a été insérée autour des paramètres développés. Par exemple, étant donné la déclaration

```
C#

void F(int x, int y, params object[] args);
```

les appels suivants de la forme développée de la méthode

```C#
F(10, 20);
F(10, 20, 30, 40);
F(10, 20, 1, "hello", 3.0);
```

correspondre exactement à

```C#
F(10, 20, new object[] {});
F(10, 20, new object[] {30, 40});
F(10, 20, new object[] {1, "hello", 3.0});
```

En particulier, Notez qu'un tableau vide est créé lorsqu'il n'y a aucun argument donné pour le tableau de paramètres.

Lorsque des arguments sont omis d'un membre de fonction avec des paramètres facultatifs correspondants, les arguments par défaut de la déclaration de membre de fonction sont passés implicitement. Étant donné qu'elles sont toujours constantes, leur évaluation n'aura pas d'impact sur l'ordre d'évaluation des arguments restants.

Inférence de type

Quand une méthode générique est appelée sans spécifier d'arguments de type, un processus d' ***inférence de type*** tente de déduire des arguments de type pour l'appel. La présence de l'inférence de type permet d'utiliser une syntaxe plus pratique pour appeler une méthode générique et permet au programmeur d'éviter de spécifier des informations de type redondantes. Par exemple, à partir de la déclaration de méthode :

```C#
class Chooser
{
    static Random rand = new Random();

    public static T Choose<T>(T first, T second) {
        return (rand.Next(2) == 0)? first: second;
    }
}
```

Il est possible d'appeler la `Choose` méthode sans spécifier explicitement un argument de type :

```
C#

int i = Chooser.Choose(5, 213);                 // Calls Choose<int>

string s = Chooser.Choose("foo", "bar");        // Calls Choose<string>
```

Par le biais de l'inférence de type, les arguments `int` de type et `string` sont déterminés à partir des arguments à la méthode.

L'inférence de type se produit dans le cadre du traitement au moment de la liaison d'un appel de méthode (appel de méthode) et a lieu avant l'étape de résolution de surcharge de l'appel. Lorsqu'un groupe de méthodes particulier est spécifié dans un appel de méthode et qu'aucun argument de type n'est spécifié dans le cadre de l'appel de méthode, l'inférence de type est appliquée à chaque méthode générique dans le groupe de méthodes. Si l'inférence de type est réussie, les arguments de type déduits sont utilisés pour déterminer les types d'arguments pour la résolution de surcharge suivante. Si la résolution de surcharge choisit une méthode générique comme une méthode à appeler, les arguments de type inférés sont utilisés en tant qu'arguments de type réels pour l'appel. Si l'inférence de type pour une méthode particulière échoue, cette méthode ne participe pas à la résolution de surcharge. L'échec de l'inférence de type, en soi-même, ne provoque pas d'erreur au moment de la liaison. Toutefois, il arrive souvent à une erreur de liaison lorsque la résolution de surcharge ne parvient pas à trouver les méthodes applicables.

Si le nombre d'arguments fourni est différent du nombre de paramètres dans la méthode, l'inférence échoue immédiatement. Dans le cas contraire, supposez que la méthode générique possède la signature suivante :

```
C#

Tr M<X1,...,Xn>(T1 x1, ..., Tm xm)
```

Avec un appel de méthode de la forme `M(E1...Em)` , la tâche de l'inférence de type consiste à trouver des arguments de type uniques `S1...Sn` pour chacun des paramètres de type `X1...Xn` afin que l'appel `M<S1...Sn>(E1...Em)` devienne valide.

Pendant le processus d'inférence, chaque paramètre de type `Xi` est *fixé* à un type particulier `Si` ou non *résolu* avec un ensemble associé de *limites*. Chacune des limites

est un type `T` . Initialement, chaque variable `Xi` de type n'est pas corrigée avec un ensemble de limites vide.

L'inférence de type a lieu en plusieurs phases. Chaque phase essaiera de déduire les arguments de type pour d'autres variables de type en fonction des résultats de la phase précédente. La première phase effectue quelques inférences initiales des limites, tandis que la deuxième phase résout des variables de type à des types spécifiques et déduit d'autres limites. La deuxième phase peut être répétée plusieurs fois.

Remarque : L'inférence de type a lieu non seulement quand une méthode générique est appelée. L'inférence de type pour la conversion de groupes de méthodes est décrite dans inférence de type pour la conversion de groupes de méthodes et la recherche du meilleur type commun d'un ensemble d'expressions est décrite dans recherche du meilleur type commun d'un ensemble d'expressions.

La première phase

Pour chacun des arguments de la méthode `Ei` :

- Si `Ei` est une fonction anonyme, une *inférence de type de paramètre explicite* (inférences de type de paramètre explicite) est apportée de `Ei` à `Ti`
- Sinon, si `Ei` a un type `U` et qu' `xi` il s'agit d'un paramètre de valeur, une *inférence de liaison inférieure* est faite *de* `U` *à* `Ti` .
- Sinon, si `Ei` a un type `U` et qu' `xi` il s'agit d'un `ref` paramètre ou, `out` une *inférence exacte* est faite *de* `U` *à* `Ti` .
- Sinon, aucune inférence n'est effectuée pour cet argument.

La deuxième phase

La deuxième phase se déroule comme suit :

- Toutes les variables de type `Xi` non fixes qui ne *dépendent pas de* (dépendance) `Xj` sont fixes (Correction).
- S'il n'existe aucune variable de type de ce type, toutes les variables de type non *fixes* `Xi` sont *fixes* pour lesquelles tous les éléments suivants sont détenus :
  - Au moins une variable de type `Xj` dépend de `Xi`
  - `Xi` a un ensemble non vide de limites
- S'il n'existe aucune variable de type de ce type et qu'il existe encore des variables de type non *fixes* , l'inférence de type échoue.
- Sinon, s'il n'existe aucune variable de type non *fixe* , l'inférence de type est réussie.

- Dans le cas contraire, pour tous les arguments `Ei` ayant le type de paramètre correspondant, `Ti` où les *types de sortie* (types de sortie) contiennent des variables de type non *fixes* `Xj` , contrairement aux types *d'entrée* (types d'entrée), une *inférence de type de sortie* (inférences de type de sortie) est effectuée *de* `Ei` *à* `Ti` . La deuxième phase est ensuite répétée.

Types d'entrée

Si `E` est un groupe de méthodes ou une fonction anonyme implicitement typée et `T` est un type délégué ou un type d'arborescence d'expression, tous les types de paramètres de `T` sont des *types d'entrée* de `E` *avec le type* `T` .

Types de sorties

Si `E` est un groupe de méthodes ou une fonction anonyme et `T` est un type délégué ou un type d'arborescence d'expression, le type de retour de `T` est un *type de sortie* `E` *avec le type* `T` .

Dépendance

Une variable de type non *fixed* `Xi` *dépend directement* d'une variable de type non fixe `Xj` si pour un argument de `Ek` type `Tk` `Xj` se produit dans un *type d'entrée* de `Ek` type `Tk` et `Xi` se produit dans un *type de sortie* de `Ek` type `Tk` .

`Xj` *dépend de* `Xi` Si `Xj` *dépend directement de* `Xi` ou si `Xi` *dépend de* `Xk` et dépend `Xk` *de* `Xj` . Par conséquent, « dépend de » est la fermeture transitive mais non réflexive de « dépend directement de ».

Inférences de type de sortie

Une *inférence de type de sortie* est effectuée *d'* une expression `E` *à* un type `T` de la façon suivante :

- Si `E` est une fonction anonyme avec le type de retour `U` déduit (type de retour déduit) et `T` est un type délégué ou un type d'arborescence d'expression avec le type de retour `Tb` , une *inférence à la limite inférieure* (inférences à liaison inférieure) est effectuée à *partir* de `U` *en* `Tb` .
- Sinon, si `E` est un groupe de méthodes et qu' `T` il s'agit d'un type délégué ou d'un type d'arborescence d'expression avec des types de paramètres `T1...Tk` et un type

de retour `Tb` , et que la résolution de surcharge de `E` avec les types `T1...Tk` génère une méthode unique avec le type de retour `U` , une *inférence de liaison inférieure* est effectuée *de* `U` *à* `Tb` .

- Sinon, si `E` est une expression de type `U` , une *inférence de liaison inférieure* est faite *de* `U` *à* `T` .
- Dans le cas contraire, aucune inférence n'est effectuée.

Inférences de type de paramètre explicites

Une *inférence de type de paramètre explicite* est effectuée *d*' une expression `E` *à* un type `T` de la façon suivante :

- Si `E` est une fonction anonyme explicitement typée avec des types de paramètres `U1...Uk` et `T` est un type délégué ou un type d'arborescence d'expression avec des types `V1...Vk` de paramètres, pour chaque `Ui` *inférence exacte* (inférences exactes) est effectuée à *partir* de `Ui` *vers* le correspondant `Vi` .

Inférences exactes

Une *inférence exacte d*' un type `U` *à* un type `V` est effectuée comme suit :

- Si `V` est l'un des non *résolus* `Xi` , `U` est ajouté au jeu de limites exactes pour `Xi` .
- Dans le cas contraire, les paramètres `V1...Vk` et `U1...Uk` sont déterminés en vérifiant si l'un des cas suivants s'applique :
  - `V` est un type tableau `V1[...]` et `U` est un type tableau `U1[...]` du même rang
  - `V` est le type `V1?` et `U` est le type `U1?`
  - `V` est un type construit `C<V1...Vk>` et `U` est un type construit `C<U1...Uk>`

  Si l'un de ces cas s'applique, une *inférence exacte* est effectuée à *partir de* chaque `Ui` *vers* le correspondant `Vi` .
- Sinon, aucune inférence n'est effectuée.

Inférences à limite inférieure

Une *inférence de liaison inférieure d*' un type `U` *à* un type `V` est effectuée comme suit :

- Si `V` est l'un des non *résolus* `Xi` , `U` est ajouté au jeu de limites inférieures pour `Xi` .

- Sinon, si `V` est le type `V1?` et `U` est le type, `U1?` une inférence de limite inférieure est faite de `U1` à `V1` .

- Dans le cas contraire, les paramètres `U1...Uk` et `V1...Vk` sont déterminés en vérifiant si l'un des cas suivants s'applique :

  - `V` est un type tableau `V1[...]` et `U` est un type tableau `U1[...]` (ou un paramètre de type dont le type de base effectif est `U1[...]`) du même rang

  - `V` est l'un `IEnumerable<V1>` des `ICollection<V1>` types, ou `IList<V1>` et `U` est un type tableau unidimensionnel `U1[]` (ou un paramètre de type dont le type de base effectif est `U1[]`)

  - `V` est une classe, un struct, une interface ou un type délégué construit, `C<V1...Vk>` et il existe un type unique `C<U1...Uk>` tel que `U` (ou, si `U` est un paramètre de type, sa classe de base effective ou tout membre de son ensemble d'interfaces effectifs) est identique à, hérite de (directement ou indirectement) ou implémente (directement ou indirectement) `C<U1...Uk>` .

    (La restriction « unicité » signifie que dans l'interface de cas `C<T> {} class U: C<X>, C<Y> {}` , aucune inférence n'est effectuée lors de la déduction de `U` en `C<T>` , car `U1` peut être `X` ou `Y` .)

  Si l'un de ces cas s'applique, une inférence est effectuée *à partir de* chaque `Ui` *vers* le correspondant `Vi` , comme suit :
  - Si `Ui` n'est pas connu comme étant un type référence, une *inférence exacte* est effectuée
  - Sinon, si `U` est un type tableau, une *inférence de liaison inférieure* est effectuée
  - Sinon, si `V` est `C<V1...Vk>` Then, l'inférence dépend du paramètre de type i-Th de `C` :
    - S'il s'agit d'un covariant, une *inférence de liaison inférieure* est effectuée.
    - S'il est contravariant, une *inférence de* la limite supérieure est effectuée.
    - S'il s'agit d'un invariant, une *inférence exacte* est effectuée.

- Dans le cas contraire, aucune inférence n'est effectuée.

Inférences à la limite supérieure

Une *inférence* à la limite supérieure *d*' un type `U` *à* un type `V` est effectuée comme suit :

- Si `V` est l'un des non *résolus* `Xi` , `U` est ajouté au jeu de limites supérieures pour `Xi` .

- Dans le cas contraire, les paramètres `V1...Vk` et `U1...Uk` sont déterminés en vérifiant si l'un des cas suivants s'applique :
  - `U` est un type tableau `U1[...]` et `V` est un type tableau `V1[...]` du même rang
  - `U` est l'un `IEnumerable<Ue>` des `ICollection<Ue>` types, ou `IList<Ue>` et `V` est un type tableau unidimensionnel `Ve[]`
  - `U` est le type `U1?` et `V` est le type `V1?`
  - `U` est une classe, un struct, une interface ou un type délégué construit `C<U1...Uk>` et `V` est une classe, un struct, une interface ou un type délégué qui est identique à, hérite de (directement ou indirectement), ou implémente (directement ou indirectement) un type unique `C<V1...Vk>`

    (La restriction « unicité » signifie que si nous en avons une `interface C<T>{}` `class V<Z>: C<X<Z>>, C<Y<Z>>{}` , aucune inférence n'est effectuée lors de l'inférence de `C<U1>` à `V<Q>` . Les inférences ne sont pas effectuées à partir de `U1` vers `X<Q>` ou `Y<Q>` .)

  Si l'un de ces cas s'applique, une inférence est effectuée *à partir de* chaque `Ui` *vers* le correspondant `Vi` , comme suit :
  - Si `Ui` n'est pas connu comme étant un type référence, une *inférence exacte* est effectuée
  - Sinon, si `V` est un type tableau, une *inférence de* la limite supérieure est effectuée
  - Sinon, si `U` est `C<U1...Uk>` Then, l'inférence dépend du paramètre de type i-Th de `C` :
    - S'il s'agit d'un covariant, une *inférence de* la limite supérieure est effectuée.
    - S'il est contravariant, une *inférence de liaison inférieure* est effectuée.
    - S'il s'agit d'un invariant, une *inférence exacte* est effectuée.
- Dans le cas contraire, aucune inférence n'est effectuée.

Résolution

Une variable de type non *fixed* `Xi` avec un ensemble de limites est *fixe* comme suit :

- L'ensemble de *types candidats* `Uj` démarre en tant que jeu de tous les types dans le jeu de limites de `Xi` .
- Nous examinons ensuite chaque limite pour à `Xi` son tour : pour chaque limite exacte `U` de `Xi` tous les types `Uj` qui ne sont pas identiques à qui sont `U`

supprimés du jeu de candidats. Pour chaque limite inférieure `U` de `Xi` tous les types `Uj` vers lesquels il n'y a *pas* de conversion implicite de `U` sont supprimés du jeu de candidats. Pour chaque limite supérieure `U` de `Xi` tous les types `Uj` à partir desquels il n'existe *pas* de conversion implicite vers `U` , est supprimé de l'ensemble de candidats.

- Si, parmi les types candidats restants `Uj` , il existe un type unique `V` à partir duquel il existe une conversion implicite vers tous les autres types candidats, `Xi` est fixe à `V` .
- Sinon, l'inférence de type échoue.

Type de retour déduit

Le type de retour déduit d'une fonction anonyme `F` est utilisé pendant l'inférence de type et la résolution de surcharge. Le type de retour déduit peut uniquement être déterminé pour une fonction anonyme dans laquelle tous les types de paramètres sont connus, soit parce qu'ils sont explicitement donnés, fournis par le biais d'une conversion de fonction anonyme ou déduits pendant l'inférence de type sur un appel de méthode générique englobant.

Le ***type de résultat inféré*** est déterminé comme suit :

- Si le corps de `F` est une *expression* qui a un type, le type de résultat déduit de `F` est le type de cette expression.
- Si le corps de `F` est un *bloc* et que l'ensemble d'expressions dans les instructions du bloc `return` a un type commun le mieux `T` (recherche du meilleur type commun d'un ensemble d'expressions), le type de résultat déduit de `F` est `T` .
- Dans le cas contraire, un type de résultat ne peut pas être déduit pour `F` .

Le ***type de retour déduit*** est déterminé comme suit :

- Si `F` est Async et que le corps de `F` est soit une expression classifiée comme Nothing (classifications d'expression), soit un bloc d'instructions où aucune instruction return n'a d'expressions, le type de retour déduit est `System.Threading.Tasks.Task`
- Si `F` est Async et a un type de résultat déduit `T` , le type de retour déduit est `System.Threading.Tasks.Task<T>` .
- Si `F` est non asynchrone et a un type de résultat déduit `T` , le type de retour déduit est `T` .
- Dans le cas contraire, un type de retour ne peut pas être déduit pour `F` .

En guise d'exemple d'inférence de type impliquant des fonctions anonymes, considérez la `Select` méthode d'extension déclarée dans la `System.Linq.Enumerable` classe :

C#

```
namespace System.Linq
{
    public static class Enumerable
    {
        public static IEnumerable<TResult> Select<TSource,TResult>(
            this IEnumerable<TSource> source,
            Func<TSource,TResult> selector)
        {
            foreach (TSource element in source) yield return
selector(element);
        }
    }
}
```

En supposant que l' `System.Linq` espace de noms a été importé avec une `using` clause et une classe `Customer` avec une `Name` propriété de type `string` , la `Select` méthode peut être utilisée pour sélectionner les noms d'une liste de clients :

C#

```
List<Customer> customers = GetCustomerList();
IEnumerable<string> names = customers.Select(c => c.Name);
```

L'appel de méthode d'extension (appels de méthode d'extension) de `Select` est traité en réécrivant l'appel à un appel de méthode statique :

C#

```
IEnumerable<string> names = Enumerable.Select(customers, c => c.Name);
```

Étant donné que les arguments de type n'ont pas été spécifiés explicitement, l'inférence de type est utilisée pour déduire les arguments de type. Tout d'abord, l' `customers` argument est lié au `source` paramètre, en déduisant que la `T` valeur est `Customer` . Ensuite, à l'aide du processus d'inférence de type de fonction anonyme décrit ci-dessus, `c` le type est donné `Customer` et l'expression `c.Name` est associée au type de retour du `selector` paramètre, ce qui déduit que la `S` valeur est `string` . Ainsi, l'appel est équivalent à

C#

```
Sequence.Select<Customer,string>(customers, (Customer c) => c.Name)
```

et le résultat est de type `IEnumerable<string>` .

L'exemple suivant montre comment l'inférence de type de fonction anonyme permet aux informations de type de « circuler » entre les arguments dans un appel de méthode générique. À partir de la méthode :

C#

```
static Z F<X,Y,Z>(X value, Func<X,Y> f1, Func<Y,Z> f2) {
    return f2(f1(value));
}
```

Inférence de type pour l'appel :

C#

```
double seconds = F("1:15:30", s => TimeSpan.Parse(s), t => t.TotalSeconds);
```

continue comme suit : tout d'abord, l'argument `"1:15:30"` est lié au `value` paramètre, déduire ainsi la `X` valeur `string` . Ensuite, le paramètre de la première fonction anonyme, `s` , reçoit le type inféré `string` et l'expression `TimeSpan.Parse(s)` est liée au type de retour de `f1` , en déduisant que la `Y` valeur est `System.TimeSpan` . Enfin, le paramètre de la deuxième fonction anonyme, `t` , reçoit le type inféré `System.TimeSpan` et l'expression `t.TotalSeconds` est liée au type de retour de `f2` , en déduisant que la `Z` valeur est `double` . Ainsi, le résultat de l'appel est de type `double` .

Inférence de type pour la conversion de groupes de méthodes

Comme pour les appels de méthodes génériques, l'inférence de type doit également être appliquée lorsqu'un groupe de méthodes `M` contenant une méthode générique est converti en un type délégué donné `D` (conversions de groupe de méthodes). Pour une méthode donnée

C#

```
Tr M<X1...Xn>(T1 x1 ... Tm xm)
```

et le groupe `M` de méthodes affecté au type délégué `D` la tâche de l'inférence de type consiste à rechercher des arguments de type `S1...Sn` afin que l'expression :

```
C#

M<S1...Sn>
```

devient compatible (déclarations déléguées) avec `D` .

Contrairement à l'algorithme d'inférence de type pour les appels de méthode générique, dans ce cas, il n'y a que des *types* d'arguments, aucune *expression* d'argument. En particulier, il n'y a pas de fonctions anonymes et, par conséquent, n'a pas besoin de plusieurs phases d'inférence.

Au lieu de cela, `Xi` ils sont tous considérés comme non *fixes* et une *inférence de liaison inférieure* est effectuée *à partir* de chaque type `Uj` d'argument de `D` *vers* le type de paramètre correspondant `Tj` de `M` . Si aucune des limites n'a `Xi` été trouvée, l'inférence de type échoue. Dans le cas contraire, tous `Xi` sont *fixes* à `Si` l'équivalent, qui sont le résultat de l'inférence de type.

Recherche du meilleur type commun d'un ensemble d'expressions

Dans certains cas, un type commun doit être déduit pour un ensemble d'expressions. En particulier, les types d'éléments des tableaux implicitement typés et les types de retour des fonctions anonymes avec des corps de *bloc* sont détectés de cette façon.

Intuitivement, étant donné un ensemble d'expressions, `E1...Em` cette inférence doit être équivalente à l'appel d'une méthode

```
C#

Tr M<X>(X x1 ... X xm)
```

avec les `Ei` arguments As.

Plus précisément, l'inférence commence par une variable de type non fixe `X` . Les *inférences de type de sortie* sont ensuite effectuées *de* chaque `Ei` *à* `X` . Enfin, `X` est *fixe* et, en cas de réussite, le type résultant `S` est le meilleur type commun pour les expressions. Si tel n' `S` est pas le cas, les expressions n'ont pas le meilleur type commun.

Résolution de surcharge

La résolution de surcharge est un mécanisme de liaison-temps permettant de sélectionner le meilleur membre de fonction à appeler à partir d'une liste d'arguments

et d'un jeu de membres de fonction candidats. La résolution de surcharge sélectionne la fonction membre à appeler dans les contextes suivants dans C# :

- Appel d'une méthode nommée dans un *invocation_expression* (appels de méthode).
- Appel d'un constructeur d'instance nommé dans une *object_creation_expression* (expressions de création d'objet).
- Appel d'un accesseur d'indexeur via un *element_access* (accès à l'élément).
- Appel d'un opérateur prédéfini ou défini par l'utilisateur référencé dans une expression (résolution desurcharge d'opérateur unaire et résolution de surcharge d'opérateur binaire).

Chacun de ces contextes définit l'ensemble des membres de fonction candidats et la liste d'arguments de manière unique, comme décrit en détail dans les sections répertoriées ci-dessus. Par exemple, l'ensemble de candidats pour un appel de méthode n'inclut pas les méthodes marquées `override` (recherche de membre) et les méthodes d'une classe de base ne sont pas candidates si une méthode dans une classe dérivée est applicable (appels de méthode).

Une fois que les membres de la fonction candidate et la liste d'arguments ont été identifiés, la sélection du meilleur membre de la fonction est la même dans tous les cas :

- Compte tenu de l'ensemble des fonctions membres candidates applicables, le meilleur membre de fonction de cet ensemble est localisé. Si le jeu contient un seul membre de fonction, ce membre de fonction est le meilleur membre de fonction. Dans le cas contraire, le meilleur membre de fonction est le membre de fonction qui est mieux que tous les autres membres de fonction par rapport à la liste d'arguments donnée, à condition que chaque membre de fonction soit comparé à tous les autres membres de fonction en utilisant les règles dans un meilleur membrede fonction. S'il n'y a pas exactement un membre de fonction qui est mieux que tous les autres membres de fonction, l'appel de membre de fonction est ambigu et une erreur de liaison s'est produite.

Les sections suivantes définissent les significations exactes des termes *__membre de fonction applicable__ _ et _ *membre de fonction amélioré* *.

Membre de fonction applicable

Un membre de fonction est considéré comme un ***membre de fonction applicable*** en ce qui concerne une liste d'arguments `A` lorsque toutes les conditions suivantes sont vraies :

- Chaque argument dans `A` correspond à un paramètre dans la déclaration de membre de fonction comme décrit dans les paramètres correspondants, et tout paramètre auquel aucun argument correspond est un paramètre facultatif.
- Pour chaque argument dans `A` , le mode de passage de paramètre de l'argument (c'est-à-dire, valeur, `ref` ou `out`) est identique au mode de passage de paramètre du paramètre correspondant, et
  - pour un paramètre de valeur ou un tableau de paramètres, une conversion implicite (conversions implicites) existe à partir de l'argument vers le type du paramètre correspondant, ou
  - pour un `ref` `out` paramètre ou, le type de l'argument est identique au type du paramètre correspondant. Après tout, un `ref` `out` paramètre ou est un alias pour l'argument passé.

Pour un membre de fonction qui comprend un tableau de paramètres, si le membre de fonction est applicable par les règles ci-dessus, il est dit applicable dans sa **forme * normale** _. Si un membre de fonction qui comprend un tableau de paramètres n'est pas applicable dans sa forme normale, la fonction membre peut être applicable à la place dans sa *forme développée* _ * * * :

- La forme développée est construite en remplaçant le tableau de paramètres dans la déclaration de membre de fonction par zéro ou plusieurs paramètres de valeur du type d'élément du tableau de paramètres, de telle sorte que le nombre d'arguments dans la liste d'arguments `A` corresponde au nombre total de paramètres. Si `A` a moins d'arguments que le nombre de paramètres fixes dans la déclaration de membre de fonction, la forme développée de la fonction membre ne peut pas être construite et n'est donc pas applicable.
- Dans le cas contraire, la forme développée est applicable si, pour chaque argument dans `A` le mode de passage de paramètre de l'argument est identique au mode de passage de paramètre du paramètre correspondant, et
  - pour un paramètre de valeur fixe ou un paramètre de valeur créé par l'expansion, une conversion implicite (conversions implicites) existe à partir du type de l'argument vers le type du paramètre correspondant, ou
  - pour un `ref` `out` paramètre ou, le type de l'argument est identique au type du paramètre correspondant.

Meilleure fonction membre

Pour déterminer le meilleur membre de la fonction, une liste d'arguments tronquée a est construite, contenant uniquement les expressions d'arguments, dans l'ordre dans lequel elles apparaissent dans la liste d'arguments d'origine.

Les listes de paramètres pour chacun des fonctions candidates sont construites de la façon suivante :

- Le formulaire développé est utilisé si le membre de fonction ne s'applique qu'au formulaire développé.
- Les paramètres facultatifs sans arguments correspondants sont supprimés de la liste de paramètres
- Les paramètres sont réorganisés afin qu'ils se produisent à la même position que l'argument correspondant dans la liste d'arguments.

À partir d'une liste `A` d'arguments avec un ensemble d'expressions d'argument `{E1, E2, ..., En}` et deux membres de fonction applicables `Mp` et `Mq` avec des types de paramètres `{P1, P2, ..., Pn}` et `{Q1, Q2, ..., Qn}` , `Mp` est défini comme étant un ***meilleur membre de fonction*** que `Mq` si

- pour chaque argument, la conversion implicite de `Ex` en `Qx` n'est pas meilleure que la conversion implicite de `Ex` en `Px` , et
- pour au moins un argument, la conversion de `Ex` en `Px` est meilleure que la conversion de `Ex` en `Qx` .

Lors de l'exécution de cette évaluation, si `Mp` ou `Mq` est applicable dans sa forme développée, `Px` ou `Qx` fait référence à un paramètre dans la forme développée de la liste de paramètres.

Si les séquences de type de paramètre `{P1, P2, ..., Pn}` et `{Q1, Q2, ..., Qn}` sont équivalentes (c'est-à-dire `Pi` qu'elles ont une conversion d'identité vers la correspondante `Qi`), les règles de rupture de liaison suivantes sont appliquées, dans l'ordre, pour déterminer le meilleur membre de fonction.

- Si `Mp` est une méthode non générique et `Mq` est une méthode générique, `Mp` est mieux que `Mq` .
- Sinon, si `Mp` est applicable dans sa forme normale et `Mq` possède un `params` tableau et s'applique uniquement dans sa forme développée, `Mp` est mieux que `Mq` .
- Sinon, si `Mp` a plus de paramètres déclarés que `Mq` , `Mp` est mieux que `Mq` . Cela peut se produire si les deux méthodes ont des `params` tableaux et s'appliquent uniquement à leurs formes développées.
- Sinon, si tous les paramètres de `Mp` ont un argument correspondant, alors que les arguments par défaut doivent être remplacés par au moins un paramètre facultatif dans `Mq` , `Mp` est mieux que `Mq` .

- Sinon, si `Mp` a des types de paramètres plus spécifiques que `Mq` , `Mp` est mieux `Mq` que. Laissez `{R1, R2, ..., Rn}` et `{S1, S2, ..., Sn}` représentent les types de paramètres non instanciés et non développés de `Mp` et `Mq` . `Mp`les types de paramètres de sont plus spécifiques que `Mq` si, pour chaque paramètre, `Rx` n'est pas moins spécifique que `Sx` , et, pour au moins un paramètre, `Rx` est plus spécifique que `Sx` :
  - Un paramètre de type est moins spécifique qu'un paramètre de non-type.
  - De manière récursive, un type construit est plus spécifique qu'un autre type construit (avec le même nombre d'arguments de type) si au moins un argument de type est plus spécifique et qu'aucun argument de type n'est moins spécifique que l'argument de type correspondant dans l'autre.
  - Un type tableau est plus spécifique qu'un autre type tableau (avec le même nombre de dimensions) si le type d'élément du premier est plus spécifique que le type d'élément du second.
- Dans le cas contraire, si un membre est un opérateur non levé et que l'autre est un opérateur levé, la meilleure est la meilleure.
- Dans le cas contraire, aucune des fonctions membres n'est meilleure.

Meilleure conversion à partir de l'expression

Dans le cas d'une conversion implicite `C1` qui convertit une expression `E` en type `T1` , et d'une conversion implicite `C2` qui convertit une expression `E` en type `T2` , `C1` est une ***meilleure conversion*** que `C2` si `E` ne correspond pas exactement à `T2` et au moins l'un des éléments suivants :

- `E` correspond exactement à `T1` (expression correspondant exactement)
- `T1` est une meilleure cible de conversion que `T2` (meilleure cible de conversion)

Expression correspondant exactement

Pour une expression `E` et un type donnés `T` , `E` correspond exactement à `T` l'un des éléments suivants :

- `E` a un type `S` et une conversion d'identité existe de `S` en `T`
- `E` est une fonction anonyme, `T` est un type délégué `D` ou un type d'arborescence d'expression `Expression<D>` et l'un des éléments suivants :
  - Un type de retour déduit `X` existe pour `E` dans le contexte de la liste de paramètres de `D` (type de retour déduit), et une conversion d'identité existe dans `X` le type de retour de `D`

- `E` Est non asynchrone et `D` a un type de retour `Y` ou `E` est Async et `D` a un type de retour `Task<Y>` , et l'une des valeurs suivantes est retournée :
  - Le corps de `E` est une expression qui correspond exactement à `Y`
  - Le corps de `E` est un bloc d'instruction où chaque instruction return retourne une expression qui correspond exactement à `Y`

Meilleure cible de conversion

À partir de deux types différents `T1` et `T2` , `T1` est une meilleure cible de conversion que `T2` si aucune conversion implicite de `T2` en `T1` existe, et au moins l'un des éléments suivants :

- Conversion implicite de `T1` en `T2` Exists
- `T1` est un type délégué `D1` ou un type d'arborescence d'expression `Expression<D1>` , `T2` est un type délégué `D2` ou un type d'arborescence d'expression `Expression<D2>` , `D1` possède un type `S1` de retour et l'un des éléments suivants :
  - `D2` est un retour void
  - `D2` a un type de retour `S2` et `S1` est une meilleure cible de conversion que `S2`
- `T1` est `Task<S1>` , `T2` est `Task<S2>` , et `S1` est une meilleure cible de conversion que `S2`
- `T1` est `S1` ou `S1?` où `S1` est un type intégral signé, et `T2` est `S2` ou `S2?` où `S2` est un type intégral non signé. Plus précisément :
  - `S1` est `sbyte` et `S2` est `byte` , `ushort` , `uint` ou `ulong`
  - `S1` est `short` et `S2` est `ushort` , `uint` ou `ulong`
  - `S1` est `int` et `S2` est `uint` , ou `ulong`
  - `S1` est `long` et `S2` est `ulong`

Surcharge dans les classes génériques

Tandis que les signatures telles qu'elles sont déclarées doivent être uniques, il est possible que la substitution des arguments de type génère des signatures identiques. Dans ce cas, les règles de résolution de surcharge ci-dessus sélectionnent le membre le plus spécifique.

Les exemples suivants montrent des surcharges qui sont valides et non valides conformément à cette règle :

```
C#
```

```
interface I1<T> {...}

interface I2<T> {...}

class G1<U>
{
    int F1(U u);                  // Overload resolution for G<int>.F1
    int F1(int i);                // will pick non-generic

    void F2(I1<U> a);             // Valid overload
    void F2(I2<U> a);
}

class G2<U,V>
{
    void F3(U u, V v);            // Valid, but overload resolution for
    void F3(V v, U u);            // G2<int,int>.F3 will fail

    void F4(U u, I1<V> v);        // Valid, but overload resolution for
    void F4(I1<V> v, U u);        // G2<I1<int>,int>.F4 will fail

    void F5(U u1, I1<V> v2);      // Valid overload
    void F5(V v1, U u2);

    void F6(ref U u);             // valid overload
    void F6(out V v);
}
```

Vérification de la résolution de surcharge dynamique au moment de la compilation

Pour la plupart des opérations liées de manière dynamique, l'ensemble des candidats possibles pour la résolution est inconnu au moment de la compilation. Dans certains cas, toutefois, l'ensemble de candidats est connu au moment de la compilation :

- Appels de méthode statiques avec des arguments dynamiques
- Appels de méthode d'instance où le récepteur n'est pas une expression dynamique
- Appels de l'indexeur où le récepteur n'est pas une expression dynamique
- Appels de constructeur avec des arguments dynamiques

Dans ces cas, un contrôle limité au moment de la compilation est effectué pour chaque candidat afin de déterminer si l'un d'eux peut éventuellement s'appliquer au moment de l'exécution. Cette vérification se compose des étapes suivantes :

- Inférence de type partiel : tout argument de type qui ne dépend pas directement ou indirectement sur un argument de type `dynamic` est déduit à l'aide des règles d' inférence de type. Les arguments de type restants sont inconnus.

- Vérification de l'applicabilité partielle : l'applicabilité est vérifiée en fonction du membre de fonction applicable, mais les paramètres dont les types sont inconnus sont ignorés.
- Si aucun candidat ne réussit ce test, une erreur de compilation se produit.

Appel de membre de fonction

Cette section décrit le processus qui a lieu au moment de l'exécution pour appeler un membre de fonction particulier. Il est supposé qu'un processus de liaison a déjà déterminé le membre particulier à appeler, éventuellement en appliquant la résolution de surcharge à un ensemble de membres de fonction candidats.

Dans le cadre de la description du processus d'appel, les membres de fonction sont divisés en deux catégories :

- Fonctions membres statiques. Il s'agit des constructeurs d'instance, des méthodes statiques, des accesseurs de propriété statique et des opérateurs définis par l'utilisateur. Les fonctions membres statiques sont toujours non virtuelles.
- Membres de fonction d'instance. Il s'agit des méthodes d'instance, des accesseurs de propriété d'instance et des accesseurs d'indexeurs. Les fonctions membres de fonction d'instance sont non virtuelles ou virtuelles et sont toujours appelées sur une instance particulière. L'instance est calculée par une expression d'instance et devient accessible dans le membre de fonction en tant que `this` (cet accès).

Le traitement au moment de l'exécution d'un appel de membre de fonction se compose des étapes suivantes, où `M` est la fonction membre et, si `M` est un membre d'instance, `E` est l'expression d'instance :

- Si `M` est un membre de fonction statique :
  - La liste d'arguments est évaluée comme décrit dans listes d'arguments.
  - `M` est appelé.

- Si `M` est un membre de fonction d'instance déclaré dans un *Value_type*:
  - `E` est évalué. Si cette évaluation provoque une exception, aucune autre étape n'est exécutée.
  - Si `E` n'est pas classé en tant que variable, une variable locale temporaire du `E` type de est créée et la valeur de `E` est assignée à cette variable. `E` est ensuite reclassifiée comme une référence à cette variable locale temporaire. La variable temporaire est accessible comme `this` dans `M` , mais pas d'une autre façon. Ainsi, uniquement lorsque `E` est une véritable variable, l'appelant peut observer les modifications apportées `M` à `this` .

    - La liste d'arguments est évaluée comme décrit dans listes d'arguments.
    - `M` est appelé. La variable référencée par `E` devient la variable référencée par `this` .

- Si `M` est un membre de fonction d'instance déclaré dans un *reference_type*:
    - `E` est évalué. Si cette évaluation provoque une exception, aucune autre étape n'est exécutée.
    - La liste d'arguments est évaluée comme décrit dans listes d'arguments.
    - Si le type de `E` est un *Value_type*, une conversion boxing (conversions boxing) est effectuée pour convertir `E` en type `object` , et `E` est considéré comme étant de type `object` dans les étapes suivantes. Dans ce cas, `M` ne peut être qu'un membre de `System.Object` .
    - La valeur de `E` est vérifiée comme valide. Si la valeur de `E` est `null` , une `System.NullReferenceException` est levée et aucune autre étape n'est exécutée.
    - L'implémentation de la fonction membre à appeler est déterminée :
        - Si le type au moment de la liaison `E` est une interface, la fonction membre à appeler est l'implémentation de `M` fournie par le type au moment de l'exécution de l'instance référencée par `E` . Cette fonction membre est déterminée en appliquant les règles de mappage d'interface (mappage d'interface) pour déterminer l'implémentation de `M` fournie par le type d'exécution de l'instance référencée par `E` .
        - Sinon, si `M` est un membre de fonction virtuelle, la fonction membre à appeler est l'implémentation de `M` fournie par le type au moment de l'exécution de l'instance référencée par `E` . Cette fonction membre est déterminée en appliquant les règles de détermination de l'implémentation la plus dérivée (méthodes virtuelles) de `M` en ce qui concerne le type d'exécution de l'instance référencée par `E` .
        - Sinon, `M` est un membre de fonction non virtuel, et la fonction membre à appeler est `M` elle-même.
    - L'implémentation de la fonction membre déterminée à l'étape ci-dessus est appelée. L'objet référencé par `E` devient l'objet référencé par `this` .

Appels sur les instances boxed

Un membre de fonction implémenté dans un *Value_type* peut être appelé via une instance boxed de qui *Value_type* dans les situations suivantes :

- Quand la fonction membre est un `override` d'une méthode héritée du type `object` et est appelée par le biais d'une expression d'instance de type `object` .

- Quand la fonction membre est une implémentation d'un membre de fonction d'interface et qu'elle est appelée par le biais d'une expression d'instance d'un *INTERFACE_TYPE*.
- Quand la fonction membre est appelée par l'intermédiaire d'un délégué.

Dans ces situations, l'instance boxed est considérée comme contenant une variable de la *Value_type*, et cette variable devient la variable référencée par `this` dans l'appel de membre de fonction. En particulier, cela signifie que lorsqu'un membre de fonction est appelé sur une instance boxed, la fonction membre peut modifier la valeur contenue dans l'instance boxed.

Expressions primaires

Les expressions primaires incluent les formes les plus simples des expressions.

antlr

```
primary_expression
    : primary_no_array_creation_expression
    | array_creation_expression
    ;

primary_no_array_creation_expression
    : literal
    | interpolated_string_expression
    | simple_name
    | parenthesized_expression
    | member_access
    | invocation_expression
    | element_access
    | this_access
    | base_access
    | post_increment_expression
    | post_decrement_expression
    | object_creation_expression
    | delegate_creation_expression
    | anonymous_object_creation_expression
    | typeof_expression
    | checked_expression
    | unchecked_expression
    | default_value_expression
    | nameof_expression
    | anonymous_method_expression
    | primary_no_array_creation_expression_unsafe
    ;
```

Les expressions primaires sont réparties entre *array_creation_expression* s et *primary_no_array_creation_expression* s. Le traitement de l'expression de création de

tableau de cette manière, plutôt que son affichage avec les autres formulaires d'expression simples, permet à la grammaire d'interdire le code potentiellement confus, tel que

```C#
object o = new int[3][1];
```

qui, autrement, serait interprété comme

```C#
object o = (new int[3])[1];
```

Littéraux

Une *primary_expression* qui se compose d'un *littéral* (littéraux) est classée en tant que valeur.

Chaînes interpolées

Un *interpolated_string_expression* se compose d'un `$` signe suivi d'un littéral de chaîne normal ou textuel, où les trous, délimités par et, délimitent les `{` `}` expressions et les spécifications de mise en forme. Une expression de chaîne interpolée est le résultat d'une *interpolated_string_literal* qui a été divisée en jetons individuels, comme décrit dans littéraux de chaîne interpolés.

```antlr
interpolated_string_expression
    : '$' interpolated_regular_string
    | '$' interpolated_verbatim_string
    ;

interpolated_regular_string
    : interpolated_regular_string_whole
    | interpolated_regular_string_start interpolated_regular_string_body
interpolated_regular_string_end
    ;

interpolated_regular_string_body
    : interpolation (interpolated_regular_string_mid interpolation)*
    ;

interpolation
```

```
    : expression
    | expression ',' constant_expression
    ;

interpolated_verbatim_string
    : interpolated_verbatim_string_whole
    | interpolated_verbatim_string_start interpolated_verbatim_string_body
interpolated_verbatim_string_end
    ;

interpolated_verbatim_string_body
    : interpolation (interpolated_verbatim_string_mid interpolation)+
    ;
```

La *constant_expression* dans une interpolation doit avoir une conversion implicite en `int` .

Une *interpolated_string_expression* est classée en tant que valeur. Si elle est immédiatement convertie en `System.IFormattable` ou `System.FormattableString` avec une conversion de chaîne interpolée implicite (conversions de chaînes interpolées implicites), l'expression de chaîne interpolée a ce type. Dans le cas contraire, il a le type `string` .

Si le type d'une chaîne interpolée est `System.IFormattable` ou `System.FormattableString` , la signification est un appel à `System.Runtime.CompilerServices.FormattableStringFactory.Create` . Si le type est `string` , la signification de l'expression est un appel à `string.Format` . Dans les deux cas, la liste d'arguments de l'appel se compose d'un littéral de chaîne de format avec des espaces réservés pour chaque interpolation, et d'un argument pour chaque expression qui correspond aux espaces réservés.

Le littéral de chaîne de format est construit comme suit, où `N` est le nombre d'interpolations dans le *interpolated_string_expression*:

- Si un *interpolated_regular_string_whole* ou un *interpolated_verbatim_string_whole* suit le `$` signe, le littéral de chaîne de format est ce jeton.
- Dans le cas contraire, le littéral de chaîne de format se compose des éléments suivants :
  - Premier *interpolated_regular_string_start* ou *interpolated_verbatim_string_start*
  - Puis, pour chaque numéro `I` de `0` à `N-1` :
    - Représentation décimale de `I`
    - Ensuite, si l' *interpolation* correspondante a un *constant_expression*, un `,` (virgule) suivi de la représentation décimale de la valeur de l' *constant_expression*

    - Ensuite, le *interpolated_regular_string_mid*, *interpolated_regular_string_end*, *interpolated_verbatim_string_mid* ou *interpolated_verbatim_string_end* qui suit immédiatement l'interpolation correspondante.

Les arguments suivants sont simplement les *expressions* des *interpolations* (le cas échéant), dans l'ordre.

TODO : exemples.

Noms simples

Un *simple_name* se compose d'un identificateur, éventuellement suivi d'une liste d'arguments de type :

antlr

```
simple_name
    : identifier type_argument_list?
    ;
```

Une *simple_name* se présente sous la forme `I` ou sous la forme `I<A1,...,Ak>` , où `I` est un identificateur unique et `<A1,...,Ak>` est un *type_argument_list* facultatif. Si aucun *type_argument_list* n'est spécifié, pensez `K` à être égal à zéro. La *simple_name* est évaluée et classée comme suit :

- Si `K` est égal à zéro et que le *simple_name* apparaît dans un *bloc* et que l'espace de déclaration de variable locale (déclarations) du *bloc*(ou du *bloc* englobant) contient une variable locale, un paramètre ou une constante portant `I` le nom, le *simple_name* fait référence à cette variable locale, ce paramètre ou cette constante et est classifié comme une variable ou une valeur.

- Si `K` est égal à zéro et que le *simple_name* apparaît dans le corps d'une déclaration de méthode générique et si cette déclaration comprend un paramètre de type portant `I` le nom, le *simple_name* fait référence à ce paramètre de type.

- Dans le cas contraire, pour chaque type `T` d'instance (type d'instance), en commençant par le type d'instance de la déclaration de type englobante immédiate et en continuant avec le type d'instance de chaque déclaration de classe ou de struct (le cas échéant) :
    - Si `K` est égal à zéro et que la déclaration de `T` comprend un paramètre de type portant `I` le nom, le *simple_name* fait référence à ce paramètre de type.

    - Sinon, si une recherche de membre (recherche de membre) de `I` dans `T` avec les `K` arguments de type produit une correspondance :
        - Si `T` est le type d'instance du type de classe ou de struct immédiatement englobant et que la recherche identifie une ou plusieurs méthodes, le résultat est un groupe de méthodes avec une expression d'instance associée de `this` . Si une liste d'arguments de type a été spécifiée, elle est utilisée pour appeler une méthode générique (appels de méthode).
        - Sinon, si `T` est le type d'instance du type de classe ou de struct immédiatement englobant, si la recherche identifie un membre d'instance et si la référence se produit dans le corps d'un constructeur d'instance, une méthode d'instance ou un accesseur d'instance, le résultat est le même qu'un accès de membre (accès aux membres) du formulaire `this.I` . Cela peut se produire uniquement lorsque `K` est égal à zéro.
        - Dans le cas contraire, le résultat est le même qu'un accès de membre (accès aux membres) du formulaire `T.I` ou `T.I<A1,...,Ak>` . Dans ce cas, il s'agit d'une erreur de liaison au moment de la liaison pour que le *simple_name* fait référence à un membre d'instance.

- Dans le cas contraire, pour chaque espace de noms `N` , en commençant par l'espace de noms dans lequel le *simple_name* se produit, en continuant avec chaque espace de noms englobant (le cas échéant) et se terminant par l'espace de noms global, les étapes suivantes sont évaluées jusqu'à ce qu'une entité soit localisée :
    - Si `K` est égal à zéro et `I` est le nom d'un espace de noms dans `N` , alors :
        - Si l'emplacement où le *simple_name* se produit est entouré d'une déclaration d'espace de noms pour `N` et que la déclaration d'espace de noms contient une *extern_alias_directive* ou *using_alias_directive* qui associe le nom `I` à un espace de noms ou à un type, la *simple_name* est ambiguë et une erreur de compilation se produit.
        - Dans le cas contraire, le *simple_name* fait référence à l'espace de noms nommé `I` dans `N` .
    - Sinon, si `N` contient un type accessible avec `I` des paramètres Name et `K` type, alors :
        - Si `K` est égal à zéro et que l'emplacement où le *simple_name* se produit est entouré d'une déclaration d'espace de noms pour `N` et que la déclaration d'espace de noms contient une *extern_alias_directive* ou *using_alias_directive* qui associe le nom `I` à un espace de noms ou à un type, le *simple_name* est ambigu et une erreur de compilation se produit.
        - Sinon, le *namespace_or_type_name* fait référence au type construit avec les arguments de type donnés.

- Sinon, si l'emplacement où le *simple_name* se produit est entouré d'une déclaration d'espace de noms pour `N` :
  - Si `K` est égal à zéro et que la déclaration d'espace de noms contient une *extern_alias_directive* ou *using_alias_directive* qui associe le nom `I` à un espace de noms ou un type importé, le *simple_name* fait référence à cet espace de noms ou type.
  - Sinon, si les espaces de noms et les déclarations de type importés par le *using_namespace_directive* s et *using_static_directive* s de la déclaration d'espace de noms contiennent exactement un type accessible ou un membre statique non-extension ayant des `I` paramètres de type et de nom `K` , le *simple_name* fait référence à ce type ou membre construit avec les arguments de type donnés.
  - Dans le cas contraire, si les espaces de noms et les types importés par le *using_namespace_directive* s de la déclaration d'espace de noms contiennent plusieurs membres statiques de type accessible ou non-méthode d'extension qui ont des `I` paramètres de type et de nom `K` , le *simple_name* est ambigu et une erreur se produit.

  Notez que cette étape entière est exactement parallèle à l'étape correspondante dans le traitement d'un *namespace_or_type_name* (espace de noms et noms de type).

- Dans le cas contraire, la *simple_name* n'est pas définie et une erreur de compilation se produit.

Expressions entre parenthèses

Une *parenthesized_expression* se compose d'une *expression* placée entre parenthèses.

antlr

```
parenthesized_expression
    : '(' expression ')'
    ;
```

Une *parenthesized_expression* est évaluée en évaluant l' *expression* entre parenthèses. Si l' *expression* entre parenthèses désigne un espace de noms ou un type, une erreur de compilation se produit. Dans le cas contraire, le résultat de l' *parenthesized_expression* est le résultat de l'évaluation de l' *expression* contenue.

Accès au membre

Un *member_access* se compose d'un *primary_expression*, d'un *predefined_type* ou d'un *qualified_alias_member*, suivi d'un `.` jeton «», suivi d'un *identificateur*, éventuellement suivi d'un *type_argument_list*.

antlr

```
member_access
    : primary_expression '.' identifier type_argument_list?
    | predefined_type '.' identifier type_argument_list?
    | qualified_alias_member '.' identifier
    ;

predefined_type
    : 'bool'   | 'byte'  | 'char'  | 'decimal' | 'double' | 'float' | 'int'
| 'long'
    | 'object' | 'sbyte' | 'short' | 'string'  | 'uint'   | 'ulong' |
'ushort'
    ;
```

La *qualified_alias_member* production est définie dans les qualificateurs d'alias d'espace de noms.

Une *member_access* se présente sous la forme `E.I` ou sous la forme `E.I<A1, ..., Ak>` , où `E` est une expression primaire, `I` est un identificateur unique et `<A1, ..., Ak>` est un *type_argument_list* facultatif. Si aucun *type_argument_list* n'est spécifié, pensez `K` à être égal à zéro.

Un *member_access* avec un *primary_expression* de type `dynamic` est lié dynamiquement (liaison dynamique). Dans ce cas, le compilateur classe l'accès au membre comme un accès à la propriété de type `dynamic` . Les règles ci-dessous pour déterminer la signification de l' *member_access* sont ensuite appliquées au moment de l'exécution, à l'aide du type au moment de l'exécution et non du type de compilation du *primary_expression*. Si la classification au moment de l'exécution provoque un groupe de méthodes, l'accès au membre doit être le *primary_expression* d'un *invocation_expression*.

La *member_access* est évaluée et classée comme suit :

- Si `K` est égal à zéro et `E` est un espace de noms et `E` contient un espace de noms imbriqué portant `I` le nom, le résultat est cet espace de noms.
- Sinon, si `E` est un espace de noms et `E` contient un type accessible avec des `I` paramètres Name et `K` type, le résultat est ce type construit avec les arguments de type donnés.
- Si `E` est un *predefined_type* ou un *primary_expression* classifié comme un type, si `E` n'est pas un paramètre de type et si une recherche de membre (recherche de

membre) de `I` dans avec des `E` paramètres de `K` type produit une correspondance, `E.I` est évaluée et classée comme suit :

- Si `I` identifie un type, le résultat est ce type construit avec les arguments de type donnés.
- Si `I` identifie une ou plusieurs méthodes, le résultat est un groupe de méthodes sans expression d'instance associée. Si une liste d'arguments de type a été spécifiée, elle est utilisée pour appeler une méthode générique (appels de méthode).
- Si `I` identifie une `static` propriété, le résultat est un accès à une propriété sans expression d'instance associée.
- Si `I` identifie un `static` champ :
  - Si le champ est `readonly` et que la référence se produit à l'extérieur du constructeur statique de la classe ou du struct dans lequel le champ est déclaré, le résultat est une valeur, à savoir la valeur du champ statique `I` dans `E` .
  - Dans le cas contraire, le résultat est une variable, à savoir le champ static `I` dans `E` .
- Si `I` identifie un `static` événement :
  - Si la référence se produit dans la classe ou le struct dans lequel l'événement est déclaré, et si l'événement a été déclaré sans *event_accessor_declarations* (événements), `E.I` est traité exactement comme s'il s' `I` agissait d'un champ statique.
  - Dans le cas contraire, le résultat est un accès à un événement sans expression d'instance associée.
- Si `I` identifie une constante, le résultat est une valeur, à savoir la valeur de cette constante.
- Si `I` identifie un membre de l'énumération, le résultat est une valeur, à savoir la valeur de ce membre de l'énumération.
- Sinon, `E.I` est une référence de membre non valide et une erreur de compilation se produit.

- Si `E` est un accès à une propriété, un indexeur, une variable ou une valeur, le type de qui est `T` , et une recherche de membre (recherche de membre) de `I` dans avec les arguments de `T` `K` type produit une correspondance, `E.I` est évaluée et classée comme suit :
  - Premièrement, si `E` est un accès à une propriété ou à un indexeur, la valeur de la propriété ou de l'indexeur est obtenue (valeurs des expressions) et `E` est reclassifiée comme une valeur.
  - Si `I` identifie une ou plusieurs méthodes, le résultat est un groupe de méthodes avec une expression d'instance associée de `E` . Si une liste d'arguments de type

a été spécifiée, elle est utilisée pour appeler une méthode générique (appels de méthode).
    - Si `I` identifie une propriété d'instance,
        - Si `E` est `this` , `I` identifie une propriété implémentée automatiquement (Propriétés implémentées automatiquement) sans accesseur Set, et la référence se produit dans un constructeur d'instance pour un type de classe ou de struct `T` , le résultat est une variable, à savoir le champ de stockage masqué pour la propriété automatique fournie par `I` dans l'instance de `T` fournie par `this` .
        - Dans le cas contraire, le résultat est un accès à une propriété avec une expression d'instance associée de `E` .
    - Si `T` est un *class_type* et `I` identifie un champ d'instance de ce *class_type*:
        - Si la valeur de `E` est `null` , une `System.NullReferenceException` est levée.
        - Sinon, si le champ est `readonly` et que la référence se produit en dehors d'un constructeur d'instance de la classe dans laquelle le champ est déclaré, le résultat est une valeur, à savoir la valeur du champ `I` dans l'objet référencé par `E` .
        - Dans le cas contraire, le résultat est une variable, à savoir le champ `I` de l'objet référencé par `E` .
    - Si `T` est un *struct_type* et `I` identifie un champ d'instance de ce *struct_type*:
        - Si `E` est une valeur, ou si le champ est `readonly` et que la référence se produit en dehors d'un constructeur d'instance du struct dans lequel le champ est déclaré, le résultat est une valeur, à savoir la valeur du champ `I` dans l'instance de struct donnée par `E` .
        - Dans le cas contraire, le résultat est une variable, à savoir le champ `I` dans l'instance de struct donnée par `E` .
    - Si `I` identifie un événement d'instance :
        - Si la référence se produit dans la classe ou le struct dans lequel l'événement est déclaré, et si l'événement a été déclaré sans *event_accessor_declarations* (événements) et si la référence ne se produit pas comme la partie gauche d'un `+=` `-=` opérateur or, `E.I` est traitée exactement comme s'il s' `I` agissait d'un champ d'instance.
        - Dans le cas contraire, le résultat est un accès aux événements avec une expression d'instance associée de `E` .
- Sinon, une tentative est faite pour traiter `E.I` en tant qu'appel de méthode d'extension (appels de méthode d'extension). En cas d'échec, `E.I` est une référence de membre non valide et une erreur de liaison s'est produite.

Noms simples et noms de types identiques

Dans un accès aux membres du formulaire `E.I` , si `E` est un identificateur unique, et si la signification de `E` comme un *simple_name* (noms simples) est une constante, un champ, une propriété, une variable locale ou un paramètre du même type que la signification de `E` comme *type_name* (espace de noms et noms de type), les deux significations possibles de `E` sont autorisées. Les deux significations possibles de `E.I` ne sont jamais ambiguës, car `I` doivent nécessairement être membres du type `E` dans les deux cas. En d'autres termes, la règle autorise simplement l'accès aux membres statiques et aux types imbriqués `E` où une erreur de compilation se produirait autrement. Par exemple :

C#

```
struct Color
{
    public static readonly Color White = new Color(...);
    public static readonly Color Black = new Color(...);

    public Color Complement() {...}
}

class A
{
    public Color Color;                 // Field Color of type Color

    void F() {
        Color = Color.Black;            // References Color.Black static
member
        Color = Color.Complement();     // Invokes Complement() on Color
field
    }

    static void G() {
        Color c = Color.White;          // References Color.White static
member
    }
}
```

Ambiguïtés grammaticales

Les productions pour *simple_name* (noms simples) et *member_access* (accès aux membres) peuvent donner lieu à des ambiguïtés dans la grammaire des expressions. Par exemple, l'instruction suivante :

C#

```
F(G<A,B>(7));
```

peut être interprété comme un appel à `F` avec deux arguments, `G < A` et `B > (7)` . Elle peut également être interprétée comme un appel à `F` avec un argument, qui est un appel à une méthode générique `G` avec deux arguments de type et un argument normal.

Si une séquence de jetons peut être analysée (en contexte) comme un *simple_name* (noms simples), *member_access* (accès aux membres) ou *pointer_member_access* (accès au membre pointeur) se terminant par un *type_argument_list* (arguments de type), le jeton qui suit immédiatement le jeton de fermeture `>` est examiné. S'il s'agit de l'un des

```
C#

( ) ] } : ; , . ? == != | ^
```

le *type_argument_list* est ensuite conservé dans le cadre du *simple_name*, *member_access* ou *pointer_member_access* et toute autre analyse possible de la séquence de jetons est ignorée. Dans le cas contraire, le *type_argument_list* n'est pas considéré comme faisant partie du *simple_name*, *member_access* ou *pointer_member_access*, même s'il n'y a pas d'autre analyse possible de la séquence de jetons. Notez que ces règles ne sont pas appliquées lors de l'analyse d'un *type_argument_list* dans un *namespace_or_type_name* (noms d'espaces de noms et de types). L'instruction

```
C#

F(G<A,B>(7));
```

, conformément à cette règle, sera interprété comme un appel à `F` avec un argument, qui est un appel à une méthode générique `G` avec deux arguments de type et un argument normal. Les instructions

```
C#

F(G < A, B > 7);
F(G < A, B >> 7);
```

est interprété comme un appel à `F` avec deux arguments. L'instruction

```
C#

x = F < A > +y;
```

est interprété comme un opérateur inférieur à, supérieur à et opérateur unaire plus, comme si l'instruction avait été écrite `x = (F < A) > (+y)` , au lieu d'être *simple_name* avec un *type_argument_list* suivi d'un opérateur binaire plus. Dans l'instruction

```
C#

x = y is C<T> + z;
```

les jetons `C<T>` sont interprétés comme un *namespace_or_type_name* avec une *type_argument_list*.

Appel d'expressions

Un *invocation_expression* est utilisé pour appeler une méthode.

```
antlr

invocation_expression
    : primary_expression '(' argument_list? ')'
    ;
```

Un *invocation_expression* est lié dynamiquement (liaison dynamique) si au moins l'un des éléments suivants est présent :

- Le *primary_expression* a le type au moment de la compilation `dynamic` .
- Au moins un argument du *argument_list* facultatif a le type au moment de la compilation `dynamic` et le *primary_expression* n'a pas de type délégué.

Dans ce cas, le compilateur classe la *invocation_expression* comme une valeur de type `dynamic` . Les règles ci-dessous pour déterminer la signification du *invocation_expression* sont ensuite appliquées au moment de l'exécution, à l'aide du type au moment de l'exécution au lieu du type de compilation de celles du *primary_expression* et des arguments ayant le type au moment de la compilation `dynamic` . Si le *primary_expression* n'a pas de type au moment de la compilation `dynamic` , l'appel de la méthode subit une vérification limitée au moment de la compilation, comme décrit dans vérification de la résolution de surcharge dynamique au moment de la compilation.

La *primary_expression* d'un *invocation_expression* doit être un groupe de méthodes ou une valeur d'un *delegate_type*. Si le *primary_expression* est un groupe de méthodes, le *invocation_expression* est un appel de méthode (appel de méthode). Si le *primary_expression* est une valeur d'un *delegate_type*, le *invocation_expression* est un

appel de délégué (appel de délégué). Si la *primary_expression* n'est ni un groupe de méthodes, ni une valeur d'un *delegate_type*, une erreur de liaison s'est produite.

La *argument_list* facultative (listes d'arguments) fournit des valeurs ou des références de variables pour les paramètres de la méthode.

Le résultat de l'évaluation d'un *invocation_expression* est classé comme suit :

- Si le *invocation_expression* appelle une méthode ou un délégué qui retourne `void`, le résultat est Nothing. Une expression classifiée comme Nothing n'est autorisée que dans le contexte d'une *statement_expression* (instructions d'expression) ou en tant que corps d'une *lambda_expression* (expressions de fonction anonymes). Sinon, une erreur de temps de liaison se produit.
- Dans le cas contraire, le résultat est une valeur du type retourné par la méthode ou le délégué.

Appels de méthode

Pour un appel de méthode, le *primary_expression* du *invocation_expression* doit être un groupe de méthodes. Le groupe de méthodes identifie la méthode à appeler ou l'ensemble de méthodes surchargées à partir desquelles choisir une méthode spécifique à appeler. Dans ce dernier cas, la détermination de la méthode spécifique à appeler est basée sur le contexte fourni par les types des arguments de la *argument_list*.

Le traitement au moment de la liaison d'un appel de méthode de la forme `M(A)`, où `M` est un groupe de méthodes (éventuellement avec un *type_argument_list*) et `A` est un *argument_list* facultatif, se compose des étapes suivantes :

- L'ensemble des méthodes candidates pour l'appel de méthode est construit. Pour chaque méthode `F` associée au groupe de méthodes `M` :
  - Si `F` est non générique, `F` est candidat lorsque :
    - `M` n'a pas de liste d'arguments de type, et
    - `F` est applicable en ce qui concerne `A` (fonction membre applicable).
  - Si `F` est générique et n' `M` a pas de liste d'arguments de type, `F` est candidat lorsque :
    - L'inférence de type (inférence de type) est réussie, en déduisant une liste d'arguments de type pour l'appel, et
    - Une fois que les arguments de type inférés sont substitués aux paramètres de type de méthode correspondants, tous les types construits dans la liste de paramètres de F répondent à leurs contraintes (quisatisfont les contraintes) et la liste de paramètres de `F` est applicable à `A` (fonction membre applicable).

    - Si `F` est générique et qu'il `M` comprend une liste d'arguments de type, `F` est candidat lorsque :
        - `F` a le même nombre de paramètres de type de méthode que ceux fournis dans la liste d'arguments de type, et
        - Une fois que les arguments de type sont substitués aux paramètres de type de méthode correspondants, tous les types construits dans la liste de paramètres de F répondent à leurs contraintes (quisatisfont les contraintes) et la liste de paramètres de `F` est applicable à `A` (fonction membre applicable).
- L'ensemble des méthodes candidates est réduit pour contenir uniquement des méthodes des types les plus dérivés : pour chaque méthode du `C.F` jeu, où `C` est le type dans lequel la méthode `F` est déclarée, toutes les méthodes déclarées dans un type de base de `C` sont supprimées de l'ensemble. En outre, si `C` est un type de classe autre que `object` , toutes les méthodes déclarées dans un type interface sont supprimées de l'ensemble. (Cette dernière règle a un impact uniquement lorsque le groupe de méthodes était le résultat d'une recherche de membre sur un paramètre de type ayant une classe de base effective autre que Object et un ensemble d'interfaces effectif non vide.)
- Si le jeu résultant de méthodes candidates est vide, le traitement suivant des étapes suivantes est abandonné, et une tentative est effectuée pour traiter l'appel comme un appel de méthode d'extension (appels de méthode d'extension). En cas d'échec, il n'existe aucune méthode applicable et une erreur de liaison s'est produite.
- La meilleure méthode de l'ensemble de méthodes candidates est identifiée à l'aide des règles de résolution de surcharge de la résolution de surcharge. Si une seule meilleure méthode ne peut pas être identifiée, l'appel de la méthode est ambigu et une erreur de liaison s'est produite. Lors de l'exécution de la résolution de surcharge, les paramètres d'une méthode générique sont pris en compte après avoir remplacé les arguments de type (fournis ou déduits) pour les paramètres de type de méthode correspondants.
- La validation finale de la meilleure méthode choisie est effectuée :
    - La méthode est validée dans le contexte du groupe de méthodes : si la meilleure méthode est une méthode statique, le groupe de méthodes doit résulter d'un *simple_name* ou d'un *member_access* par le biais d'un type. Si la meilleure méthode est une méthode d'instance, le groupe de méthodes doit résulter d'une *simple_name*, d'une *member_access* par le biais d'une variable ou d'une valeur ou d'un *base_access*. Si aucune de ces conditions n'est remplie, une erreur de liaison au moment de la liaison se produit.
    - Si la meilleure méthode est une méthode générique, les arguments de type (fournis ou déduits) sont vérifiés par rapport aux contraintes (contraintes

satisfaisantes) déclarées dans la méthode générique. Si un argument de type ne satisfait pas la ou les contraintes correspondantes sur le paramètre de type, une erreur de liaison s'est produite.

Une fois qu'une méthode a été sélectionnée et validée au moment de la liaison par les étapes ci-dessus, l'appel réel au moment de l'exécution est traité en fonction des règles d'appel des membres de fonction décrites dans vérification de la résolution de surcharge dynamiquepar la compilation.

L'effet intuitif des règles de résolution décrites ci-dessus est le suivant : pour localiser la méthode particulière appelée par un appel de méthode, commencez par le type indiqué par l'appel de méthode et poursuivez la chaîne d'héritage jusqu'à ce qu'au moins une déclaration de méthode applicable, accessible et non substituée soit trouvée. Effectuez ensuite l'inférence de type et la résolution de surcharge sur l'ensemble des méthodes applicables, accessibles et non substituées déclarées dans ce type et appelez la méthode sélectionnée. Si aucune méthode n'a été trouvée, essayez plutôt de traiter l'appel comme un appel de méthode d'extension.

Appels de méthode d'extension

Dans un appel de méthode (appels sur des instances boxed) de l'un des formulaires

```
C#

expr . identifier ( )

expr . identifier ( args )

expr . identifier < typeargs > ( )

expr . identifier < typeargs > ( args )
```

Si le traitement normal de l'appel ne trouve aucune méthode applicable, une tentative est faite pour traiter la construction comme un appel de méthode d'extension. Si *expr* ou l'un des *arguments* a le type au moment de la compilation `dynamic` , les méthodes d'extension ne s'appliquent pas.

L'objectif est de trouver le meilleur *type_name* `C` , afin que l'appel de la méthode statique correspondante puisse avoir lieu :

```
C#

C . identifier ( expr )

C . identifier ( expr , args )
```

```
C . identifier < typeargs > ( expr )

C . identifier < typeargs > ( expr , args )
```

Une méthode d'extension `Ci.Mj` est ***éligible*** dans les cas suivants :

- `Ci` est une classe non générique et non imbriquée
- Le nom de `Mj` est *identifier*
- `Mj` est accessible et applicable lorsqu'il est appliqué aux arguments en tant que méthode statique, comme indiqué ci-dessus
- Une identité, une référence ou une conversion boxing implicite existe entre *expr* et le type du premier paramètre de `Mj` .

La recherche de `C` continue comme suit :

- À partir de la déclaration d'espace de noms englobante la plus proche, en continuant avec chaque déclaration d'espace de noms englobant, et se terminant par l'unité de compilation conteneur, des tentatives successives sont effectuées pour trouver un ensemble candidat de méthodes d'extension :
  - Si l'espace de noms ou l'unité de compilation donné contient directement des déclarations de type non génériques `Ci` avec les méthodes d'extension éligibles `Mj` , le jeu de ces méthodes d'extension est l'ensemble de candidats.
  - Si les types `Ci` importés par *using_static_declarations* et déclarés directement dans les espaces de noms importés par *using_namespace_directive* s dans l'espace de noms ou l'unité de compilation donné contiennent directement des méthodes d'extension éligibles `Mj` , le jeu de ces méthodes d'extension est l'ensemble candidat.
- Si aucun jeu de candidats n'est trouvé dans une déclaration d'espace de noms ou une unité de compilation englobante, une erreur de compilation se produit.
- Sinon, la résolution de surcharge est appliquée à l'ensemble de candidats comme décrit dans (résolution de surcharge). Si aucune meilleure méthode n'est trouvée, une erreur de compilation se produit.
- `C` est le type dans lequel la meilleure méthode est déclarée en tant que méthode d'extension.

À l'aide de `C` en tant que cible, l'appel de méthode est ensuite traité comme un appel de méthode statique (vérification au moment de la compilation de la résolution de surcharge dynamique).

Les règles précédentes signifient que les méthodes d'instance sont prioritaires sur les méthodes d'extension, que les méthodes d'extension disponibles dans les déclarations

d'espace de noms interne ont la priorité sur les méthodes d'extension disponibles dans les déclarations d'espaces de noms externes et que les méthodes d'extension déclarées directement dans un espace de noms ont priorité sur les méthodes d'extension importées dans ce même espace de noms Par exemple :

C#

```
public static class E
{
    public static void F(this object obj, int i) { }

    public static void F(this object obj, string s) { }
}

class A { }

class B
{
    public void F(int i) { }
}

class C
{
    public void F(object obj) { }
}

class X
{
    static void Test(A a, B b, C c) {
        a.F(1);             // E.F(object, int)
        a.F("hello");       // E.F(object, string)

        b.F(1);             // B.F(int)
        b.F("hello");       // E.F(object, string)

        c.F(1);             // C.F(object)
        c.F("hello");       // C.F(object)
    }
}
```

Dans l'exemple, la `B` méthode de est prioritaire sur la première méthode d'extension et la `C` méthode de est prioritaire sur les deux méthodes d'extension.

C#

```
public static class C
{
    public static void F(this int i) { Console.WriteLine("C.F({0})", i); }
    public static void G(this int i) { Console.WriteLine("C.G({0})", i); }
    public static void H(this int i) { Console.WriteLine("C.H({0})", i); }
}
```

```
namespace N1
{
    public static class D
    {
        public static void F(this int i) { Console.WriteLine("D.F({0})", i);
}
        public static void G(this int i) { Console.WriteLine("D.G({0})", i);
}
    }
}

namespace N2
{
    using N1;

    public static class E
    {
        public static void F(this int i) { Console.WriteLine("E.F({0})", i);
}
    }

    class Test
    {
        static void Main(string[] args)
        {
            1.F();
            2.G();
            3.H();
        }
    }
}
```

La sortie de cet exemple est la suivante :

Console

```
E.F(1)
D.G(2)
C.H(3)
```

`D.G` est prioritaire sur `C.G` `E.F` et est prioritaire sur `D.F` et `C.F` .

Appels délégués

Pour un appel de délégué, le *primary_expression* du *invocation_expression* doit être une valeur d'un *delegate_type*. En outre, si l'on considère que le *delegate_type* être une fonction membre avec la même liste de paramètres que l' *delegate_type*, la *delegate_type*

doit être applicable (membre de fonction applicable) en ce qui concerne la *argument_list* du *invocation_expression*.

Le traitement au moment de l'exécution d'un appel de délégué de la forme `D(A)` , où `D` est un *primary_expression* d'un *delegate_type* et `A` est un *argument_list* facultatif, se compose des étapes suivantes :

- `D` est évalué. Si cette évaluation provoque une exception, aucune autre étape n'est exécutée.
- La valeur de `D` est vérifiée comme valide. Si la valeur de `D` est `null` , une `System.NullReferenceException` est levée et aucune autre étape n'est exécutée.
- Sinon, `D` est une référence à une instance de délégué. Les appels de membre de fonction (vérification au moment de la compilation de la résolution de surcharge dynamique) sont exécutés sur chacune des entités pouvant être appelées dans la liste d'appel du délégué. Pour les entités pouvant être appelées constituées d'une méthode d'instance et d'instance, l'instance de l'appel est l'instance contenue dans l'entité pouvant être appelée.

Accès aux éléments

Un *element_access* se compose d'un *primary_no_array_creation_expression*, suivi d'un `[` jeton « », suivi d'un *argument_list*, suivi d'un `]` jeton « ». Le *argument_list* se compose d'un ou plusieurs *arguments*, séparés par des virgules.

antlr

```
element_access
    : primary_no_array_creation_expression '[' expression_list ']'
    ;
```

La *argument_list* d'un *element_access* n'est pas autorisée à contenir des `ref` `out` arguments ou.

Un *element_access* est lié dynamiquement (liaison dynamique) si au moins l'un des éléments suivants est présent :

- Le *primary_no_array_creation_expression* a le type au moment de la compilation `dynamic` .
- Au moins une expression de la *argument_list* a un type au moment de la compilation `dynamic` et le *primary_no_array_creation_expression* n'a pas de type tableau.

Dans ce cas, le compilateur classe la *element_access* comme une valeur de type `dynamic` . Les règles ci-dessous pour déterminer la signification du *element_access* sont ensuite appliquées au moment de l'exécution, à l'aide du type au moment de l'exécution au lieu du type de compilation de celles du *primary_no_array_creation_expression* et des expressions *argument_list* qui ont le type au moment de la compilation `dynamic` . Si le *primary_no_array_creation_expression* n'a pas de type au moment de la compilation `dynamic` , l'accès à l'élément subit une vérification limitée au moment de la compilation, comme décrit dans contrôle de la résolution de surcharge dynamique au moment de la compilation.

Si la *primary_no_array_creation_expression* d'un *element_access* est une valeur d'un *array_type*, l' *element_access* est un accès à un tableau (accès au tableau). Sinon, le *primary_no_array_creation_expression* doit être une variable ou une valeur d'un type de classe, de struct ou d'interface qui possède un ou plusieurs membres de l'indexeur, auquel cas le *element_access* est un accès à un indexeur (accès à l'indexeur).

Accès aux tableaux

Pour un accès à un tableau, le *primary_no_array_creation_expression* du *element_access* doit être une valeur d'un *array_type*. En outre, le *argument_list* d'un accès à un tableau n'est pas autorisé à contenir des arguments nommés. Le nombre d'expressions dans le *argument_list* doit être le même que le rang du *array_type*, et chaque expression doit être de type `int` ,, `uint` , `long` `ulong` ou doit être implicitement convertible en un ou plusieurs de ces types.

Le résultat de l'évaluation d'un accès à un tableau est une variable du type d'élément du tableau, à savoir l'élément de tableau sélectionné par la ou les valeurs de la ou des expressions de la *argument_list*.

Le traitement au moment de l'exécution d'un accès à un tableau de la forme `P[A]` , où `P` est un *primary_no_array_creation_expression* d'un *array_type* et `A` est un *argument_list*, se compose des étapes suivantes :

- `P` est évalué. Si cette évaluation provoque une exception, aucune autre étape n'est exécutée.
- Les expressions d'index des *argument_list* sont évaluées dans l'ordre, de gauche à droite. Après l'évaluation de chaque expression d'index, une conversion implicite (conversions implicites) en l'un des types suivants est effectuée : `int` , `uint` , `long` , `ulong` . Le premier type de cette liste pour lequel une conversion implicite existe est choisi. Par exemple, si l'expression d'index est de type, `short` une conversion implicite vers `int` est effectuée, puisque les conversions implicites de `short` vers

`int` et de à `short` `long` sont possibles. Si l'évaluation d'une expression d'index ou de la conversion implicite suivante provoque une exception, aucune autre expression d'index n'est évaluée et aucune autre étape n'est exécutée.

- La valeur de `P` est vérifiée comme valide. Si la valeur de `P` est `null` , une `System.NullReferenceException` est levée et aucune autre étape n'est exécutée.
- La valeur de chaque expression dans le *argument_list* est vérifiée par rapport aux limites réelles de chaque dimension de l'instance de tableau référencée par `P` . Si une ou plusieurs valeurs sont hors limites, une `System.IndexOutOfRangeException` est levée et aucune autre étape n'est exécutée.
- L'emplacement de l'élément de tableau donné par la ou les expressions d'index est calculé et cet emplacement devient le résultat de l'accès au tableau.

Accès aux indexeurs

Pour un accès à un indexeur, le *primary_no_array_creation_expression* du *element_access* doit être une variable ou une valeur d'un type de classe, de struct ou d'interface, et ce type doit implémenter un ou plusieurs indexeurs applicables en ce qui concerne le *argument_list* du *element_access*.

Le traitement au moment de la liaison d'un accès à un indexeur de la forme `P[A]` , où `P` est un *primary_no_array_creation_expression* d'un type de classe, de struct ou d'interface `T` , et `A` est un *argument_list*, se compose des étapes suivantes :

- L'ensemble d'indexeurs fourni par `T` est construit. L'ensemble se compose de tous les indexeurs déclarés dans `T` ou d'un type de base de `T` qui ne sont pas des `override` déclarations et qui sont accessibles dans le contexte actuel (accès aux membres).
- Le jeu est réduit aux indexeurs applicables et non masqués par d'autres indexeurs. Les règles suivantes sont appliquées à chaque indexeur du `S.I` jeu, où `S` est le type dans lequel l'indexeur `I` est déclaré :
  - Si `I` n'est pas applicable en ce qui concerne `A` (fonction membre applicable), `I` est supprimé de l'ensemble.
  - Si `I` est applicable en ce qui concerne `A` (fonction membre applicable), tous les indexeurs déclarés dans un type de base de `S` sont supprimés du jeu.
  - Si `I` est applicable en ce qui concerne `A` (fonction membre applicable) et `S` est un type de classe autre que `object` , tous les indexeurs déclarés dans une interface sont supprimés du jeu.
- Si le jeu d'indexeurs candidats obtenu est vide, aucun indexeur applicable n'existe et une erreur de liaison est générée.

- Le meilleur indexeur de l'ensemble des indexeurs candidats est identifié à l'aide des règles de résolution de surcharge de la résolution de surcharge. Si un seul meilleur indexeur ne peut pas être identifié, l'accès à l'indexeur est ambigu et une erreur de liaison s'est produite.
- Les expressions d'index des *argument_list* sont évaluées dans l'ordre, de gauche à droite. Le résultat du traitement de l'accès à l'indexeur est une expression classée comme un accès à un indexeur. L'expression d'accès de l'indexeur fait référence à l'indexeur déterminé à l'étape ci-dessus et a une expression d'instance associée de `P` et une liste d'arguments associée de `A` .

En fonction du contexte dans lequel il est utilisé, un accès à l'indexeur entraîne l'appel de l' *accesseur Get* ou de l' *accesseur Set* de l'indexeur. Si l'accès à l'indexeur est la cible d'une assignation, l' *accesseur Set* est appelé pour assigner une nouvelle valeur (assignation simple). Dans tous les autres cas, l' *accesseur Get* est appelé pour obtenir la valeur actuelle (valeurs des expressions).

Cet accès

Un *This_Access* se compose du mot réservé `this` .

antlr

```
this_access
    : 'this'
    ;
```

Une *This_Access* est autorisée uniquement dans le *bloc* d'un constructeur d'instance, d'une méthode d'instance ou d'un accesseur d'instance. Il a l'une des significations suivantes :

- Quand `this` est utilisé dans un *primary_expression* au sein d'un constructeur d'instance d'une classe, il est classé comme une valeur. Le type de la valeur est le type d'instance (type d'instance) de la classe dans laquelle l'utilisation se produit, et la valeur est une référence à l'objet en cours de construction.
- Quand `this` est utilisé dans un *primary_expression* dans une méthode d'instance ou un accesseur d'instance d'une classe, il est classé comme une valeur. Le type de la valeur est le type d'instance (type d'instance) de la classe dans laquelle l'utilisation se produit, et la valeur est une référence à l'objet pour lequel la méthode ou l'accesseur a été appelé.
- Quand `this` est utilisé dans un *primary_expression* dans un constructeur d'instance d'un struct, il est classé en tant que variable. Le type de la variable est le type d'instance (type d'instance) de la structure dans laquelle l'utilisation se produit, et

la variable représente le struct en cours de construction. La `this` variable d'un constructeur d'instance d'un struct se comporte exactement comme un `out` paramètre du type struct (en particulier, cela signifie que la variable doit être assignée de manière définitive dans chaque chemin d'exécution du constructeur d'instance).

- Quand `this` est utilisé dans un *primary_expression* dans une méthode d'instance ou un accesseur d'instance d'un struct, il est classé en tant que variable. Le type de la variable est le type d'instance (type d'instance) de la structure dans laquelle l'utilisation se produit.
  - Si la méthode ou l'accesseur n'est pas un itérateur (itérateurs), la `this` variable représente le struct pour lequel la méthode ou l'accesseur a été appelé, et se comporte exactement comme un `ref` paramètre du type struct.
  - Si la méthode ou l'accesseur est un itérateur, la `this` variable représente une copie du struct pour lequel la méthode ou l'accesseur a été appelé, et se comporte exactement comme un paramètre de valeur du type struct.

L'utilisation de `this` dans un *primary_expression* dans un contexte autre que ceux listés ci-dessus est une erreur de compilation. En particulier, il n'est pas possible de faire référence à `this` dans une méthode statique, un accesseur de propriété statique ou dans une *variable_initializer* d'une déclaration de champ.

Accès de base

Un *base_access* se compose du mot réservé `base` suivi d'un jeton « `.` » et d'un identificateur ou d'un *argument_list* placé entre crochets :

antlr

```
base_access
    : 'base' '.' identifier
    | 'base' '[' expression_list ']'
    ;
```

Un *base_access* est utilisé pour accéder aux membres de la classe de base qui sont masqués par des membres portant le même nom dans la classe ou le struct actuel. Une *base_access* est autorisée uniquement dans le *bloc* d'un constructeur d'instance, d'une méthode d'instance ou d'un accesseur d'instance. Quand `base.I` se produit dans une classe ou un struct, `I` doit désigner un membre de la classe de base de cette classe ou de ce struct. De même, lorsque `base[E]` se produit dans une classe, un indexeur applicable doit exister dans la classe de base.

Au moment de la liaison, *base_access* expressions de la forme `base.I` et `base[E]` sont évaluées exactement comme si elles étaient écrites `((B)this).I` et `((B)this)[E]` , où `B` est la classe de base de la classe ou du struct dans lequel la construction se produit. Ainsi, `base.I` et `base[E]` correspondent à `this.I` et `this[E]` , sauf `this` que est affiché en tant qu'instance de la classe de base.

Lorsqu'un *base_access* fait référence à un membre de fonction virtuelle (une méthode, une propriété ou un indexeur), la détermination du membre de fonction à appeler au moment de l'exécution (vérification de la résolution de surcharge dynamiqueau moment de la compilation) est modifiée. Le membre de fonction appelé est déterminé par la recherche de l'implémentation la plus dérivée (méthodes virtuelles) du membre de fonction par rapport à `B` (au lieu de par rapport au type d'exécution de `this` , comme c'est normalement le cas dans un accès non de base). Ainsi, dans un `override` d'un `virtual` membre de fonction, un *base_access* peut être utilisé pour appeler l'implémentation héritée du membre de fonction. Si le membre de fonction référencé par un *base_access* est abstrait, une erreur de liaison au moment de la liaison se produit.

Opérateurs suffixés d'incrémentation et de décrémentation

antlr

```
post_increment_expression
    : primary_expression '++'
    ;

post_decrement_expression
    : primary_expression '--'
    ;
```

L'opérande d'une opération d'incrémentation ou de décrémentation suffixée doit être une expression classifiée comme une variable, un accès à une propriété ou un accès à un indexeur. Le résultat de l'opération est une valeur du même type que l'opérande.

Si le *primary_expression* a le type au moment de la compilation `dynamic` , l'opérateur est lié dynamiquement (liaison dynamique), le *post_increment_expression* ou *post_decrement_expression* a le type au moment de la compilation `dynamic` et les règles suivantes sont appliquées au moment de l'exécution en utilisant le type d'exécution de l' *primary_expression*.

Si l'opérande d'une opération d'incrémentation ou de décrémentation suffixée est un accès à une propriété ou un indexeur, la propriété ou l'indexeur doit avoir à la fois un

`get` `set` accesseur et un accesseur. Si ce n'est pas le cas, une erreur de temps de liaison se produit.

La résolution de surcharge d'opérateur unaire (résolution de surcharge d'opérateur unaire) est appliquée pour sélectionner une implémentation d'opérateur spécifique. `++`Des opérateurs et prédéfinis `--` existent pour les types suivants : `sbyte` , `byte` , `short` ,,, `ushort` `int` `uint` , `long` , `ulong` , `char` , `float` , `double` , `decimal` et tout type enum. Les opérateurs prédéfinis `++` retournent la valeur produite par l'ajout de 1 à l'opérande, et les opérateurs prédéfinis `--` retournent la valeur produite par la soustraction de 1 de l'opérande. Dans un `checked` contexte, si le résultat de l'addition ou de la soustraction est en dehors de la plage du type de résultat et que le type de résultat est un type intégral ou un type enum, une `System.OverflowException` est levée.

Le traitement au moment de l'exécution d'une opération d'incrémentation ou de décrémentation suffixée du formulaire `x++` ou `x--` se compose des étapes suivantes :

- Si `x` est classé en tant que variable :
  - `x` est évalué pour produire la variable.
  - La valeur de `x` est enregistrée.
  - L'opérateur sélectionné est appelé avec la valeur enregistrée de `x` comme argument.
  - La valeur retournée par l'opérateur est stockée dans l'emplacement donné par l'évaluation de `x` .
  - La valeur enregistrée de `x` devient le résultat de l'opération.
- Si `x` est classifié comme un accès à une propriété ou un indexeur :
  - L'expression d'instance (si `x` n'est pas `static`) et la liste d'arguments (si `x` est un accès à un indexeur) associé à `x` sont évaluées, et les résultats sont utilisés dans les `get` appels suivants et d' `set` accesseur.
  - L' `get` accesseur de `x` est appelé et la valeur retournée est enregistrée.
  - L'opérateur sélectionné est appelé avec la valeur enregistrée de `x` comme argument.
  - L' `set` accesseur de `x` est appelé avec la valeur retournée par l'opérateur comme son `value` argument.
  - La valeur enregistrée de `x` devient le résultat de l'opération.

Les `++` `--` opérateurs et prennent également en charge la notation de préfixe (préfixe d'incrémentation et de décrémentation). En général, le résultat de `x++` ou `x--` est la valeur de `x` avant l'opération, alors que le résultat de `++x` ou `--x` est la valeur de `x` après l'opération. Dans les deux cas, `x` elle a la même valeur après l'opération.

Une `operator ++` `operator --` implémentation ou peut être appelée à l'aide de la notation suffixée ou de préfixe. Il n'est pas possible d'avoir des implémentations d'opérateur distinctes pour les deux notations.

new, opérateur

L' `new` opérateur est utilisé pour créer de nouvelles instances de types.

Il existe trois formes d' `new` expressions :

- Les expressions de création d'objet sont utilisées pour créer des instances de types de classe et de valeur.
- Les expressions de création de tableau sont utilisées pour créer de nouvelles instances de types tableau.
- Les expressions de création de délégué sont utilisées pour créer de nouvelles instances de types délégués.

L' `new` opérateur implique la création d'une instance d'un type, mais n'implique pas nécessairement l'allocation dynamique de la mémoire. En particulier, les instances de types valeur ne requièrent aucune mémoire supplémentaire au-delà des variables dans lesquelles elles résident et aucune allocation dynamique ne se produit lorsque `new` est utilisé pour créer des instances de types valeur.

Expressions de création d'objet

Un *object_creation_expression* est utilisé pour créer une nouvelle instance d'un *class_type* ou d'un *Value_type*.

antlr

```
object_creation_expression
    : 'new' type '(' argument_list? ')' object_or_collection_initializer?
    | 'new' type object_or_collection_initializer
    ;

object_or_collection_initializer
    : object_initializer
    | collection_initializer
    ;
```

Le *type* d'un *object_creation_expression* doit être un *class_type*, un *Value_type* ou un *type_parameter*. Le *type* ne peut pas être un `abstract` *class_type*.

La *argument_list* facultative (listes d'arguments) est autorisée uniquement si le *type* est un *class_type* ou un *struct_type*.

Une expression de création d'objet peut omettre la liste d'arguments de constructeur et les parenthèses englobantes fournies, elle comprend un initialiseur d'objet ou un initialiseur de collection. L'omission de la liste d'arguments du constructeur et des parenthèses englobantes équivaut à spécifier une liste d'arguments vide.

Le traitement d'une expression de création d'objet qui inclut un initialiseur d'objet ou un initialiseur de collection consiste à traiter en premier le constructeur d'instance, puis à traiter les initialisations de membre ou d'élément spécifiées par l'initialiseur d'objet (initialiseurs d'objets) ou l'initialiseur de collection (initialiseurs de collection).

Si l'un des arguments de la *argument_list* facultative a le type au moment de la compilation `dynamic` , le *object_creation_expression* est lié dynamiquement (liaison dynamique) et les règles suivantes sont appliquées au moment de l'exécution à l'aide du type d'exécution des arguments de l' *argument_list* qui ont le type au moment de la compilation `dynamic` . Toutefois, la création d'objet subit une vérification limitée au moment de la compilation, comme décrit dans vérification de la résolution de surcharge dynamiquepar la compilation.

Le traitement au moment de la liaison d'une *object_creation_expression* de la forme `new T(A)` , où `T` est un *class_type* ou un *Value_type* et `A` est un *argument_list* facultatif, se compose des étapes suivantes :

- Si `T` est un *Value_type* et `A` n'est pas présent :
  - Le *object_creation_expression* est un appel de constructeur par défaut. Le résultat de l' *object_creation_expression* est une valeur de type `T` , à savoir la valeur par défaut de `T` telle que définie dans le type System. ValueType.
- Sinon, si `T` est un *type_parameter* et `A` n'est pas présent :
  - Si aucune contrainte de type valeur ou contrainte de constructeur (contraintes de paramètre de type) n'a été spécifiée pour `T` , une erreur de liaison s'est produite.
  - Le résultat de l' *object_creation_expression* est une valeur du type au moment de l'exécution auquel le paramètre de type a été lié, à savoir le résultat de l'appel du constructeur par défaut de ce type. Le type au moment de l'exécution peut être un type référence ou un type valeur.
- Sinon, si `T` est un *class_type* ou un *struct_type*:
  - Si `T` est un `abstract` *class_type*, une erreur de compilation se produit.
  - Le constructeur d'instance à appeler est déterminé à l'aide des règles de résolution de surcharge de la résolution de surcharge. L'ensemble de constructeurs d'instance candidat est constitué de tous les constructeurs

d'instance accessibles déclarés dans `T` qui sont applicables en ce qui concerne `A` (fonction membre applicable). Si le jeu de constructeurs d'instance candidat est vide, ou si un seul constructeur d'instance unique ne peut pas être identifié, une erreur de liaison s'est produite.
    - Le résultat de l' *object_creation_expression* est une valeur de type `T` , à savoir la valeur produite par l'appel du constructeur d'instance déterminé à l'étape ci-dessus.
- Sinon, la *object_creation_expression* n'est pas valide et une erreur de liaison s'est produite.

Même si le *object_creation_expression* est lié dynamiquement, le type au moment de la compilation est toujours `T` .

Le traitement au moment de l'exécution d'une *object_creation_expression* de la forme `new T(A)` , où `T` est *class_type* ou un *struct_type* et `A` est un *argument_list* facultatif, se compose des étapes suivantes :

- Si `T` est un *class_type*:
    - Une nouvelle instance de la classe `T` est allouée. Si la mémoire disponible est insuffisante pour allouer la nouvelle instance, une `System.OutOfMemoryException` est levée et aucune autre étape n'est exécutée.
    - Tous les champs de la nouvelle instance sont initialisés à leurs valeurs par défaut (valeurs par défaut).
    - Le constructeur d'instance est appelé selon les règles d'appel de membre de fonction (vérification au moment de la compilation de la résolution de surcharge dynamique). Une référence à l'instance qui vient d'être allouée est automatiquement passée au constructeur d'instance et l'instance est accessible à partir de ce constructeur en tant que `this` .
- Si `T` est un *struct_type*:
    - Une instance de type `T` est créée en allouant une variable locale temporaire. Étant donné qu'un constructeur d'instance d'un *struct_type* est requis pour assigner une valeur à chaque champ de l'instance en cours de création, aucune initialisation de la variable temporaire n'est nécessaire.
    - Le constructeur d'instance est appelé selon les règles d'appel de membre de fonction (vérification au moment de la compilation de la résolution de surcharge dynamique). Une référence à l'instance qui vient d'être allouée est automatiquement passée au constructeur d'instance et l'instance est accessible à partir de ce constructeur en tant que `this` .

Initialiseurs d'objet

Un ***initialiseur d'objet*** spécifie des valeurs pour zéro, un ou plusieurs champs, propriétés ou éléments indexés d'un objet.

antlr

```
object_initializer
    : '{' member_initializer_list? '}'
    | '{' member_initializer_list ',' '}'
    ;

member_initializer_list
    : member_initializer (',' member_initializer)*
    ;

member_initializer
    : initializer_target '=' initializer_value
    ;

initializer_target
    : identifier
    | '[' argument_list ']'
    ;

initializer_value
    : expression
    | object_or_collection_initializer
    ;
```

Un initialiseur d'objet se compose d'une séquence d'initialiseurs de membres, délimités par des `{` `}` jetons et séparés par des virgules. Chaque *member_initializer* désigne une cible pour l'initialisation. Un *identificateur* doit nommer un champ ou une propriété accessible de l'objet en cours d'initialisation, alors qu'un *argument_list* placé entre crochets doit spécifier des arguments pour un indexeur accessible sur l'objet en cours d'initialisation. Il y a une erreur pour qu'un initialiseur d'objet inclue plusieurs initialiseurs de membre pour le même champ ou la même propriété.

Chaque *initializer_target* est suivi d'un signe égal et d'une expression, d'un initialiseur d'objet ou d'un initialiseur de collection. Les expressions dans l'initialiseur d'objet ne peuvent pas faire référence à l'objet nouvellement créé qu'il initialise.

Un initialiseur de membre qui spécifie une expression après que le signe égal est traité de la même façon qu'une assignation (assignation simple) à la cible.

Un initialiseur de membre qui spécifie un initialiseur d'objet après que le signe égal est un ***initialiseur d'objet imbriqué***, c.-à-d. une initialisation d'un objet incorporé. Au lieu d'assigner une nouvelle valeur au champ ou à la propriété, les assignations dans l'initialiseur d'objet imbriqué sont traitées comme des assignations aux membres du

champ ou de la propriété. Les initialiseurs d'objets imbriqués ne peuvent pas être appliqués à des propriétés avec un type valeur, ou à des champs en lecture seule avec un type valeur.

Un initialiseur de membre qui spécifie un initialiseur de collection après le signe égal est l'initialisation d'une collection incorporée. Au lieu d'assigner une nouvelle collection au champ, à la propriété ou à l'indexeur cible, les éléments fournis dans l'initialiseur sont ajoutés à la collection référencée par la cible. La cible doit être d'un type de collection qui satisfait aux exigences spécifiées dans les initialiseurs de collection.

Les arguments d'un initialiseur d'index seront toujours évalués une seule fois. Ainsi, même si les arguments finissent par ne jamais être utilisés (par exemple, en raison d'un initialiseur imbriqué vide), ils sont évalués pour leurs effets secondaires.

La classe suivante représente un point avec deux coordonnées :

C#

```
public class Point
{
    int x, y;

    public int X { get { return x; } set { x = value; } }
    public int Y { get { return y; } set { y = value; } }
}
```

Une instance de `Point` peut être créée et initialisée comme suit :

C#

```
Point a = new Point { X = 0, Y = 1 };
```

ce qui a le même effet que

C#

```
Point __a = new Point();
__a.X = 0;
__a.Y = 1;
Point a = __a;
```

où `__a` est une variable temporaire inaccessible, invisible et inaccessible. La classe suivante représente un rectangle créé à partir de deux points :

C#

```
public class Rectangle
{
    Point p1, p2;

    public Point P1 { get { return p1; } set { p1 = value; } }
    public Point P2 { get { return p2; } set { p2 = value; } }
}
```

Une instance de `Rectangle` peut être créée et initialisée comme suit :

C#

```
Rectangle r = new Rectangle {
    P1 = new Point { X = 0, Y = 1 },
    P2 = new Point { X = 2, Y = 3 }
};
```

ce qui a le même effet que

C#

```
Rectangle __r = new Rectangle();
Point __p1 = new Point();
__p1.X = 0;
__p1.Y = 1;
__r.P1 = __p1;
Point __p2 = new Point();
__p2.X = 2;
__p2.Y = 3;
__r.P2 = __p2;
Rectangle r = __r;
```

où `__r` `__p1` et `__p2` sont des variables temporaires qui, sinon, sont inaccessibles et inaccessibles.

Si `Rectangle` le constructeur de l'alloue les deux `Point` instances incorporées

C#

```
public class Rectangle
{
    Point p1 = new Point();
    Point p2 = new Point();

    public Point P1 { get { return p1; } }
    public Point P2 { get { return p2; } }
}
```

la construction suivante peut être utilisée pour initialiser les instances incorporées `Point` au lieu d'assigner de nouvelles instances :

C#

```
Rectangle r = new Rectangle {
    P1 = { X = 0, Y = 1 },
    P2 = { X = 2, Y = 3 }
};
```

ce qui a le même effet que

C#

```
Rectangle __r = new Rectangle();
__r.P1.X = 0;
__r.P1.Y = 1;
__r.P2.X = 2;
__r.P2.Y = 3;
Rectangle r = __r;
```

Dans le cas d'une définition appropriée de C, l'exemple suivant :

C#

```
var c = new C {
    x = true,
    y = { a = "Hello" },
    z = { 1, 2, 3 },
    ["x"] = 5,
    [0,0] = { "a", "b" },
    [1,2] = {}
};
```

équivaut à cette série d'affectations :

C#

```
C __c = new C();
__c.x = true;
__c.y.a = "Hello";
__c.z.Add(1);
__c.z.Add(2);
__c.z.Add(3);
string __i1 = "x";
__c[__i1] = 5;
int __i2 = 0, __i3 = 0;
__c[__i2,__i3].Add("a");
__c[__i2,__i3].Add("b");
```

```
int __i4 = 1, __i5 = 2;
var c = __c;
```

où `__c` , etc., sont des variables générées qui sont invisibles et inaccessibles au code source. Notez que les arguments de `[0,0]` sont évalués une seule fois, et les arguments de `[1,2]` sont évalués une seule fois, même s'ils ne sont jamais utilisés.

Initialiseurs de collection

Un initialiseur de collection spécifie les éléments d'une collection.

antlr

```
collection_initializer
    : '{' element_initializer_list '}'
    | '{' element_initializer_list ',' '}'
    ;

element_initializer_list
    : element_initializer (',' element_initializer)*
    ;

element_initializer
    : non_assignment_expression
    | '{' expression_list '}'
    ;

expression_list
    : expression (',' expression)*
    ;
```

Un initialiseur de collection se compose d'une séquence d'initialiseurs d'éléments, encadrés par des `{` `}` jetons et séparés par des virgules. Chaque initialiseur d'élément spécifie un élément à ajouter à l'objet de collection qui est initialisé, et se compose d'une liste d'expressions encadrées par `{` des `}` jetons et séparés par des virgules. Un initialiseur d'élément à une seule expression peut être écrit sans accolades, mais il ne peut pas être une expression d'assignation, afin d'éviter toute ambiguïté avec les initialiseurs de membres. La *non_assignment_expression* production est définie dans expression.

L'exemple suivant illustre une expression de création d'objet qui comprend un initialiseur de collection :

C#

```
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

L'objet de collection auquel un initialiseur de collection est appliqué doit être d'un type qui implémente `System.Collections.IEnumerable` ou une erreur de compilation se produit. Pour chaque élément spécifié dans l'ordre, l'initialiseur de collection appelle une `Add` méthode sur l'objet cible avec la liste d'expressions de l'initialiseur d'élément comme liste d'arguments, en appliquant une recherche de membre normale et une résolution de surcharge pour chaque appel. Ainsi, l'objet de collection doit avoir une instance applicable ou une méthode d'extension avec le nom `Add` de chaque initialiseur d'élément.

La classe suivante représente un contact avec un nom et une liste de numéros de téléphone :

C#

```
public class Contact
{
    string name;
    List<string> phoneNumbers = new List<string>();

    public string Name { get { return name; } set { name = value; } }

    public List<string> PhoneNumbers { get { return phoneNumbers; } }
}
```

Une `List<Contact>` peut être créée et initialisée comme suit :

C#

```
var contacts = new List<Contact> {
    new Contact {
        Name = "Chris Smith",
        PhoneNumbers = { "206-555-0101", "425-882-8080" }
    },
    new Contact {
        Name = "Bob Harris",
        PhoneNumbers = { "650-555-0199" }
    }
};
```

ce qui a le même effet que

C#

```
var __clist = new List<Contact>();
Contact __c1 = new Contact();
__c1.Name = "Chris Smith";
```

```
__c1.PhoneNumbers.Add("206-555-0101");
__c1.PhoneNumbers.Add("425-882-8080");
__clist.Add(__c1);
Contact __c2 = new Contact();
__c2.Name = "Bob Harris";
__c2.PhoneNumbers.Add("650-555-0199");
__clist.Add(__c2);
var contacts = __clist;
```

où `__clist` `__c1` et `__c2` sont des variables temporaires qui, sinon, sont inaccessibles et inaccessibles.

Expressions de création de tableau

Un *array_creation_expression* est utilisé pour créer une nouvelle instance d'une *array_type*.

antlr

```
array_creation_expression
    : 'new' non_array_type '[' expression_list ']' rank_specifier*
array_initializer?
    | 'new' array_type array_initializer
    | 'new' rank_specifier array_initializer
    ;
```

Une expression de création de tableau de la première forme alloue une instance de tableau du type qui résulte de la suppression de chacune des expressions individuelles de la liste d'expressions. Par exemple, l'expression de création `new int[10,20]` de tableau produit une instance de tableau de type `int[,]` , et l'expression de création `new int[10][,]` de tableau produit un tableau de type `int[][,]` . Chaque expression de la liste d'expressions doit être de type `int` ,, `uint` `long` ou `ulong` , ou implicitement convertible en un ou plusieurs de ces types. La valeur de chaque expression détermine la longueur de la dimension correspondante dans l'instance de tableau qui vient d'être allouée. Dans la mesure où la longueur d'une dimension de tableau ne doit pas être négative, il s'agit d'une erreur au moment de la compilation pour avoir un *constant_expression* avec une valeur négative dans la liste d'expressions.

Sauf dans un contexte non sécurisé (contextes non sécurisés), la disposition des tableaux n'est pas spécifiée.

Si une expression de création de tableau du premier formulaire comprend un initialiseur de tableau, chaque expression de la liste d'expressions doit être une constante et le rang

et les longueurs de dimensions spécifiés par la liste d'expressions doivent correspondre à ceux de l'initialiseur de tableau.

Dans une expression de création de tableau de la deuxième ou de la troisième forme, le rang du type de tableau ou du spécificateur de rangs spécifié doit correspondre à celui de l'initialiseur de tableau. Les longueurs de dimensions individuelles sont déduites du nombre d'éléments dans chacun des niveaux d'imbrication correspondants de l'initialiseur de tableau. Ainsi, l'expression

```
C#

new int[,] {{0, 1}, {2, 3}, {4, 5}}
```

correspond exactement à

```
C#

new int[3, 2] {{0, 1}, {2, 3}, {4, 5}}
```

Une expression de création de tableau de la troisième forme est connue sous le terme d'une **expression de création de tableau typé implicitement** _. Elle est similaire à la deuxième forme, à ceci près que le type d'élément du tableau n'est pas explicitement donné, mais déterminé comme le meilleur type commun (recherche du meilleur type commun d'un ensemble d'expressions) du jeu d'expressions dans l'initialiseur de tableau. Dans le cas d'un tableau multidimensionnel, c'est-à-dire un où le _rank_specifier * contient au moins une virgule, cet ensemble comprend toutes les *expressions* qui se trouvent dans les *array_initializer* imbriqué s.

Les initialiseurs de tableau sont décrits plus en détail dans les initialiseurs de tableau.

Le résultat de l'évaluation d'une expression de création de tableau est classé comme une valeur, à savoir une référence à l'instance de tableau nouvellement allouée. Le traitement au moment de l'exécution d'une expression de création de tableau se compose des étapes suivantes :

- Les expressions de longueur de dimension des *expression_list* sont évaluées dans l'ordre, de gauche à droite. Après l'évaluation de chaque expression, une conversion implicite (conversions implicites) vers l'un des types suivants est effectuée : `int` , `uint` , `long` , `ulong` . Le premier type de cette liste pour lequel une conversion implicite existe est choisi. Si l'évaluation d'une expression ou de la conversion implicite suivante provoque une exception, aucune autre expression n'est évaluée et aucune autre étape n'est exécutée.

- Les valeurs calculées pour les longueurs de dimensions sont validées comme suit. Si une ou plusieurs valeurs sont inférieures à zéro, une `System.OverflowException` est levée et aucune autre étape n'est exécutée.
- Une instance de tableau avec les longueurs de dimensions données est allouée. Si la mémoire disponible est insuffisante pour allouer la nouvelle instance, une `System.OutOfMemoryException` est levée et aucune autre étape n'est exécutée.
- Tous les éléments de la nouvelle instance de tableau sont initialisés à leurs valeurs par défaut (valeurs par défaut).
- Si l'expression de création de tableau contient un initialiseur de tableau, chaque expression de l'initialiseur de tableau est évaluée et assignée à son élément de tableau correspondant. Les évaluations et les assignations sont effectuées dans l'ordre dans lequel les expressions sont écrites dans l'initialiseur de tableau. en d'autres termes, les éléments sont initialisés dans l'ordre d'index de croissance, avec la dimension la plus à droite qui s'incrémente en premier. Si l'évaluation d'une expression donnée ou de l'assignation suivante à l'élément de tableau correspondant provoque une exception, aucun autre élément n'est initialisé (et les éléments restants auront donc leurs valeurs par défaut).

Une expression de création de tableau autorise l'instanciation d'un tableau avec des éléments d'un type tableau, mais les éléments d'un tel tableau doivent être initialisés manuellement. Par exemple, l'instruction

C#

```
int[][] a = new int[100][];
```

crée un tableau unidimensionnel avec 100 éléments de type `int[]` . La valeur initiale de chaque élément est `null` . Il n'est pas possible pour la même expression de création de tableau d'instancier également les sous-tableaux, et l'instruction

C#

```
int[][] a = new int[100][5];        // Error
```

génère une erreur au moment de la compilation. L'instanciation des sous-tableaux doit plutôt être effectuée manuellement, comme dans

C#

```
int[][] a = new int[100][];
for (int i = 0; i < 100; i++) a[i] = new int[5];
```

Lorsqu'un tableau de tableaux a une forme « rectangulaire », c'est-à-dire lorsque les sous-tableaux sont tous de la même longueur, il est plus efficace d'utiliser un tableau multidimensionnel. Dans l'exemple ci-dessus, l'instanciation du tableau de tableaux crée 101 objets, un tableau externe et des sous-tableaux 100. En revanche,

C#

```
int[,] = new int[100, 5];
```

crée un seul objet, un tableau à deux dimensions, et accomplit l'allocation dans une instruction unique.

Voici des exemples d'expressions de création de tableau implicitement typées :

C#

```
var a = new[] { 1, 10, 100, 1000 };                     // int[]

var b = new[] { 1, 1.5, 2, 2.5 };                       // double[]

var c = new[,] { { "hello", null }, { "world", "!" } }; // string[,]

var d = new[] { 1, "one", 2, "two" };                   // Error
```

La dernière expression provoque une erreur au moment de la compilation, car ni `int` ni `string` n'est implicitement convertible en l'autre, et il n'existe donc aucun meilleur type commun. Une expression de création de tableau explicitement typée doit être utilisée dans ce cas, par exemple, en spécifiant le type `object[]` . L'un des éléments peut également être casté en un type de base commun, qui devient alors le type d'élément inféré.

Les expressions de création de tableau implicitement typées peuvent être combinées avec des initialiseurs d'objets anonymes (expressions de création d'objets anonymes) pour créer des structures de données typées anonymement. Par exemple :

C#

```
var contacts = new[] {
    new {
        Name = "Chris Smith",
        PhoneNumbers = new[] { "206-555-0101", "425-882-8080" }
    },
    new {
        Name = "Bob Harris",
        PhoneNumbers = new[] { "650-555-0199" }
```

```
    }
};
```

Expressions de création de délégué

Un *delegate_creation_expression* est utilisé pour créer une nouvelle instance d'une *delegate_type*.

antlr

```
delegate_creation_expression
    : 'new' delegate_type '(' expression ')'
    ;
```

L'argument d'une expression de création de délégué doit être un groupe de méthodes, une fonction anonyme ou une valeur du type au moment de la compilation `dynamic` ou une *delegate_type*. Si l'argument est un groupe de méthodes, il identifie la méthode et, pour une méthode d'instance, l'objet pour lequel créer un délégué. Si l'argument est une fonction anonyme, il définit directement les paramètres et le corps de la méthode de la cible du délégué. Si l'argument est une valeur, il identifie une instance de délégué de laquelle créer une copie.

Si l' *expression* a le type au moment de la compilation `dynamic` , le *delegate_creation_expression* est lié dynamiquement (liaison dynamique) et les règles ci-dessous sont appliquées au moment de l'exécution en utilisant le type d'exécution de l' *expression*. Dans le cas contraire, les règles sont appliquées au moment de la compilation.

Le traitement au moment de la liaison d'une *delegate_creation_expression* de la forme `new D(E)` , où `D` est un *delegate_type* et `E` est une *expression*, se compose des étapes suivantes :

- Si `E` est un groupe de méthodes, l'expression de création de délégué est traitée de la même façon qu'une conversion de groupe de méthodes (conversions de groupe de méthodes) de `E` vers `D` .
- Si `E` est une fonction anonyme, l'expression de création de délégué est traitée de la même façon qu'une conversion de fonction anonyme (conversions de fonctions anonymes) de `E` en `D` .
- Si `E` est une valeur, `E` doit être compatible (déclarations de délégué) avec `D` , et le résultat est une référence à un délégué nouvellement créé de type `D` qui fait

référence à la même liste d'appel que `E` . Si `E` n'est pas compatible avec `D` , une erreur de compilation se produit.

Le traitement au moment de l'exécution d'une *delegate_creation_expression* de la forme `new D(E)` , où `D` est un *delegate_type* et `E` est une *expression*, se compose des étapes suivantes :

- Si `E` est un groupe de méthodes, l'expression de création de délégué est évaluée comme une conversion de groupe de méthodes (conversions de groupe de méthodes) de `E` en `D` .
- Si `E` est une fonction anonyme, la création du délégué est évaluée comme une conversion de fonction anonyme de `E` en `D` (conversions de fonctions anonymes).
- Si `E` est une valeur d'un *delegate_type*:
  - `E` est évalué. Si cette évaluation provoque une exception, aucune autre étape n'est exécutée.
  - Si la valeur de `E` est `null` , une `System.NullReferenceException` est levée et aucune autre étape n'est exécutée.
  - Une nouvelle instance du type délégué `D` est allouée. Si la mémoire disponible est insuffisante pour allouer la nouvelle instance, une `System.OutOfMemoryException` est levée et aucune autre étape n'est exécutée.
  - La nouvelle instance de délégué est initialisée avec la même liste d'appel que l'instance de délégué donnée par `E` .

La liste d'appel d'un délégué est déterminée lorsque le délégué est instancié, puis reste constante pendant toute la durée de vie du délégué. En d'autres termes, il n'est pas possible de modifier les entités pouvant être appelées par Target d'un délégué une fois qu'il a été créé. Quand deux délégués sont combinés ou s'ils sont supprimés d'un autre (déclarations déléguées), un nouveau délégué est obtenu ; aucun contenu n'a été modifié pour le délégué existant.

Il n'est pas possible de créer un délégué qui fait référence à une propriété, un indexeur, un opérateur défini par l'utilisateur, un constructeur d'instance, un destructeur ou un constructeur statique.

Comme décrit ci-dessus, quand un délégué est créé à partir d'un groupe de méthodes, la liste de paramètres formels et le type de retour du délégué déterminent les méthodes surchargées à sélectionner. Dans l'exemple

C#

```
delegate double DoubleFunc(double x);

class A
```

```
{
    DoubleFunc f = new DoubleFunc(Square);

    static float Square(float x) {
        return x * x;
    }

    static double Square(double x) {
        return x * x;
    }
}
```

le `A.f` champ est initialisé avec un délégué qui fait référence à la deuxième `Square` méthode, car cette méthode correspond exactement à la liste de paramètres formels et au type de retour de `DoubleFunc` . Si la seconde `Square` méthode était absente, une erreur de compilation s'est produite.

Expressions de création d'objets anonymes

Un *anonymous_object_creation_expression* est utilisé pour créer un objet d'un type anonyme.

antlr

```
anonymous_object_creation_expression
    : 'new' anonymous_object_initializer
    ;

anonymous_object_initializer
    : '{' member_declarator_list? '}'
    | '{' member_declarator_list ',' '}'
    ;

member_declarator_list
    : member_declarator (',' member_declarator)*
    ;

member_declarator
    : simple_name
    | member_access
    | base_access
    | null_conditional_member_access
    | identifier '=' expression
    ;
```

Un initialiseur d'objet anonyme déclare un type anonyme et retourne une instance de ce type. Un type anonyme est un type de classe sans type qui hérite directement de `object` . Les membres d'un type anonyme sont une séquence de propriétés en lecture seule

déduites de l'initialiseur d'objet anonyme utilisé pour créer une instance du type. Plus précisément, un initialiseur d'objet anonyme de la forme

```
C#

new { p1 = e1, p2 = e2, ..., pn = en }
```

déclare un type anonyme de la forme

```
C#

class __Anonymous1
{
    private readonly T1 f1;
    private readonly T2 f2;
    ...
    private readonly Tn fn;

    public __Anonymous1(T1 a1, T2 a2, ..., Tn an) {
        f1 = a1;
        f2 = a2;
        ...
        fn = an;
    }

    public T1 p1 { get { return f1; } }
    public T2 p2 { get { return f2; } }
    ...
    public Tn pn { get { return fn; } }

    public override bool Equals(object __o) { ... }
    public override int GetHashCode() { ... }
}
```

où each `Tx` est le type de l'expression correspondante `ex` . L'expression utilisée dans un *member_declarator* doit avoir un type. Par conséquent, il s'agit d'une erreur de compilation pour qu'une expression d'un *member_declarator* soit null ou une fonction anonyme. Il s'agit également d'une erreur au moment de la compilation pour que l'expression ait un type non sécurisé.

Les noms d'un type anonyme et du paramètre de sa `Equals` méthode sont générés automatiquement par le compilateur et ne peuvent pas être référencés dans le texte du programme.

Dans le même programme, deux initialiseurs d'objets anonymes qui spécifient une séquence de propriétés des mêmes noms et types au moment de la compilation dans le même ordre produisent des instances du même type anonyme.

Dans l'exemple

```C#
var p1 = new { Name = "Lawnmower", Price = 495.00 };
var p2 = new { Name = "Shovel", Price = 26.95 };
p1 = p2;
```

l'assignation sur la dernière ligne est autorisée, car `p1` et `p2` sont du même type anonyme.

Les `Equals` `GetHashcode` méthodes et sur les types anonymes substituent les méthodes héritées de `object` , et sont définies en termes de `Equals` et `GetHashcode` des propriétés, afin que deux instances du même type anonyme soient égales si et seulement si toutes leurs propriétés sont égales.

Un déclarateur de membre peut être abrégé en un nom simple (inférence de type), un accès de membre (vérification au moment de la compilation de la résolution de surcharge dynamique), un accès de base (accès de base) ou un accès de membre conditionnel null (expressions conditionnelles NULL comme initialiseurs de projection). C'est ce qu'on appelle un ***initialiseur de projection*** et est un raccourci pour une déclaration et une assignation à une propriété du même nom. Plus précisément, les déclarateurs de membres des formulaires

```C#
identifier
expr.identifier
```

sont exactement équivalents à ce qui suit, respectivement :

```C#
identifier = identifier
identifier = expr.identifier
```

Ainsi, dans un initialiseur de projection, l' *identificateur* sélectionne à la fois la valeur et le champ ou la propriété auquel la valeur est assignée. Intuitivement, un initialiseur de projection projette non seulement une valeur, mais également le nom de la valeur.

Opérateur typeof

L' `typeof` opérateur est utilisé pour obtenir l' `System.Type` objet pour un type.

antlr

```
typeof_expression
    : 'typeof' '(' type ')'
    | 'typeof' '(' unbound_type_name ')'
    | 'typeof' '(' 'void' ')'
    ;

unbound_type_name
    : identifier generic_dimension_specifier?
    | identifier '::' identifier generic_dimension_specifier?
    | unbound_type_name '.' identifier generic_dimension_specifier?
    ;

generic_dimension_specifier
    : '<' comma* '>'
    ;

comma
    : ','
    ;
```

La première forme de *typeof_expression* se compose d'un `typeof` mot clé suivi d'un *type* entre parenthèses. Le résultat d'une expression de ce formulaire est l' `System.Type` objet du type indiqué. Il n'existe qu'un seul `System.Type` objet pour un type donné. Cela signifie que pour un type `T` , `typeof(T) == typeof(T)` a toujours la valeur true. Le *type* ne peut pas être `dynamic` .

La deuxième forme de *typeof_expression* se compose d'un `typeof` mot clé suivi d'un *unbound_type_name* entre parenthèses. Un *unbound_type_name* est très similaire à un *type_name* (nom de l'espace de noms et type), à ceci près qu'un *unbound_type_name* contient *generic_dimension_specifier* s où un *type_name* contient *type_argument_list* s. Lorsque l'opérande d'un *typeof_expression* est une séquence de jetons qui satisfait les grammaires de *unbound_type_name* et *type_name*, à savoir lorsqu'il ne contient ni un *generic_dimension_specifier* ni un *type_argument_list*, la séquence de jetons est considérée comme un *type_name*. La signification d'un *unbound_type_name* est déterminée comme suit :

- Convertissez la séquence de jetons en *type_name* en remplaçant chaque *generic_dimension_specifier* par un *type_argument_list* ayant le même nombre de virgules et le mot clé `object` comme chaque *type_argument*.
- Évaluez le *type_name* résultant, en ignorant toutes les contraintes de paramètre de type.
- Le *unbound_type_name* est résolu en type générique indépendant associé au type construit résultant (types liés et indépendants).

Le résultat de l' *typeof_expression* est l' `System.Type` objet pour le type générique indépendant qui en résulte.

La troisième forme de *typeof_expression* se compose d'un `typeof` mot clé suivi d'un `void` mot clé entre parenthèses. Le résultat d'une expression de ce formulaire est l' `System.Type` objet qui représente l'absence d'un type. L'objet de type retourné par `typeof(void)` est différent de l'objet de type retourné pour tout type. Cet objet de type spécial est utile dans les bibliothèques de classes qui autorisent la réflexion sur les méthodes dans le langage, où ces méthodes souhaitent avoir un moyen de représenter le type de retour d'une méthode quelconque, y compris les méthodes void, avec une instance de `System.Type` .

L' `typeof` opérateur peut être utilisé sur un paramètre de type. Le résultat est l' `System.Type` objet pour le type au moment de l'exécution qui a été lié au paramètre de type. L' `typeof` opérateur peut également être utilisé sur un type construit ou un type générique indépendant (types liés et indépendants). L' `System.Type` objet pour un type générique indépendant n'est pas le même que l' `System.Type` objet du type d'instance. Le type d'instance est toujours un type construit fermé au moment de l'exécution, de sorte que son `System.Type` objet dépend des arguments de type au moment de l'exécution en cours d'utilisation, tandis que le type générique indépendant n'a aucun argument de type.

L'exemple

C#

```
using System;

class X<T>
{
    public static void PrintTypes() {
        Type[] t = {
            typeof(int),
            typeof(System.Int32),
            typeof(string),
            typeof(double[]),
            typeof(void),
            typeof(T),
            typeof(X<T>),
            typeof(X<X<T>>),
            typeof(X<>)
        };
        for (int i = 0; i < t.Length; i++) {
            Console.WriteLine(t[i]);
        }
    }
}
```

```
class Test
{
    static void Main() {
        X<int>.PrintTypes();
    }
}
```

produit la sortie suivante :

Console

```
System.Int32
System.Int32
System.String
System.Double[]
System.Void
System.Int32
X`1[System.Int32]
X`1[X`1[System.Int32]]
X`1[T]
```

Notez que `int` et `System.Int32` sont du même type.

Notez également que le résultat de `typeof(X<>)` ne dépend pas de l'argument de type, mais du résultat de `typeof(X<T>)` .

Opérateurs vérifiés et non vérifiés

Les `checked` `unchecked` opérateurs et sont utilisés pour contrôler le ***contexte*** de contrôle de dépassement de capacité pour les opérations arithmétiques de type intégral et les conversions.

antlr

```
checked_expression
    : 'checked' '(' expression ')'
    ;

unchecked_expression
    : 'unchecked' '(' expression ')'
    ;
```

L' `checked` opérateur évalue l'expression contenue dans un contexte vérifié, et l' `unchecked` opérateur évalue l'expression contenue dans un contexte non vérifié. Un *checked_expression* ou *unchecked_expression* correspond exactement à un

parenthesized_expression (expressions entre parenthèses), sauf que l'expression contenue est évaluée dans le contexte de contrôle de dépassement de capacité donné.

Le contexte de contrôle de dépassement de capacité peut également être contrôlé via les `checked` `unchecked` instructions et (les instructions checked et unchecked).

Les opérations suivantes sont affectées par le contexte de contrôle de dépassement de capacité établi par les `checked` `unchecked` opérateurs et et les instructions :

- Les opérateurs prédéfinis `++` et `--` unaires (suffixes d'incrémentation et de décrémentation et opérateurs d'incrémentation et de décrémentation de préfixe), lorsque l'opérande est de type intégral.
- Opérateur unaire prédéfini `-` (opérateur moins unaire), lorsque l'opérande est de type intégral.
- Les opérateurs binaires prédéfinis, `+` `-` , `*` et `/` (opérateurs arithmétiques), lorsque les deux opérandes sont des types intégraux.
- Conversions numériques explicites (conversions numériques explicites) d'un type intégral vers un autre type intégral, ou à partir de `float` ou `double` en un type intégral.

Lorsque l'une des opérations ci-dessus génère un résultat qui est trop grand pour être représenté dans le type de destination, le contexte dans lequel l'opération est effectuée contrôle le comportement résultant :

- Dans un `checked` contexte, si l'opération est une expression constante (expressions constantes), une erreur de compilation se produit. Dans le cas contraire, lorsque l'opération est effectuée au moment de l'exécution, une `System.OverflowException` exception est levée.
- Dans un `unchecked` contexte, le résultat est tronqué en ignorant les bits de poids fort qui ne tiennent pas dans le type de destination.

Pour les expressions non constantes (expressions évaluées au moment de l'exécution) qui ne sont pas incluses dans les `checked` `unchecked` opérateurs ou ou les instructions, le contexte de contrôle de dépassement de capacité par défaut est, `unchecked` sauf si des facteurs externes (tels que les commutateurs de compilateur et la configuration de l'environnement d'exécution) appellent pour l' `checked` évaluation.

Pour les expressions constantes (expressions qui peuvent être entièrement évaluées au moment de la compilation), le contexte de contrôle de dépassement de capacité par défaut est toujours `checked` . À moins qu'une expression constante ne soit explicitement placée dans un `unchecked` contexte, les débordements qui se produisent pendant

l'évaluation de l'expression au moment de la compilation provoquent toujours des erreurs au moment de la compilation.

Le corps d'une fonction anonyme n'est pas affecté par `checked` ou `unchecked` les contextes dans lesquels la fonction anonyme se produit.

Dans l'exemple

C#

```
class Test
{
    static readonly int x = 1000000;
    static readonly int y = 1000000;

    static int F() {
        return checked(x * y);      // Throws OverflowException
    }

    static int G() {
        return unchecked(x * y);    // Returns -727379968
    }

    static int H() {
        return x * y;               // Depends on default
    }
}
```

aucune erreur de compilation n'est signalée, car aucune des expressions ne peut être évaluée au moment de la compilation. Au moment de l'exécution, la `F` méthode lève une `System.OverflowException` , et la `G` méthode retourne-727379968 (les 32 de poids faible du résultat hors limites). Le comportement de la `H` méthode dépend du contexte de vérification de dépassement de capacité par défaut pour la compilation, mais elle est égale à `F` ou identique à `G` .

Dans l'exemple

C#

```
class Test
{
    const int x = 1000000;
    const int y = 1000000;

    static int F() {
        return checked(x * y);      // Compile error, overflow
    }

    static int G() {
```

```
        return unchecked(x * y);    // Returns -727379968
    }

    static int H() {
        return x * y;               // Compile error, overflow
    }
}
```

les débordements qui se produisent lors de l'évaluation des expressions constantes dans `F` et `H` entraînent le signalement des erreurs de compilation, car les expressions sont évaluées dans un `checked` contexte. Un dépassement de capacité se produit également lors de l'évaluation de l'expression constante dans `G` , mais étant donné que l'évaluation a lieu dans un `unchecked` contexte, le dépassement de capacité n'est pas signalé.

Les `checked` `unchecked` opérateurs et affectent uniquement le contexte de contrôle de dépassement de capacité pour les opérations qui sont textuellement contenues dans les `(` jetons « » et « » `)` . Les opérateurs n'ont aucun effet sur les fonctions membres qui sont appelées suite à l'évaluation de l'expression contenue. Dans l'exemple

C#

```
class Test
{
    static int Multiply(int x, int y) {
        return x * y;
    }

    static int F() {
        return checked(Multiply(1000000, 1000000));
    }
}
```

l'utilisation de `checked` dans `F` n'affecte pas l'évaluation de `x * y` dans `Multiply` `x * y` . par conséquent, est évalué dans le contexte de contrôle de dépassement de capacité par défaut.

L' `unchecked` opérateur est pratique lors de l'écriture de constantes des types intégraux signés en notation hexadécimale. Par exemple :

C#

```
class Test
{
    public const int AllBits = unchecked((int)0xFFFFFFFF);

    public const int HighBit = unchecked((int)0x80000000);
}
```

Les deux constantes hexadécimales ci-dessus sont de type `uint` . Étant donné que les constantes sont en dehors de la `int` plage, sans l' `unchecked` opérateur, les casts `int` génèrent des erreurs au moment de la compilation.

Les `checked` `unchecked` opérateurs et et les instructions permettent aux programmeurs de contrôler certains aspects de certains calculs numériques. Toutefois, le comportement de certains opérateurs numériques dépend des types de données des opérandes. Par exemple, la multiplication de deux décimales produit toujours une exception en cas de dépassement de capacité, même dans une `unchecked` construction explicite. De même, la multiplication de deux valeurs float n'entraîne jamais d'exception en cas de dépassement de capacité, même dans une `checked` construction explicite. En outre, les autres opérateurs ne sont jamais affectés par le mode de vérification, qu'il s'agisse d'une valeur par défaut ou explicite.

Expressions de valeur par défaut

Une expression de valeur par défaut est utilisée pour obtenir la valeur par défaut (valeurs par défaut) d'un type. En général, une expression de valeur par défaut est utilisée pour les paramètres de type, car elle peut ne pas être connue si le paramètre de type est un type valeur ou un type référence. (Aucune conversion n'existe du `null` littéral à un paramètre de type, sauf si le paramètre de type est connu comme étant un type référence.)

antlr

```
default_value_expression
    : 'default' '(' type ')'
    ;
```

Si le *type* d'une *default_value_expression* évalue au moment de l'exécution sur un type référence, le résultat est `null` converti en ce type. Si le *type* d'une *default_value_expression* évalue au moment de l'exécution sur un type valeur, le résultat est la valeur par défaut de *Value_type*(constructeurs par défaut).

Une *default_value_expression* est une expression constante (expressions constantes) si le type est un type référence ou un paramètre de type qui est connu comme étant un type référence (contraintes de paramètre de type). En outre, une *default_value_expression* est une expression constante si le type est l'un des types de valeurs suivants : `sbyte` , `byte` ,,, `short` `ushort` `int` , `uint` , `long` , `ulong` , `char` , `float` , `double` , `decimal` , `bool` ou tout type d'énumération.

Expressions Nameof

Un *nameof_expression* est utilisé pour obtenir le nom d'une entité de programme sous la forme d'une chaîne constante.

antlr

```
nameof_expression
    : 'nameof' '(' named_entity ')'
    ;

named_entity
    : simple_name
    | named_entity_target '.' identifier type_argument_list?
    ;

named_entity_target
    : 'this'
    | 'base'
    | named_entity
    | predefined_type
    | qualified_alias_member
    ;
```

Par programmation, l'opérande *named_entity* est toujours une expression. Étant donné que `nameof` n'est pas un mot clé réservé, une expression nameof est toujours ambiguë syntaxiquement avec un appel du nom simple `nameof` . Pour des raisons de compatibilité, si une recherche de nom (noms simples) du nom `nameof` aboutit, l'expression est traitée comme un *invocation_expression* , que l'appel soit légal ou non. Dans le cas contraire, il s'agit d'un *nameof_expression*.

La signification de la *named_entity* d'une *nameof_expression* est la signification de celle-ci en tant qu'expression ; autrement dit, qu'il s'agisse d'un *simple_name*, d'un *base_access* ou d'un *member_access*. Toutefois, lorsque la recherche décrite dans noms simples et accès aux membres génère une erreur parce qu'un membre d'instance a été trouvé dans un contexte statique, une *nameof_expression* ne produit aucune erreur de ce type.

Il s'agit d'une erreur de compilation pour un *named_entity* désignant un groupe de méthodes pour avoir une *type_argument_list*. Il s'agit d'une erreur au moment de la compilation pour qu'un *named_entity_target* ait le type `dynamic` .

Une *nameof_expression* est une expression constante de type `string` et n'a aucun effet au moment de l'exécution. En particulier, sa *named_entity* n'est pas évaluée et est ignorée dans le cadre de l'analyse de l'assignation définie (règles générales pour les

expressions simples). Sa valeur est le dernier identificateur de la *named_entity* avant le *type_argument_list* final facultatif, transformée de la façon suivante :

- Le préfixe « `@` », s'il est utilisé, est supprimé.
- Chaque *unicode_escape_sequence* est transformée en son caractère Unicode correspondant.
- Toutes les *formatting_characters* sont supprimées.

Ce sont les mêmes transformations appliquées dans les identificateurs lors du test d'égalité entre les identificateurs.

TODO : exemples

Expressions de méthode anonymes

Une *anonymous_method_expression* est l'une des deux façons de définir une fonction anonyme. Celles-ci sont décrites plus en détail dans expressions de fonction anonymes.

Opérateurs unaires

Les `?` `+` opérateurs,, `-` ,,,, `!` `~` `++` `--` , Cast et `await` sont appelés des opérateurs unaires.

antlr

```
unary_expression
    : primary_expression
    | null_conditional_expression
    | '+' unary_expression
    | '-' unary_expression
    | '!' unary_expression
    | '~' unary_expression
    | pre_increment_expression
    | pre_decrement_expression
    | cast_expression
    | await_expression
    | unary_expression_unsafe
    ;
```

Si l'opérande d'un *unary_expression* a le type au moment de la compilation `dynamic` , il est lié dynamiquement (liaison dynamique). Dans ce cas, le type au moment de la compilation de l' *unary_expression* est `dynamic` , et la résolution décrite ci-après aura lieu au moment de l'exécution à l'aide du type d'exécution de l'opérande.

Opérateur conditionnel Null

L'opérateur conditionnel null applique une liste d'opérations à son opérande uniquement si cet opérande n'a pas la valeur null. Sinon, le résultat de l'application de l'opérateur est `null` .

antlr

```
null_conditional_expression
    : primary_expression null_conditional_operations
    ;

null_conditional_operations
    : null_conditional_operations? '?' '.' identifier type_argument_list?
    | null_conditional_operations? '?' '[' argument_list ']'
    | null_conditional_operations '.' identifier type_argument_list?
    | null_conditional_operations '[' argument_list ']'
    | null_conditional_operations '(' argument_list? ')'
    ;
```

La liste des opérations peut inclure l'accès au membre et les opérations d'accès aux éléments (qui peuvent eux-mêmes être conditionnels null), ainsi que l'appel.

Par exemple, l'expression `a.b?[0]?.c()` est une *null_conditional_expression* avec un *primary_expression* `a.b` et *null_conditional_operations* `?[0]` (accès conditionnel null), `?.c` (accès au membre conditionnel null) et `()` (appel).

Pour une *null_conditional_expression* `E` avec un *primary_expression* `P` , indiquez `E0` l'expression obtenue en supprimant textuellement le début de `?` chaque *null_conditional_operations* de qui en `E` possède un. D'un point de vue conceptuel, `E0` est l'expression qui sera évaluée si aucune des vérifications null représentées par le `?` s ne trouve un `null` .

En outre, il `E1` s'agit de l'expression obtenue en supprimant textuellement le début du `?` premier du *null_conditional_operations* dans `E` . Cela peut aboutir à une *expression primaire* (s'il y en a une `?`) ou à une autre *null_conditional_expression*.

Par exemple, si `E` est l'expression `a.b?[0]?.c()` , `E0` est l'expression `a.b[0].c()` et `E1` est l'expression `a.b[0]?.c()` .

Si `E0` est classifié comme Nothing, `E` est classifié comme Nothing. Dans le cas contraire, E est classé comme une valeur.

`E0` et `E1` sont utilisés pour déterminer la signification de `E` :

- Si `E` se produit en tant que *statement_expression* la signification de `E` est identique à celle de l'instruction

  ```
  C#

  if ((object)P != null) E1;
  ```

  sauf que P n'est évalué qu'une seule fois.

- Sinon, si `E0` est classifié comme rien, une erreur de compilation se produit.

- Dans le cas contraire, indiquez `T0` le type de `E0` .

  - Si `T0` est un paramètre de type qui n'est pas connu comme étant un type référence ou un type valeur n'acceptant pas les valeurs NULL, une erreur de compilation se produit.

  - Si `T0` est un type valeur qui n'autorise pas les valeurs NULL, le type de `E` est `T0?` , et la signification de `E` est identique à

    ```
    C#

    ((object)P == null) ? (T0?)null : E1
    ```

    sauf que `P` n'est évalué qu'une seule fois.

  - Dans le cas contraire, le type de E est T0, et la signification de E est la même que

    ```
    C#

    ((object)P == null) ? null : E1
    ```

    sauf que `P` n'est évalué qu'une seule fois.

Si `E1` est lui-même un *null_conditional_expression*, ces règles sont appliquées de nouveau, en imbriquant les tests `null` jusqu'à ce qu'il n'y ait plus `?` de valeur, et l'expression a été réduite jusqu'à l'expression primaire `E0` .

Par exemple, si l'expression `a.b?[0]?.c()` se présente sous la forme d'une instruction-expression, comme dans l'instruction :

```
C#

a.b?[0]?.c();
```

son sens est équivalent à :

```C#
if (a.b != null) a.b[0]?.c();
```

ce qui équivaut à :

```C#
if (a.b != null) if (a.b[0] != null) a.b[0].c();
```

Sauf que `a.b` et `a.b[0]` sont évalués une seule fois.

Si elle se produit dans un contexte où sa valeur est utilisée, comme dans :

```C#
var x = a.b?[0]?.c();
```

en partant du principe que le type de l'appel final n'est pas un type valeur non Nullable, son sens est équivalent à :

```C#
var x = (a.b == null) ? null : (a.b[0] == null) ? null : a.b[0].c();
```

sauf que `a.b` et `a.b[0]` sont évalués une seule fois.

Expressions conditionnelles null en tant qu'initialiseurs de projection

Une expression conditionnelle null est uniquement autorisée en tant que *member_declarator* dans un *anonymous_object_creation_expression* (expressions de création d'objet anonyme) si elle se termine par un accès au membre (éventuellement nullement conditionnel). Par programmation, cette exigence peut être exprimée comme suit :

```antlr
null_conditional_member_access
    : primary_expression null_conditional_operations? '?' '.' identifier
type_argument_list?
```

```
    | primary_expression null_conditional_operations '.' identifier
type_argument_list?
    ;
```

Il s'agit d'un cas particulier de la grammaire pour *null_conditional_expression* ci-dessus. La production pour *member_declarator* dans les expressions de création d'objet anonyme comprend alors uniquement *null_conditional_member_access*.

Expressions conditionnelles null en tant qu'expressions d'instruction

Une expression conditionnelle null est uniquement autorisée en tant que *statement_expression* (instructions d'expression) si elle se termine par un appel. Par programmation, cette exigence peut être exprimée comme suit :

antlr

```
null_conditional_invocation_expression
    : primary_expression null_conditional_operations '(' argument_list? ')'
    ;
```

Il s'agit d'un cas particulier de la grammaire pour *null_conditional_expression* ci-dessus. La production de *statement_expression* dans les instructions d'expression comprend alors uniquement *null_conditional_invocation_expression*.

Opérateur plus unaire

Pour une opération de ce type `+x` , la résolution de surcharge d'opérateur unaire (résolution de surcharge d'opérateur unaire) est appliquée pour sélectionner une implémentation d'opérateur spécifique. L'opérande est converti en type de paramètre de l'opérateur sélectionné et le type du résultat est le type de retour de l'opérateur. Les opérateurs plus unaires prédéfinis sont les suivants :

C#

```
int operator +(int x);
uint operator +(uint x);
long operator +(long x);
ulong operator +(ulong x);
float operator +(float x);
double operator +(double x);
decimal operator +(decimal x);
```

Pour chacun de ces opérateurs, le résultat est simplement la valeur de l'opérande.

Opérateur moins unaire

Pour une opération de ce type `-x` , la résolution de surcharge d'opérateur unaire (résolution de surcharge d'opérateur unaire) est appliquée pour sélectionner une implémentation d'opérateur spécifique. L'opérande est converti en type de paramètre de l'opérateur sélectionné et le type du résultat est le type de retour de l'opérateur. Les opérateurs de négation prédéfinis sont les suivants :

- Négation d'entier :

  C#

  ```
  int operator -(int x);
  long operator -(long x);
  ```

  Le résultat est calculé en soustrayant `x` de zéro. Si la valeur de `x` est la plus petite valeur représentable du type d'opérande (-2 ^ 31 pour `int` ou-2 ^ 63 pour `long`), la négation mathématique de `x` n'est pas représentable dans le type d'opérande. Si cela se produit dans un `checked` contexte, une `System.OverflowException` est levée ; si elle se produit dans un `unchecked` contexte, le résultat est la valeur de l'opérande et le dépassement de capacité n'est pas signalé.

  Si l'opérande de l'opérateur de négation est de type `uint` , il est converti en type `long` , et le type du résultat est `long` . Une exception est la règle qui autorise l' `int` écriture de la valeur-2147483648 (-2 ^ 31) sous la forme d'un littéral d'entier décimal (littéraux entiers).

  Si l'opérande de l'opérateur de négation est de type `ulong` , une erreur de compilation se produit. Une exception est la règle qui autorise l' `long` écriture de la valeur-9223372036854775808 (-2 ^ 63) sous la forme d'un littéral d'entier décimal (littéraux entiers).

- Négation à virgule flottante :

  C#

  ```
  float operator -(float x);
  double operator -(double x);
  ```

  Le résultat est la valeur de `x` avec son signe inversé. Si `x` est NaN, le résultat est également Nan.

- Négation décimale :

```
C#

decimal operator -(decimal x);
```

Le résultat est calculé en soustrayant `x` de zéro. La négation décimale équivaut à utiliser l'opérateur moins unaire de type `System.Decimal` .

Opérateur de négation logique

Pour une opération de ce type `!x` , la résolution de surcharge d'opérateur unaire (résolution de surcharge d'opérateur unaire) est appliquée pour sélectionner une implémentation d'opérateur spécifique. L'opérande est converti en type de paramètre de l'opérateur sélectionné et le type du résultat est le type de retour de l'opérateur. Il n'existe qu'un seul opérateur de négation logique prédéfini :

```
C#

bool operator !(bool x);
```

Cet opérateur calcule la négation logique de l'opérande : si l'opérande est `true` , le résultat est `false` . Si l'opérande est `false`, le résultat est `true`.

Opérateur de complément de bits

Pour une opération de ce type `~x` , la résolution de surcharge d'opérateur unaire (résolution de surcharge d'opérateur unaire) est appliquée pour sélectionner une implémentation d'opérateur spécifique. L'opérande est converti en type de paramètre de l'opérateur sélectionné et le type du résultat est le type de retour de l'opérateur. Les opérateurs de complément de bits prédéfinis sont les suivants :

```
C#

int operator ~(int x);
uint operator ~(uint x);
long operator ~(long x);
ulong operator ~(ulong x);
```

Pour chacun de ces opérateurs, le résultat de l'opération est le complément de bits de `x` .

Chaque type d'énumération `E` fournit implicitement l'opérateur de complément de bits suivant :

C#

```
E operator ~(E x);
```

Le résultat de l'évaluation de `~x` , où `x` est une expression d'un type d'énumération `E` avec un type sous-jacent `U` , est exactement identique à l'évaluation de `(E)(~(U)x)` , sauf que la conversion en `E` est toujours effectuée comme si dans un `unchecked` contexte (opérateurs activés et désactivés).

Opérateurs préfixés d'incrémentation et de décrémentation

antlr

```
pre_increment_expression
    : '++' unary_expression
    ;

pre_decrement_expression
    : '--' unary_expression
    ;
```

L'opérande d'une opération d'incrémentation ou de décrémentation de préfixe doit être une expression classifiée comme une variable, un accès à une propriété ou un accès à un indexeur. Le résultat de l'opération est une valeur du même type que l'opérande.

Si l'opérande d'une opération d'incrémentation ou de décrémentation de préfixe est un accès à une propriété ou un indexeur, la propriété ou l'indexeur doit avoir à la fois un `get` `set` accesseur et un accesseur. Si ce n'est pas le cas, une erreur de temps de liaison se produit.

La résolution de surcharge d'opérateur unaire (résolution de surcharge d'opérateur unaire) est appliquée pour sélectionner une implémentation d'opérateur spécifique. `++`Des opérateurs et prédéfinis `--` existent pour les types suivants : `sbyte` , `byte` , `short` ,,, `ushort` `int` `uint` , `long` , `ulong` , `char` , `float` , `double` , `decimal` et tout type enum. Les opérateurs prédéfinis `++` retournent la valeur produite par l'ajout de 1 à l'opérande, et les opérateurs prédéfinis `--` retournent la valeur produite par la soustraction de 1 de l'opérande. Dans un `checked` contexte, si le résultat de l'addition ou de la soustraction est en dehors de la plage du type de résultat et que le type de résultat est un type intégral ou un type enum, une `System.OverflowException` est levée.

Le traitement au moment de l'exécution d'une opération de préfixe d'incrémentation ou de décrémentation du formulaire `++x` ou `--x` se compose des étapes suivantes :

- Si `x` est classé en tant que variable :
  - `x` est évalué pour produire la variable.
  - L'opérateur sélectionné est appelé avec la valeur de `x` comme argument.
  - La valeur retournée par l'opérateur est stockée dans l'emplacement donné par l'évaluation de `x` .
  - La valeur retournée par l'opérateur devient le résultat de l'opération.
- Si `x` est classifié comme un accès à une propriété ou un indexeur :
  - L'expression d'instance (si `x` n'est pas `static`) et la liste d'arguments (si `x` est un accès à un indexeur) associé à `x` sont évaluées, et les résultats sont utilisés dans les `get` appels suivants et d' `set` accesseur.
  - L' `get` accesseur de `x` est appelé.
  - L'opérateur sélectionné est appelé avec la valeur retournée par l' `get` accesseur comme argument.
  - L' `set` accesseur de `x` est appelé avec la valeur retournée par l'opérateur comme son `value` argument.
  - La valeur retournée par l'opérateur devient le résultat de l'opération.

Les `++` `--` opérateurs et prennent également en charge la notation suffixée (opérateurs d'incrémentation et de décrémentation suffixés). En général, le résultat de `x++` ou `x--` est la valeur de `x` avant l'opération, alors que le résultat de `++x` ou `--x` est la valeur de `x` après l'opération. Dans les deux cas, `x` elle a la même valeur après l'opération.

Une `operator++` `operator--` implémentation ou peut être appelée à l'aide de la notation suffixée ou de préfixe. Il n'est pas possible d'avoir des implémentations d'opérateur distinctes pour les deux notations.

Expressions cast

Un *cast_expression* est utilisé pour convertir explicitement une expression en un type donné.

antlr

```
cast_expression
    : '(' type ')' unary_expression
    ;
```

Cast_expression de la forme `(T)E` , où `T` est un *type* et `E` est un *unary_expression*, effectue une conversion explicite (conversions explicites) de la valeur de `E` en type `T` . S'il n'existe aucune conversion explicite de `E` à `T` , une erreur de liaison au moment de la liaison se produit. Dans le cas contraire, le résultat est la valeur produite par la conversion explicite. Le résultat est toujours classifié comme une valeur, même si `E` indique une variable.

La grammaire d'un *cast_expression* provoque certaines ambiguïtés syntaxiques. Par exemple, l'expression `(x)-y` peut être interprétée comme une *cast_expression* (un cast de `-y` en type `x`) ou comme une *additive_expression* associée à un *parenthesized_expression* (qui calcule la valeur `x - y)` .

Pour résoudre *cast_expression* ambiguïtés, la règle suivante existe : une séquence d'un ou plusieurs *jetons*(espace blanc) entre parenthèses est considérée comme le début d'un *cast_expression* uniquement si au moins l'une des conditions suivantes est vraie :

- La séquence de jetons est une grammaire correcte pour un *type*, mais pas pour une *expression*.
- La séquence de jetons est une grammaire correcte pour un *type*, et le jeton qui suit immédiatement les parenthèses fermantes est le jeton « », le jeton « », `~` `!` le jeton « `(` », un *identificateur* (séquences d'échappement de caractères Unicode), un *littéral* (littéraux) ou tout *mot clé* (Mots clés) sauf `as` et `is` .

Le terme « grammaire correcte » ci-dessus signifie uniquement que la séquence de jetons doit se conformer à la production grammaticale particulière. Elle ne tient pas compte de la signification réelle des identificateurs constitutifs. Par exemple, si `x` et `y` sont des identificateurs, `x.y` est une grammaire correcte pour un type, même si `x.y` ne désigne pas un type.

À partir de la règle de désambiguïté, elle suit que, si `x` et `y` sont des identificateurs, `(x)y` , `(x)(y)` et `(x)(-y)` sont *cast_expression* s, mais `(x)-y` n'est pas, même si `x` identifie un type. Toutefois, si `x` est un mot clé qui identifie un type prédéfini (tel que `int`), les quatre formes sont *cast_expression* s (car ce mot clé n'est peut-être pas une expression par lui-même).

Expressions await

L'opérateur await est utilisé pour interrompre l'évaluation de la fonction Async englobante jusqu'à ce que l'opération asynchrone représentée par l'opérande soit terminée.

```
```

antlr

```
await_expression
    : 'await' unary_expression
    ;
```

Une *await_expression* est uniquement autorisée dans le corps d'une fonction Async (fonctions Async). Dans la fonction Async englobante la plus proche, un *await_expression* peut ne pas se produire dans les emplacements suivants :

- Dans une fonction anonyme imbriquée (non asynchrone)
- À l'intérieur du bloc d'un *lock_statement*
- Dans un contexte non sécurisé

Notez qu'un *await_expression* ne peut pas se produire dans la plupart des cas au sein d'un *query_expression*, car ceux-ci sont convertis de façon syntaxique pour utiliser des expressions lambda non asynchrones.

À l'intérieur d'une fonction Async, `await` ne peut pas être utilisé en tant qu'identificateur. Il n'y a donc pas d'ambiguïté syntaxique entre await-expressions et diverses expressions impliquant des identificateurs. En dehors des fonctions Async, `await` agit comme un identificateur normal.

L'opérande d'un *await_expression* est appelé la **tâche***. Il représente une opération asynchrone qui peut ou non se terminer au moment où le _await_expression * est évalué. L'objectif de l'opérateur await est de suspendre l'exécution de la fonction Async englobante jusqu'à ce que la tâche attendue soit terminée, puis d'obtenir son résultat.

Expressions await

La tâche d'une expression await doit être ***attendue***. Une expression `t` est await si l'un des éléments suivants est attendu :

- `t` est de type au moment de la compilation `dynamic`
- `t` a une instance accessible ou une méthode d'extension appelée sans `GetAwaiter` paramètres et sans paramètres de type, et un type `A` de retour pour lequel tous les éléments suivants sont détenus :
  - `A` implémente l'interface `System.Runtime.CompilerServices.INotifyCompletion` (appelée par souci `INotifyCompletion` de concision)
  - `A` a une propriété d'instance accessible en lecture `IsCompleted` de type `bool`
  - `A` a une méthode d'instance accessible sans `GetResult` paramètres et sans paramètres de type

L'objectif de la `GetAwaiter` méthode est d'obtenir un ***await** _ pour la tâche. Le type `A` est appelé _ *await type** pour l'expression await.

L'objectif de la `IsCompleted` propriété est de déterminer si la tâche est déjà terminée. Dans ce cas, il n'est pas nécessaire d'interrompre l'évaluation.

L'objectif de la `INotifyCompletion.OnCompleted` méthode consiste à inscrire une « continuation » à la tâche, c'est-à-dire un délégué (de type `System.Action`) qui sera appelé une fois la tâche terminée.

L'objectif de la `GetResult` méthode est d'obtenir le résultat de la tâche une fois qu'elle est terminée. Ce résultat peut être terminé avec succès, éventuellement avec une valeur de résultat, ou il peut s'agir d'une exception levée par la `GetResult` méthode.

Classification des expressions await

L'expression `await t` est classée de la même façon que l'expression `(t).GetAwaiter().GetResult()` . Ainsi, si le type de retour de `GetResult` est `void` , le *await_expression* est classé comme Nothing. S'il a un type de retour non void `T` , le *await_expression* est classé comme une valeur de type `T` .

Évaluation du runtime des expressions await

Au moment de l'exécution, l'expression `await t` est évaluée comme suit :

- Un await `a` est obtenu en évaluant l'expression `(t).GetAwaiter()` .
- Un `bool` `b` est obtenu en évaluant l'expression `(a).IsCompleted` .
- Si `b` la valeur est `false` , l'évaluation dépend de si `a` implémente l'interface `System.Runtime.CompilerServices.ICriticalNotifyCompletion` (appelée ci-après `ICriticalNotifyCompletion` par souci de concision). Cette vérification s'effectue au moment de la liaison. par exemple, lors de `a` l'exécution si a le type au moment de la compilation `dynamic` et au moment de la compilation, sinon. `r`Désignez le délégué de reprise (fonctions Async) :
  - Si `a` n'implémente pas `ICriticalNotifyCompletion` , l'expression `(a as (INotifyCompletion)).OnCompleted(r)` est évaluée.
  - Si `a` implémente `ICriticalNotifyCompletion` , l'expression `(a as (ICriticalNotifyCompletion)).UnsafeOnCompleted(r)` est évaluée.
  - L'évaluation est ensuite interrompue et le contrôle est retourné à l'appelant actuel de la fonction Async.

- Soit immédiatement après (si `b` was `true`), soit lors de l'appel ultérieur du délégué de reprise (si `b` was `false`), l'expression `(a).GetResult()` est évaluée. Si elle retourne une valeur, cette valeur est le résultat de l' *await_expression*. Sinon, le résultat est Nothing.

Une implémentation d'await des méthodes d'interface `INotifyCompletion.OnCompleted` et `ICriticalNotifyCompletion.UnsafeOnCompleted` doit entraîner l'appel du délégué `r` au plus une fois. Dans le cas contraire, le comportement de la fonction Async englobante n'est pas défini.

Opérateurs arithmétiques

Les `*` `/` opérateurs,,, `%` `+` et `-` sont appelés opérateurs arithmétiques.

antlr

```
multiplicative_expression
    : unary_expression
    | multiplicative_expression '*' unary_expression
    | multiplicative_expression '/' unary_expression
    | multiplicative_expression '%' unary_expression
    ;

additive_expression
    : multiplicative_expression
    | additive_expression '+' multiplicative_expression
    | additive_expression '-' multiplicative_expression
    ;
```

Si un opérande d'un opérateur arithmétique a le type au moment de la compilation `dynamic` , l'expression est liée dynamiquement (liaison dynamique). Dans ce cas, le type au moment de la compilation de l'expression est `dynamic` , et la résolution décrite ci-dessous se produit au moment de l'exécution à l'aide du type d'exécution des opérandes qui ont le type au moment de la compilation `dynamic` .

Opérateur de multiplication

Pour une opération de la forme `x * y` , la résolution de surcharge d'opérateur binaire (résolution de surcharge d'opérateur binaire) est appliquée pour sélectionner une implémentation d'opérateur spécifique. Les opérandes sont convertis en types de paramètre de l'opérateur sélectionné, et le type du résultat est le type de retour de l'opérateur.

Les opérateurs de multiplication prédéfinis sont répertoriés ci-dessous. Les opérateurs calculent tous le produit de `x` et `y` .

- Multiplication d'entiers :

```
C#

int operator *(int x, int y);
uint operator *(uint x, uint y);
long operator *(long x, long y);
ulong operator *(ulong x, ulong y);
```

Dans un `checked` contexte, si le produit est en dehors de la plage du type de résultat, une `System.OverflowException` est levée. Dans un `unchecked` contexte, les dépassements de capacité ne sont pas signalés et les bits de poids fort significatifs en dehors de la plage du type de résultat sont ignorés.

- Multiplication à virgule flottante :

```
C#

float operator *(float x, float y);
double operator *(double x, double y);
```

Le produit est calculé en fonction des règles de l'arithmétique IEEE 754. Le tableau suivant liste les résultats de toutes les combinaisons possibles de valeurs finies non nulles, de zéros, de valeurs infinies et de valeurs NaN. Dans le tableau, `x` et `y` sont des valeurs finies positives. `z` est le résultat de `x * y`. Si le résultat est trop grand pour le type de destination, `z` est une valeur infinie. Si le résultat est trop petit pour le type de destination, `z` est égal à zéro.

	+y	-y	+0	-0	+inf	-inf	NaN
+x	+z	-Z	+0	-0	+inf	-inf	NaN
-X	-Z	+z	-0	+0	-inf	+inf	NaN
+0	+0	-0	+0	-0	NaN	NaN	NaN
-0	-0	+0	-0	+0	NaN	NaN	NaN
+inf	+inf	-inf	NaN	NaN	+inf	-inf	NaN
-inf	-inf	+inf	NaN	NaN	-inf	+inf	NaN

NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Multiplication décimale :

```
C#

decimal operator *(decimal x, decimal y);
```

Si la valeur résultante est trop grande pour être représentée au `decimal` format, une `System.OverflowException` est levée. Si la valeur de résultat est trop petite pour être représentée au `decimal` format, le résultat est égal à zéro. L'échelle du résultat, avant tout arrondi, est la somme des échelles des deux opérandes.

La multiplication décimale équivaut à utiliser l'opérateur de multiplication de type `System.Decimal` .

Opérateur de division

Pour une opération de la forme `x / y` , la résolution de surcharge d'opérateur binaire (résolution de surcharge d'opérateur binaire) est appliquée pour sélectionner une implémentation d'opérateur spécifique. Les opérandes sont convertis en types de paramètre de l'opérateur sélectionné, et le type du résultat est le type de retour de l'opérateur.

Les opérateurs de division prédéfinis sont répertoriés ci-dessous. Les opérateurs calculent tous le quotient de `x` et `y` .

- Division d'entier :

```
C#

int operator /(int x, int y);
uint operator /(uint x, uint y);
long operator /(long x, long y);
ulong operator /(ulong x, ulong y);
```

Si la valeur de l'opérande de droite est égale à zéro, une `System.DivideByZeroException` exception est levée.

La Division arrondit le résultat vers zéro. Par conséquent, la valeur absolue du résultat est le plus grand entier possible qui est inférieur ou égal à la valeur absolue du quotient des deux opérandes. Le résultat est zéro ou positif lorsque les

deux opérandes ont le même signe et zéro ou une valeur négative lorsque les deux opérandes ont des signes opposés.

Si l'opérande de gauche est le plus petit représentable ou la plus petite `int` `long` valeur et que l'opérande de droite est `-1` , un dépassement de capacité se produit. Dans un `checked` contexte, cela entraîne la `System.ArithmeticException` levée d'un (ou d'une sous-classe). Dans un `unchecked` contexte, elle est définie par l'implémentation, à savoir si une `System.ArithmeticException` (ou une sous-classe) est levée ou si le dépassement de capacité n'est pas signalé avec la valeur résultante qui est celle de l'opérande de gauche.

- Division à virgule flottante :

```
C#

float operator /(float x, float y);
double operator /(double x, double y);
```

Le quotient est calculé en fonction des règles de l'arithmétique IEEE 754. Le tableau suivant liste les résultats de toutes les combinaisons possibles de valeurs finies non nulles, de zéros, de valeurs infinies et de valeurs NaN. Dans le tableau, `x` et `y` sont des valeurs finies positives. `z` est le résultat de `x / y`. Si le résultat est trop grand pour le type de destination, `z` est une valeur infinie. Si le résultat est trop petit pour le type de destination, `z` est égal à zéro.

	+y	-y	+0	-0	+inf	-inf	NaN
+x	+z	-Z	+inf	-inf	+0	-0	NaN
-X	-Z	+z	-inf	+inf	-0	+0	NaN
+0	+0	-0	NaN	NaN	+0	-0	NaN
-0	-0	+0	NaN	NaN	-0	+0	NaN
+inf	+inf	-inf	+inf	-inf	NaN	NaN	NaN
-inf	-inf	+inf	-inf	+inf	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Division décimale :

```
C#
```

```
decimal operator /(decimal x, decimal y);
```

Si la valeur de l'opérande de droite est égale à zéro, une `System.DivideByZeroException` exception est levée. Si la valeur résultante est trop grande pour être représentée au `decimal` format, une `System.OverflowException` est levée. Si la valeur de résultat est trop petite pour être représentée au `decimal` format, le résultat est égal à zéro. L'échelle du résultat est la plus petite échelle qui conserve un résultat égal à la valeur décimale représentable la plus proche du résultat mathématique vrai.

La Division décimale équivaut à utiliser l'opérateur de division de type `System.Decimal` .

Opérateur de reste

Pour une opération de la forme `x % y` , la résolution de surcharge d'opérateur binaire (résolution de surcharge d'opérateur binaire) est appliquée pour sélectionner une implémentation d'opérateur spécifique. Les opérandes sont convertis en types de paramètre de l'opérateur sélectionné, et le type du résultat est le type de retour de l'opérateur.

Les opérateurs de reste prédéfinis sont répertoriés ci-dessous. Les opérateurs calculent tous le reste de la division entre `x` et `y` .

- Reste entier :

```
C#
```

```
int operator %(int x, int y);
uint operator %(uint x, uint y);
long operator %(long x, long y);
ulong operator %(ulong x, ulong y);
```

Le résultat de `x % y` est la valeur produite par `x - (x / y) * y` . Si `y` est égal à zéro, une `System.DivideByZeroException` exception est levée.

Si l'opérande de gauche est la plus petite `int` ou la `long` valeur et que l'opérande de droite est `-1` , une `System.OverflowException` est levée. Dans le cas contraire, `x % y` lève une exception qui ne `x / y` lèvera pas d'exception.

- Reste à virgule flottante :

```
```

```
C#
```

```
float operator %(float x, float y);
double operator %(double x, double y);
```

Le tableau suivant liste les résultats de toutes les combinaisons possibles de valeurs finies non nulles, de zéros, de valeurs infinies et de valeurs NaN. Dans le tableau, `x` et `y` sont des valeurs finies positives. `z` est le résultat de `x % y` et est calculé en tant que `x - n * y` , où `n` est le plus grand entier possible qui est inférieur ou égal à `x / y` . Cette méthode de calcul du reste est analogue à celle utilisée pour les opérandes entiers, mais diffère de la définition IEEE 754 (dans laquelle `n` est l'entier le plus proche de `x / y`).

	+y	-y	+0	-0	+inf	-inf	NaN
+x	+z	+z	NaN	NaN	x	x	NaN
-X	-Z	-Z	NaN	NaN	-X	-X	NaN
+0	+0	+0	NaN	NaN	+0	+0	NaN
-0	-0	-0	NaN	NaN	-0	-0	NaN
+inf	NaN	NaN	NaN	NaN	NaN	NaN	NaN
-inf	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Reste décimal :

```
C#
```

```
decimal operator %(decimal x, decimal y);
```

Si la valeur de l'opérande de droite est égale à zéro, une `System.DivideByZeroException` exception est levée. L'échelle du résultat, avant tout arrondi, est la plus grande des échelles des deux opérandes, et le signe du résultat, s'il est différent de zéro, est le même que celui de `x` .

Le reste décimal est équivalent à l'utilisation de l'opérateur de reste de type `System.Decimal` .

Opérateur d'addition

Pour une opération de la forme `x + y` , la résolution de surcharge d'opérateur binaire (résolution de surcharge d'opérateur binaire) est appliquée pour sélectionner une implémentation d'opérateur spécifique. Les opérandes sont convertis en types de paramètre de l'opérateur sélectionné, et le type du résultat est le type de retour de l'opérateur.

Les opérateurs d'addition prédéfinis sont répertoriés ci-dessous. Pour les types numériques et d'énumération, les opérateurs d'addition prédéfinis calculent la somme des deux opérandes. Quand l'un des opérandes ou les deux sont de type chaîne, les opérateurs d'addition prédéfinis concatènent la représentation sous forme de chaîne des opérandes.

- Ajout d'un entier :

  C#

  ```
  int operator +(int x, int y);
  uint operator +(uint x, uint y);
  long operator +(long x, long y);
  ulong operator +(ulong x, ulong y);
  ```

  Dans un `checked` contexte, si la somme est en dehors de la plage du type de résultat, une `System.OverflowException` est levée. Dans un `unchecked` contexte, les dépassements de capacité ne sont pas signalés et les bits de poids fort significatifs en dehors de la plage du type de résultat sont ignorés.

- Addition à virgule flottante :

  C#

  ```
  float operator +(float x, float y);
  double operator +(double x, double y);
  ```

  La somme est calculée en fonction des règles de l'arithmétique IEEE 754. Le tableau suivant liste les résultats de toutes les combinaisons possibles de valeurs finies non nulles, de zéros, de valeurs infinies et de valeurs NaN. Dans le tableau, `x` et `y` sont des valeurs finies non nulles, et `z` est le résultat de `x + y`. Si `x` et `y` ont la même grandeur, mais des signes opposés, `z` est un zéro positif. Si `x + y` est trop grand pour être représenté dans le type de destination, `z` est un infini avec le même signe que `x + y` .

	y	+0	-0	+inf	-inf	NaN

x	z	x	x	+inf	-inf	NaN
+0	y	+0	+0	+inf	-inf	NaN
-0	y	+0	-0	+inf	-inf	NaN
+inf	+inf	+inf	+inf	+inf	NaN	NaN
-inf	-inf	-inf	-inf	NaN	-inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Addition de nombres décimaux :

```C#
decimal operator +(decimal x, decimal y);
```

  Si la valeur résultante est trop grande pour être représentée au `decimal` format, une `System.OverflowException` est levée. L'échelle du résultat, avant tout arrondi, est la plus grande des échelles des deux opérandes.

  L'addition décimale équivaut à utiliser l'opérateur d'addition de type `System.Decimal` .

- Addition d'énumération. Chaque type d'énumération fournit implicitement les opérateurs prédéfinis suivants, où `E` est le type enum, et `U` est le type sous-jacent de `E` :

```C#
E operator +(E x, U y);
E operator +(U x, E y);
```

  Au moment de l'exécution, ces opérateurs sont évalués exactement comme `(E)((U)x + (U)y)` .

- Concaténation de chaînes :

```C#
string operator +(string x, string y);
string operator +(string x, object y);
string operator +(object x, string y);
```

Ces surcharges de l' `+` opérateur binaire effectuent la concaténation de chaînes. Si un opérande de concaténation de chaîne est `null` , une chaîne vide est substituée. Sinon, tout argument qui n'est pas une chaîne est converti en sa représentation sous forme de chaîne en appelant la `ToString` méthode virtuelle héritée du type `object` . Si `ToString` retourne `null` , une chaîne vide est substituée.

```
C#

using System;

class Test
{
    static void Main() {
        string s = null;
        Console.WriteLine("s = >" + s + "<");        // displays s = ><
        int i = 1;
        Console.WriteLine("i = " + i);               // displays i = 1
        float f = 1.2300E+15F;
        Console.WriteLine("f = " + f);               // displays f =
1.23E+15
        decimal d = 2.900m;
        Console.WriteLine("d = " + d);               // displays d =
2.900
    }
}
```

Le résultat de l'opérateur de concaténation de chaînes est une chaîne qui se compose des caractères de l'opérande de gauche, suivis des caractères de l'opérande de droite. L'opérateur de concaténation de chaînes ne retourne jamais de `null` valeur. Une `System.OutOfMemoryException` exception peut être levée si la mémoire disponible est insuffisante pour allouer la chaîne résultante.

- Combinaison de délégués. Chaque type délégué fournit implicitement l'opérateur prédéfini suivant, où `D` est le type délégué :

```
C#

D operator +(D x, D y);
```

L' `+` opérateur binaire effectue une combinaison de délégués lorsque les deux opérandes sont d'un type délégué `D` . (Si les opérandes ont des types délégués différents, une erreur de temps de liaison se produit.) Si le premier opérande est `null` , le résultat de l'opération est la valeur du deuxième opérande (même si c'est également le cas `null`). Sinon, si le second opérande est `null` , le résultat de l'opération est la valeur du premier opérande. Dans le cas contraire, le résultat de

l'opération est une nouvelle instance de délégué qui, lorsqu'elle est appelée, appelle le premier opérande, puis appelle le second opérande. Pour obtenir des exemples de combinaison de délégués, consultez opérateur de soustraction et appel de délégué. Étant donné que `System.Delegate` n'est pas un type délégué, `operator +` n'est pas défini pour celui-ci.

Opérateur de soustraction

Pour une opération de la forme `x - y` , la résolution de surcharge d'opérateur binaire (résolution de surcharge d'opérateur binaire) est appliquée pour sélectionner une implémentation d'opérateur spécifique. Les opérandes sont convertis en types de paramètre de l'opérateur sélectionné, et le type du résultat est le type de retour de l'opérateur.

Les opérateurs de soustraction prédéfinis sont répertoriés ci-dessous. Les opérateurs soustraitnt tous `y` de `x` .

- Soustraction d'entiers :

  C#

  ```
  int operator -(int x, int y);
  uint operator -(uint x, uint y);
  long operator -(long x, long y);
  ulong operator -(ulong x, ulong y);
  ```

  Dans un `checked` contexte, si la différence se situe en dehors de la plage du type de résultat, une `System.OverflowException` est levée. Dans un `unchecked` contexte, les dépassements de capacité ne sont pas signalés et les bits de poids fort significatifs en dehors de la plage du type de résultat sont ignorés.

- Soustraction à virgule flottante :

  C#

  ```
  float operator -(float x, float y);
  double operator -(double x, double y);
  ```

  La différence est calculée en fonction des règles de l'arithmétique IEEE 754. Le tableau suivant répertorie les résultats de toutes les combinaisons possibles de valeurs finies différentes de zéro, de zéros, d'infinis et de valeurs NaN. Dans le tableau, `x` et `y` sont des valeurs finies non nulles, et `z` est le résultat de `x - y`. Si `x` et `y` sont égaux, `z` est un zéro positif. Si `x - y` est trop grand pour être

représenté dans le type de destination, `z` est un infini avec le même signe que `x - y` .

	y	+0	-0	+inf	-inf	NaN
x	z	x	x	-inf	+inf	NaN
+0	-y	+0	+0	-inf	+inf	NaN
-0	-y	-0	+0	-inf	+inf	NaN
+inf	+inf	+inf	+inf	NaN	+inf	NaN
-inf	-inf	-inf	-inf	-inf	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Soustraction de décimale :

```
C#

decimal operator -(decimal x, decimal y);
```

Si la valeur résultante est trop grande pour être représentée au `decimal` format, une `System.OverflowException` est levée. L'échelle du résultat, avant tout arrondi, est la plus grande des échelles des deux opérandes.

La soustraction de décimale équivaut à utiliser l'opérateur de soustraction de type `System.Decimal` .

- Soustraction d'énumération. Chaque type d'énumération fournit implicitement l'opérateur prédéfini suivant, où `E` est le type enum et `U` est le type sous-jacent de `E` :

```
C#

U operator -(E x, E y);
```

Cet opérateur est évalué exactement comme `(U)((U)x - (U)y)` . En d'autres termes, l'opérateur calcule la différence entre les valeurs ordinales de `x` et `y` , et le type du résultat est le type sous-jacent de l'énumération.

```
C#

E operator -(E x, U y);
```

Cet opérateur est évalué exactement comme `(E)((U)x - y)` . En d'autres termes, l'opérateur soustrait une valeur du type sous-jacent de l'énumération, ce qui produit une valeur de l'énumération.

- Suppression du délégué. Chaque type délégué fournit implicitement l'opérateur prédéfini suivant, où `D` est le type délégué :

```
C#

D operator -(D x, D y);
```

L' `-` opérateur binaire effectue une suppression de délégué lorsque les deux opérandes sont d'un type délégué `D` . Si les opérandes ont des types délégués différents, une erreur au moment de la liaison se produit. Si le premier opérande a la valeur `null`, le résultat de l'opération est `null`. Sinon, si le second opérande est `null` , le résultat de l'opération est la valeur du premier opérande. Sinon, les deux opérandes représentent les listes d'appel (déclarations de délégué) ayant une ou plusieurs entrées, et le résultat est une nouvelle liste d'appel comprenant la liste de la première opérande avec les entrées du deuxième opérande supprimées, à condition que la liste du deuxième opérande soit une sous-liste contiguë correcte du premier. (Pour déterminer l'égalité des sous-listes, les entrées correspondantes sont comparées à celles de l'opérateur d'égalité de délégué (opérateurs d'égalité de délégué).) Dans le cas contraire, le résultat est la valeur de l'opérande gauche. Aucune des listes d'opérandes n'est modifiée dans le processus. Si la liste du deuxième opérande correspond à plusieurs sous-listes d'entrées contiguës dans la liste du premier opérande, la sous-liste correspondante la plus à droite des entrées contiguës est supprimée. Si la suppression aboutit à une liste vide, le résultat est `null`. Par exemple :

```
C#

delegate void D(int x);

class C
{
    public static void M1(int i) { /* ... */ }
    public static void M2(int i) { /* ... */ }
}

class Test
{
    static void Main() {
        D cd1 = new D(C.M1);
```

```
        D cd2 = new D(C.M2);
        D cd3 = cd1 + cd2 + cd2 + cd1;    // M1 + M2 + M2 + M1
        cd3 -= cd1;                       // => M1 + M2 + M2

        cd3 = cd1 + cd2 + cd2 + cd1;      // M1 + M2 + M2 + M1
        cd3 -= cd1 + cd2;                 // => M2 + M1

        cd3 = cd1 + cd2 + cd2 + cd1;      // M1 + M2 + M2 + M1
        cd3 -= cd2 + cd2;                 // => M1 + M1

        cd3 = cd1 + cd2 + cd2 + cd1;      // M1 + M2 + M2 + M1
        cd3 -= cd2 + cd1;                 // => M1 + M2

        cd3 = cd1 + cd2 + cd2 + cd1;      // M1 + M2 + M2 + M1
        cd3 -= cd1 + cd1;                 // => M1 + M2 + M2 + M1
    }
}
```

Opérateurs de décalage

Les `<<` `>>` opérateurs et sont utilisés pour effectuer des opérations de décalage de bits.

antlr

```
shift_expression
    : additive_expression
    | shift_expression '<<' additive_expression
    | shift_expression right_shift additive_expression
    ;
```

Si un opérande d'un *shift_expression* a le type au moment de la compilation `dynamic` , l'expression est liée dynamiquement (liaison dynamique). Dans ce cas, le type au moment de la compilation de l'expression est `dynamic` , et la résolution décrite ci-dessous se produit au moment de l'exécution à l'aide du type d'exécution des opérandes qui ont le type au moment de la compilation `dynamic` .

Pour une opération de la forme `x << count` ou `x >> count` , la résolution de surcharge d'opérateur binaire (résolution de surcharge d'opérateur binaire) est appliquée pour sélectionner une implémentation d'opérateur spécifique. Les opérandes sont convertis en types de paramètre de l'opérateur sélectionné, et le type du résultat est le type de retour de l'opérateur.

Lors de la déclaration d'un opérateur de décalage surchargé, le type du premier opérande doit toujours être la classe ou le struct contenant la déclaration d'opérateur, et le type du second opérande doit toujours être `int` .

Les opérateurs de décalage prédéfinis sont répertoriés ci-dessous.

- Décalage vers la gauche :

  ```
  C#
  ```

  ```
  int operator <<(int x, int count);
  uint operator <<(uint x, int count);
  long operator <<(long x, int count);
  ulong operator <<(ulong x, int count);
  ```

  L' `<<` opérateur décale vers la `x` gauche d'un nombre de bits calculés comme décrit ci-dessous.

  Les bits de poids fort en dehors de la plage du type de résultat de `x` sont ignorés, les bits restants sont décalés vers la gauche et les positions de bit vides de poids faible sont définies sur zéro.

- Décalage vers la droite :

  ```
  C#
  ```

  ```
  int operator >>(int x, int count);
  uint operator >>(uint x, int count);
  long operator >>(long x, int count);
  ulong operator >>(ulong x, int count);
  ```

  L' `>>` opérateur décale vers la `x` droite d'un nombre de bits calculés comme décrit ci-dessous.

  Lorsque `x` est de type `int` ou `long` , les bits de poids faible de `x` sont ignorés, les bits restants sont décalés vers la droite et les positions de bits vides de poids fort ont la valeur zéro si `x` est non négative et défini sur un si `x` est négatif.

  Lorsque `x` est de type `uint` ou `ulong` , les bits de poids faible de `x` sont ignorés, les bits restants sont décalés vers la droite et les positions de bits vides de poids fort ont la valeur zéro.

Pour les opérateurs prédéfinis, le nombre de bits à décaler est calculé comme suit :

- Lorsque le type de `x` est `int` ou `uint` , le nombre de décalages est donné par les cinq bits de poids faible de `count` . En d'autres termes, le nombre de décalages est calculé à partir de `count & 0x1F` .
- Lorsque le type de `x` est `long` ou `ulong` , le nombre de décalages est donné par les six bits de poids faible de `count` . En d'autres termes, le nombre de décalages

est calculé à partir de `count & 0x3F` .

Si le nombre de décalages résultant est égal à zéro, les opérateurs de décalage retournent simplement la valeur de `x` .

Les opérations de décalage ne génèrent jamais de dépassement de capacité et produisent les mêmes résultats dans `checked` et `unchecked` contextes.

Lorsque l'opérande gauche de l' `>>` opérateur est d'un type intégral signé, l'opérateur effectue un décalage arithmétique vers la droite, où la valeur du bit le plus significatif (le bit de signe) de l'opérande est propagée aux positions de bit vides de poids fort. Lorsque l'opérande gauche de l' `>>` opérateur est d'un type intégral non signé, l'opérateur effectue un décalage logique vers la droite où les positions de bits vides d'ordre élevé ont toujours la valeur zéro. Pour effectuer l'opération inverse de celle déduite du type d'opérande, vous pouvez utiliser des casts explicites. Par exemple, si `x` est une variable de type `int` , l'opération `unchecked((int)((uint)x >> y))` effectue un décalage logique vers la droite de `x` .

Opérateurs relationnels et de test de type

Les `==` `!=` opérateurs,, `<` ,,, `>` `<=` `>=` `is` et `as` sont appelés opérateurs relationnels et de test de type.

antlr

```
relational_expression
    : shift_expression
    | relational_expression '<' shift_expression
    | relational_expression '>' shift_expression
    | relational_expression '<=' shift_expression
    | relational_expression '>=' shift_expression
    | relational_expression 'is' type
    | relational_expression 'as' type
    ;

equality_expression
    : relational_expression
    | equality_expression '==' relational_expression
    | equality_expression '!=' relational_expression
    ;
```

L' `is` opérateur est décrit dans l'opérateur is et l' `as` opérateur est décrit dans l'opérateur as.

Les `==` `!=` opérateurs,,, `<` `>` `<=` et `>=` sont des ***opérateurs de comparaison***.

Si un opérande d'un opérateur de comparaison a le type au moment de la compilation `dynamic` , l'expression est liée dynamiquement (liaison dynamique). Dans ce cas, le type au moment de la compilation de l'expression est `dynamic` , et la résolution décrite ci-dessous se produit au moment de l'exécution à l'aide du type d'exécution des opérandes qui ont le type au moment de la compilation `dynamic` .

Pour une opération de la forme `x` *op* `y` , où *op* est un opérateur de comparaison, la résolution de surcharge (résolution de surcharge d'opérateur binaire) est appliquée pour sélectionner une implémentation d'opérateur spécifique. Les opérandes sont convertis en types de paramètre de l'opérateur sélectionné, et le type du résultat est le type de retour de l'opérateur.

Les opérateurs de comparaison prédéfinis sont décrits dans les sections suivantes. Tous les opérateurs de comparaison prédéfinis retournent un résultat de type `bool` , comme décrit dans le tableau suivant.

Opération	Résultat
`x == y`	`true` Si `x` est égal à `y` , `false` sinon
`x != y`	`true` Si `x` n'est pas égal à `y` , `false` sinon
`x < y`	`true` si `x` est inférieur à `y`, `false` sinon
`x > y`	`true` si `x` est supérieur à `y`, `false` sinon
`x <= y`	`true` si `x` est inférieur ou égal à `y`, `false` sinon
`x >= y`	`true` si `x` est supérieur ou égal à `y`, `false` sinon

Opérateurs de comparaison d'entiers

Les opérateurs de comparaison d'entiers prédéfinis sont les suivants :

C#

```
bool operator ==(int x, int y);
bool operator ==(uint x, uint y);
bool operator ==(long x, long y);
bool operator ==(ulong x, ulong y);

bool operator !=(int x, int y);
bool operator !=(uint x, uint y);
bool operator !=(long x, long y);
bool operator !=(ulong x, ulong y);

bool operator <(int x, int y);
```

```
bool operator <(uint x, uint y);
bool operator <(long x, long y);
bool operator <(ulong x, ulong y);

bool operator >(int x, int y);
bool operator >(uint x, uint y);
bool operator >(long x, long y);
bool operator >(ulong x, ulong y);

bool operator <=(int x, int y);
bool operator <=(uint x, uint y);
bool operator <=(long x, long y);
bool operator <=(ulong x, ulong y);

bool operator >=(int x, int y);
bool operator >=(uint x, uint y);
bool operator >=(long x, long y);
bool operator >=(ulong x, ulong y);
```

Chacun de ces opérateurs compare les valeurs numériques des deux opérandes entiers et retourne une `bool` valeur qui indique si la relation particulière est `true` ou `false` .

Opérateurs de comparaison de nombres à virgule flottante

Les opérateurs de comparaison prédéfinis en virgule flottante sont les suivants :

C#

```
bool operator ==(float x, float y);
bool operator ==(double x, double y);

bool operator !=(float x, float y);
bool operator !=(double x, double y);

bool operator <(float x, float y);
bool operator <(double x, double y);

bool operator >(float x, float y);
bool operator >(double x, double y);

bool operator <=(float x, float y);
bool operator <=(double x, double y);

bool operator >=(float x, float y);
bool operator >=(double x, double y);
```

Les opérateurs comparent les opérandes en fonction des règles de la norme IEEE 754 :

- Si l'un des opérandes est NaN, le résultat est `false` pour tous les opérateurs sauf `!=` , pour lesquels le résultat est `true` . Pour les deux opérandes, `x != y` produit toujours le même résultat que `!(x == y)` . Toutefois, lorsque l'un des opérandes ou les deux sont Nan, les `<` opérateurs,, `>` `<=` et `>=` ne produisent pas les mêmes résultats que la négation logique de l'opérateur opposé. Par exemple, si l'une des deux `x` et `y` est NaN, `x < y` est `false` , mais `!(x >= y)` est `true` .

- Quand aucun des opérandes n'est NaN, les opérateurs comparent les valeurs des deux opérandes à virgule flottante en ce qui concerne le classement.

  C#

  ```
  -inf < -max < ... < -min < -0.0 == +0.0 < +min < ... < +max < +inf
  ```

  où `min` et `max` sont les valeurs finies positives les plus petites et les plus grandes qui peuvent être représentées dans le format à virgule flottante donné. Les effets notables de ce classement sont les suivants :
  - Les zéros positifs et négatifs sont considérés égaux.
  - Un infini négatif est considéré comme inférieur à toutes les autres valeurs, mais égal à un autre infini négatif.
  - Un infini positif est considéré comme supérieur à toutes les autres valeurs, mais égal à un autre infini positif.

Opérateurs de comparaison décimale

Les opérateurs de comparaison décimaux prédéfinis sont les suivants :

C#

```
bool operator ==(decimal x, decimal y);
bool operator !=(decimal x, decimal y);
bool operator <(decimal x, decimal y);
bool operator >(decimal x, decimal y);
bool operator <=(decimal x, decimal y);
bool operator >=(decimal x, decimal y);
```

Chacun de ces opérateurs compare les valeurs numériques des deux opérandes décimaux et retourne une `bool` valeur qui indique si la relation particulière est `true` ou `false` . Chaque comparaison décimale équivaut à utiliser l'opérateur relationnel ou d'égalité correspondant de type `System.Decimal` .

Opérateurs d'égalité booléennes

Les opérateurs d'égalité booléenne prédéfinis sont les suivants :

```
C#
```

```
bool operator ==(bool x, bool y);
bool operator !=(bool x, bool y);
```

Le résultat de `==` est `true` si et sont tous deux, `x` `y` `true` ou si et sont tous les deux `x` `y` `false` . Sinon, le résultat est `false`.

Le résultat de `!=` est `false` si et sont tous deux, `x` `y` `true` ou si et sont tous les deux `x` `y` `false` . Sinon, le résultat est `true`. Quand les opérandes sont de type `bool` , l' `!=` opérateur produit le même résultat que l' `^` opérateur.

Opérateurs de comparaison d'énumération

Chaque type d'énumération fournit implicitement les opérateurs de comparaison prédéfinis suivants :

```
C#
```

```
bool operator ==(E x, E y);
bool operator !=(E x, E y);
bool operator <(E x, E y);
bool operator >(E x, E y);
bool operator <=(E x, E y);
bool operator >=(E x, E y);
```

Le résultat de l'évaluation de `x op y` , où `x` et `y` sont les expressions d'un type énumération `E` avec un type sous-jacent `U` et `op` est l'un des opérateurs de comparaison, est exactement le même que l'évaluation `((U)x) op ((U)y)` . En d'autres termes, les opérateurs de comparaison de type énumération comparent simplement les valeurs intégrales sous-jacentes des deux opérandes.

Opérateurs d'égalité de type référence

Les opérateurs d'égalité de type référence prédéfinis sont les suivants :

```
C#
```

```
bool operator ==(object x, object y);
bool operator !=(object x, object y);
```

Les opérateurs retournent le résultat de la comparaison des deux références pour l'égalité ou l'absence d'égalité.

Étant donné que les opérateurs d'égalité de type référence prédéfinis acceptent les opérandes de type `object` , ils s'appliquent à tous les types qui ne déclarent pas les `operator ==` membres applicables et `operator !=` . À l'inverse, tous les opérateurs d'égalité définis par l'utilisateur applicables masquent effectivement les opérateurs d'égalité de type référence prédéfinis.

Les opérateurs d'égalité de type référence prédéfinis nécessitent l'un des éléments suivants :

- Les deux opérandes sont une valeur d'un type connu comme étant un *reference_type* ou le littéral `null` . En outre, une conversion de référence explicite (conversions de références explicites) existe à partir du type de l'un des opérandes vers le type de l'autre opérande.
- Un opérande est une valeur de type `T` où `T` est un *type_parameter* et l'autre opérande est le littéral `null` . En outre `T` , n'a pas la contrainte de type valeur.

À moins que l'une de ces conditions ne soit vraie, une erreur de liaison au moment de la liaison se produit. Les implications notables de ces règles sont les suivantes :

- Il s'agit d'une erreur de liaison au moment de la liaison pour utiliser les opérateurs d'égalité de type référence prédéfinis pour comparer deux références qui sont connues pour être différentes au moment de la liaison. Par exemple, si les types au moment de la liaison des opérandes sont deux types de classe `A` et `B` , et si ni n'est `A` `B` dérivé de l'autre, il serait impossible pour les deux opérandes de référencer le même objet. Ainsi, l'opération est considérée comme une erreur au moment de la liaison.
- Les opérateurs d'égalité de type référence prédéfinis n'autorisent pas la comparaison des opérandes de type valeur. Par conséquent, à moins qu'un type struct ne déclare ses propres opérateurs d'égalité, il n'est pas possible de comparer les valeurs de ce type struct.
- Les opérateurs d'égalité de type référence prédéfinis n'entraînent jamais la levée des opérations de boxing pour leurs opérandes. Il serait inutile d'effectuer ces opérations de boxing, car les références aux instances boxed nouvellement allouées seraient nécessairement différentes de toutes les autres références.
- Si un opérande d'un type de paramètre de type `T` est comparé à `null` , et que le type d'exécution de `T` est un type valeur, le résultat de la comparaison est `false` .

L'exemple suivant vérifie si un argument d'un type de paramètre de type sans contrainte est `null` .

C#

```
class C<T>
{
    void F(T x) {
        if (x == null) throw new ArgumentNullException();
        ...
    }
}
```

La `x == null` construction est autorisée même si `T` peut représenter un type valeur, et le résultat est simplement défini comme étant `false` `T` un type valeur.

Pour une opération sous la forme `x == y` ou `x != y` , si `operator ==` `operator !=` elle est applicable ou existante, les règles de résolution de surcharge d'opérateur (résolution de surcharge d'opérateur binaire) sélectionnent cet opérateur au lieu de l'opérateur d'égalité de type référence prédéfini. Toutefois, il est toujours possible de sélectionner l'opérateur d'égalité de type référence prédéfini en effectuant un cast explicite de l'un ou des deux opérandes en type `object` . L'exemple

C#

```
using System;

class Test
{
    static void Main() {
        string s = "Test";
        string t = string.Copy(s);
        Console.WriteLine(s == t);
        Console.WriteLine((object)s == t);
        Console.WriteLine(s == (object)t);
        Console.WriteLine((object)s == (object)t);
    }
}
```

génère la sortie

Console

```
True
False
False
False
```

Les `s` `t` variables et font référence à deux `string` instances distinctes contenant les mêmes caractères. La première comparaison `True` est due au fait que l'opérateur

d'égalité de chaînes prédéfini (opérateurs d'égalité de chaînes) est sélectionné lorsque les deux opérandes sont de type `string` . Les autres comparaisons sont toutes générées `False` , car l'opérateur d'égalité de type référence prédéfini est sélectionné lorsqu'au moins l'un des opérandes est de type `object` .

Notez que la technique ci-dessus n'est pas significative pour les types valeur. L'exemple

C#

```
class Test
{
    static void Main() {
        int i = 123;
        int j = 123;
        System.Console.WriteLine((object)i == (object)j);
    }
}
```

génère `False` des résultats, car les casts créent des références à deux instances distinctes de `int` valeurs boxed.

Opérateurs d'égalité de chaînes

Les opérateurs d'égalité de chaînes prédéfinis sont les suivants :

C#

```
bool operator ==(string x, string y);
bool operator !=(string x, string y);
```

Deux `string` valeurs sont considérées comme égales quand l'une des conditions suivantes est vraie :

- Les deux valeurs sont `null` .
- Les deux valeurs sont des références non null aux instances de chaîne qui ont des longueurs identiques et des caractères identiques dans chaque position de caractère.

Les opérateurs d'égalité de chaînes comparent les valeurs de chaîne plutôt que les références de chaîne. Lorsque deux instances de chaîne distinctes contiennent exactement la même séquence de caractères, les valeurs des chaînes sont égales, mais les références sont différentes. Comme décrit dans opérateurs d'égalité de type référence, les opérateurs d'égalité de type référence peuvent être utilisés pour comparer des références de chaîne à la place de valeurs de chaîne.

Opérateurs d'égalité de délégué

Chaque type délégué fournit implicitement les opérateurs de comparaison prédéfinis suivants :

```
C#
```

```
bool operator ==(System.Delegate x, System.Delegate y);
bool operator !=(System.Delegate x, System.Delegate y);
```

Deux instances de délégué sont considérées comme étant égales comme suit :

- Si l'une des instances de délégué est `null` , elles sont égales si et seulement si les deux sont `null` .
- Si les délégués ont un type au moment de l'exécution différent, ils ne sont jamais égaux.
- Si les deux instances de délégué ont une liste d'appel (déclarations de délégué), ces instances sont égales si et seulement si leurs listes d'appel sont de la même longueur, et que chaque entrée de la liste d'appel de l'une d'elles est égale (comme indiqué ci-dessous) à l'entrée correspondante, dans l'ordre, dans la liste d'appel de l'autre.

Les règles suivantes régissent l'égalité des entrées de la liste d'appel :

- Si deux entrées de liste d'appel font toutes deux référence à la même méthode statique, les entrées sont égales.
- Si deux entrées de liste d'appel font toutes deux référence à la même méthode non statique sur le même objet cible (tel que défini par les opérateurs d'égalité de référence), les entrées sont égales.
- Les entrées de la liste d'invocation générées à partir de l'évaluation d' *anonymous_method_expression* s ou d' *lambda_expression* s sémantiquement identiques avec le même jeu d'instances de variable externe capturées sont autorisées (mais pas obligatoires) pour être égales.

Opérateurs d'égalité et null

Les `==` `!=` opérateurs et permettent à un opérande d'être une valeur d'un type Nullable et l'autre est le `null` littéral, même si aucun opérateur prédéfini ou défini par l'utilisateur (sous forme non levée) n'existe pour l'opération.

Pour une opération de l'une des formes

```
C#
```

```
x == null
null == x
x != null
null != x
```

où `x` est une expression d'un type Nullable, si la résolution de surcharge d'opérateur (résolution de surcharge d'opérateur binaire) ne parvient pas à trouver un opérateur applicable, le résultat est à la place calculé à partir de la `HasValue` propriété de `x` . Plus précisément, les deux premières formes sont traduites en `!x.HasValue` , et les deux derniers sont traduits en `x.HasValue` .

L'opérateur is

L' `is` opérateur est utilisé pour vérifier dynamiquement si le type d'exécution d'un objet est compatible avec un type donné. Le résultat de l'opération `E is T` , où `E` est une expression et `T` est un type, est une valeur booléenne indiquant si `E` peut être converti en type `T` par une conversion de référence, une conversion boxing ou une conversion unboxing. L'opération est évaluée comme suit, après que les arguments de type ont été substitués à tous les paramètres de type :

- Si `E` est une fonction anonyme, une erreur de compilation se produit
- Si `E` est un groupe de méthodes ou le `null` littéral, si le type de `E` est un type référence ou un type Nullable et si la valeur de `E` est null, le résultat est false.
- Dans le cas contraire, laissez `D` représenter le type dynamique de `E` comme suit :
  - Si le type de `E` est un type référence, `D` est le type au moment de l'exécution de la référence de l'instance de `E` .
  - Si le type de `E` est un type Nullable, `D` est le type sous-jacent de ce type Nullable.
  - Si le type de `E` est un type valeur n'acceptant pas les valeurs NULL, `D` est le type de `E` .
- Le résultat de l'opération dépend de `D` et de `T` comme suit :
  - Si `T` est un type référence, le résultat est true si `D` et `T` sont du même type, si `D` est un type référence et une conversion de référence implicite de `D` en `T` Exists, ou si `D` est un type valeur et qu'une conversion boxing de `D` en `T` existe.
  - Si `T` est un type Nullable, le résultat est true si `D` est le type sous-jacent de `T` .
  - Si `T` est un type valeur qui n'autorise pas les valeurs NULL, le résultat est true si `D` et `T` sont du même type.
  - Dans le cas contraire, le résultat est false.

Notez que les conversions définies par l'utilisateur ne sont pas prises en compte par l' `is` opérateur.

L'opérateur as

L' `as` opérateur est utilisé pour convertir explicitement une valeur en un type référence ou un type Nullable donné. Contrairement à une expression de cast (expressions de cast), l' `as` opérateur ne lève jamais d'exception. Au lieu de cela, si la conversion indiquée n'est pas possible, la valeur résultante est `null` .

Dans une opération de la forme `E as T` , `E` doit être une expression et `T` doit être un type référence, un paramètre de type connu comme un type référence ou un type Nullable. En outre, au moins l'une des conditions suivantes doit être vraie, sinon une erreur de compilation se produit :

- Une identité (conversion d'identité), un Nullable implicite (conversions implicites Nullable), une référence implicite (conversions deréférence implicites), une conversion boxing (conversions boxing), un Nullable explicite (conversionsexplicites Nullable), une référence explicite (conversions de références explicites) ou une conversion unboxing (conversionunboxing) de `E` vers `T`
- Le type de `E` ou `T` est un type ouvert.
- `E` est le `null` littéral.

Si le type au moment de la compilation de `E` n'a pas la valeur `dynamic` , l'opération `E as T` produit le même résultat que

```
C#

E is T ? (T)(E) : (T)null
```

sauf que `E` n'est évalué qu'une seule fois. Le compilateur peut être censé optimiser `E as T` pour exécuter au plus un contrôle de type dynamique au lieu des deux contrôles de type dynamique qui sont implicites par l'expansion ci-dessus.

Si le type au moment de la compilation de `E` est `dynamic` , contrairement à l'opérateur de cast, l' `as` opérateur n'est pas lié dynamiquement (liaison dynamique). Par conséquent, dans ce cas, le développement est le suivant :

```
C#

E is T ? (T)(object)(E) : (T)null
```

Notez que certaines conversions, telles que les conversions définies par l'utilisateur, ne sont pas possibles avec l' `as` opérateur et doivent plutôt être effectuées à l'aide d'expressions de cast.

Dans l'exemple

C#

```
class X
{

    public string F(object o) {
        return o as string;        // OK, string is a reference type
    }

    public T G<T>(object o) where T: Attribute {
        return o as T;             // Ok, T has a class constraint
    }

    public U H<U>(object o) {
        return o as U;             // Error, U is unconstrained
    }
}
```

le paramètre `T` de type de `G` est connu comme étant un type référence, car il a la contrainte de classe. Le paramètre `U` de type de `H` n'est toutefois pas ; par conséquent, l'utilisation de l' `as` opérateur dans `H` est interdite.

Opérateurs logiques

Les `&` `^` opérateurs, et `|` sont appelés opérateurs logiques.

antlr

```
and_expression
    : equality_expression
    | and_expression '&' equality_expression
    ;

exclusive_or_expression
    : and_expression
    | exclusive_or_expression '^' and_expression
    ;

inclusive_or_expression
    : exclusive_or_expression
    | inclusive_or_expression '|' exclusive_or_expression
    ;
```

Si un opérande d'un opérateur logique a le type au moment de la compilation `dynamic`, l'expression est liée dynamiquement (liaison dynamique). Dans ce cas, le type au moment de la compilation de l'expression est `dynamic`, et la résolution décrite ci-dessous se produit au moment de l'exécution à l'aide du type d'exécution des opérandes qui ont le type au moment de la compilation `dynamic`.

Pour une opération de la forme `x op y`, où `op` est l'un des opérateurs logiques, la résolution de surcharge (résolution de surcharge d'opérateur binaire) est appliquée pour sélectionner une implémentation d'opérateur spécifique. Les opérandes sont convertis en types de paramètre de l'opérateur sélectionné, et le type du résultat est le type de retour de l'opérateur.

Les opérateurs logiques prédéfinis sont décrits dans les sections suivantes.

Opérateurs logiques Integer

Les opérateurs logiques d'entiers prédéfinis sont les suivants :

C#

```
int operator &(int x, int y);
uint operator &(uint x, uint y);
long operator &(long x, long y);
ulong operator &(ulong x, ulong y);

int operator |(int x, int y);
uint operator |(uint x, uint y);
long operator |(long x, long y);
ulong operator |(ulong x, ulong y);

int operator ^(int x, int y);
uint operator ^(uint x, uint y);
long operator ^(long x, long y);
ulong operator ^(ulong x, ulong y);
```

L' `&` opérateur calcule le logique au niveau du bit `AND` des deux opérandes, l' `|` opérateur calcule le logique au niveau du bit `OR` des deux opérandes, et l' `^` opérateur calcule l'exception logique au niveau du bit `OR` des deux opérandes. Aucun dépassement de capacité n'est possible à partir de ces opérations.

Opérateurs logiques d'énumération

Chaque type d'énumération `E` fournit implicitement les opérateurs logiques prédéfinis suivants :

C#

```
E operator &(E x, E y);
E operator |(E x, E y);
E operator ^(E x, E y);
```

Le résultat de l'évaluation de `x op y` , où `x` et `y` sont les expressions d'un type énumération `E` avec un type sous-jacent `U` et `op` est l'un des opérateurs logiques, est exactement le même que l'évaluation `(E)((U)x op (U)y)` . En d'autres termes, les opérateurs logiques de type énumération effectuent simplement l'opération logique sur le type sous-jacent des deux opérandes.

Opérateurs logiques booléens

Les opérateurs logiques booléens prédéfinis sont les suivants :

C#

```
bool operator &(bool x, bool y);
bool operator |(bool x, bool y);
bool operator ^(bool x, bool y);
```

Le résultat de `x & y` est `true` si `x` et `y` sont `true`. Sinon, le résultat est `false`.

Le résultat de `x | y` est `true` si `x` ou `y` a `true` la valeur. Sinon, le résultat est `false`.

Le résultat de est si a la valeur et que a la valeur `x ^ y` `true` `x` `true` `y` `false` , ou a la valeur `x` `false` et `y` a `true` la valeur. Sinon, le résultat est `false`. Quand les opérandes sont de type `bool` , l' `^` opérateur calcule le même résultat que l' `!=` opérateur.

Opérateurs logiques booléens Nullable

Le type booléen Nullable `bool?` peut représenter trois valeurs, `true` , `false` et `null` , et est conceptuellement similaire au type à trois valeurs utilisé pour les expressions booléennes dans SQL. Pour vous assurer que les résultats produits par `&` les `|` opérateurs et pour les `bool?` opérandes sont cohérents avec la logique à trois valeurs de SQL, les opérateurs prédéfinis suivants sont fournis :

C#

```
bool? operator &(bool? x, bool? y);
bool? operator |(bool? x, bool? y);
```

Le tableau suivant répertorie les résultats produits par ces opérateurs pour toutes les combinaisons des valeurs `true` , `false` et `null` .

`x`	`y`	`x & y`	`x | y`
`true`	`true`	`true`	`true`
`true`	`false`	`false`	`true`
`true`	`null`	`null`	`true`
`false`	`true`	`false`	`true`
`false`	`false`	`false`	`false`
`false`	`null`	`false`	`null`
`null`	`true`	`null`	`true`
`null`	`false`	`false`	`null`
`null`	`null`	`null`	`null`

Opérateurs logiques conditionnels

Les opérateurs `&&` et `||` sont appelés opérateurs logiques conditionnels. Elles sont également appelées opérateurs logiques de « court-circuit ».

antlr

```
conditional_and_expression
    : inclusive_or_expression
    | conditional_and_expression '&&' inclusive_or_expression
    ;

conditional_or_expression
    : conditional_and_expression
    | conditional_or_expression '||' conditional_and_expression
    ;
```

Les `&&` `||` opérateurs et sont des versions conditionnelles `&` des `|` opérateurs et :

- L'opération `x && y` correspond à l'opération `x & y` , sauf qu' `y` elle est évaluée uniquement si `x` n'a pas la valeur `false` .
- L'opération `x || y` correspond à l'opération `x | y` , sauf qu' `y` elle est évaluée uniquement si `x` n'a pas la valeur `true` .

Si un opérande d'un opérateur logique conditionnel a le type au moment de la compilation `dynamic`, l'expression est liée dynamiquement (liaison dynamique). Dans ce cas, le type au moment de la compilation de l'expression est `dynamic`, et la résolution décrite ci-dessous se produit au moment de l'exécution à l'aide du type d'exécution des opérandes qui ont le type au moment de la compilation `dynamic`.

Une opération du formulaire `x && y` ou `x || y` est traitée en appliquant la résolution de surcharge (résolution de surcharge d'opérateur binaire) comme si l'opération avait été écrite `x & y` ou `x | y`. Ainsi,

- Si la résolution de surcharge ne parvient pas à trouver un seul opérateur le mieux, ou si la résolution de surcharge sélectionne l'un des opérateurs logiques d'entiers prédéfinis, une erreur de liaison s'est produite.
- Sinon, si l'opérateur sélectionné est l'un des opérateurs logiques booléens prédéfinis (opérateurs logiques booléens) ou des opérateurs logiques booléens Nullable (opérateurs logiques booléens Nullable), l'opération est traitée comme décrit dans opérateurs logiques conditionnels booléens.
- Dans le cas contraire, l'opérateur sélectionné est un opérateur défini par l'utilisateur et l'opération est traitée comme décrit dans opérateurs logiques conditionnels définis par l'utilisateur.

Il n'est pas possible de surcharger directement les opérateurs logiques conditionnels. Toutefois, étant donné que les opérateurs logiques conditionnels sont évalués en termes d'opérateurs logiques standard, les surcharges des opérateurs logiques normaux sont, avec certaines restrictions, également considérées comme des surcharges des opérateurs logiques conditionnels. Cela est décrit plus en détail dans opérateurs logiques conditionnels définis par l'utilisateur.

Opérateurs logiques conditionnels booléens

Lorsque les opérandes de `&&` ou `||` sont de type `bool`, ou lorsque les opérandes sont de types qui ne définissent pas un applicable `operator &` ou `operator |`, mais qui définissent des conversions implicites en `bool`, l'opération est traitée comme suit :

- L'opération `x && y` est évaluée comme `x ? y : false`. En d'autres termes, `x` est d'abord évalué et converti en type `bool`. Ensuite, si `x` a `true` la valeur, `y` est évalué et converti en type `bool`, ce qui devient le résultat de l'opération. Dans le cas contraire, le résultat de l'opération est `false`.
- L'opération `x || y` est évaluée comme `x ? true : y`. En d'autres termes, `x` est d'abord évalué et converti en type `bool`. Ensuite, si `x` a `true` la valeur, le résultat

de l'opération est `true` . Sinon, `y` est évalué et converti en type `bool` , ce qui devient le résultat de l'opération.

Opérateurs logiques conditionnels définis par l'utilisateur

Quand les opérandes de `&&` ou `||` sont de types qui déclarent un défini par l'utilisateur applicable `operator &` ou `operator |` , les deux conditions suivantes doivent être vraies, où `T` est le type dans lequel l'opérateur sélectionné est déclaré :

- Le type de retour et le type de chaque paramètre de l'opérateur sélectionné doivent être `T` . En d'autres termes, l'opérateur doit calculer le logique `AND` ou le logique `OR` de deux opérandes de type `T` , et doit retourner un résultat de type `T` .
- `T` doit contenir des déclarations de `operator true` et `operator false` .

Une erreur de liaison au moment de la liaison se produit si l'une de ces conditions n'est pas satisfaite. Dans le cas contraire, l' `&& ||` opération ou est évaluée par la combinaison de l'opérateur défini par l'utilisateur `operator true` ou `operator false` de l'opérateur défini par l'utilisateur sélectionné :

- L'opération `x && y` est évaluée comme `T.false(x) ? x : T.&(x, y)` , où `T.false(x)` est un appel de la `operator false` déclarée dans `T` et `T.&(x, y)` est un appel du sélectionné `operator &` . En d'autres termes, `x` est évaluée en premier et `operator false` est appelé sur le résultat pour déterminer si `x` a la valeur false. Ensuite, si `x` a la valeur false, le résultat de l'opération est la valeur précédemment calculée pour `x` . Sinon, `y` est évalué et le sélectionné `operator &` est appelé sur la valeur précédemment calculée pour `x` et la valeur calculée pour `y` pour produire le résultat de l'opération.
- L'opération `x || y` est évaluée comme `T.true(x) ? x : T.|(x, y)` , où `T.true(x)` est un appel de la `operator true` déclarée dans `T` et `T.|(x,y)` est un appel du sélectionné `operator|` . En d'autres termes, `x` est évaluée en premier et `operator true` est appelé sur le résultat pour déterminer si `x` a la valeur true. Ensuite, si `x` a la valeur true, le résultat de l'opération est la valeur précédemment calculée pour `x` . Sinon, `y` est évalué et le sélectionné `operator |` est appelé sur la valeur précédemment calculée pour `x` et la valeur calculée pour `y` pour produire le résultat de l'opération.

Dans l'une ou l'autre de ces opérations, l'expression donnée par `x` n'est évaluée qu'une seule fois, et l'expression donnée par `y` n'est pas évaluée ou évaluée une seule fois.

Pour obtenir un exemple de type qui implémente `operator true` et `operator false` , consultez Database Boolean type.

Opérateur de fusion de Null

L' `??` opérateur est appelé opérateur de fusion Null.

antlr

```
null_coalescing_expression
    : conditional_or_expression
    | conditional_or_expression '??' null_coalescing_expression
    ;
```

Une expression de fusion Null de la forme `a ?? b` requiert `a` un type Nullable ou un type référence. Si `a` est non null, le résultat de `a ?? b` est `a` ; sinon, le résultat est `b` . L'opération évalue `b` uniquement si `a` a la valeur null.

L'opérateur de fusion Null est associatif à droite, ce qui signifie que les opérations sont regroupées de droite à gauche. Par exemple, une expression de la forme `a ?? b ?? c` est évaluée comme `a ?? (b ?? c)` . En général, une expression de la forme `E1 ?? E2 ?? ... ?? En` retourne le premier des opérandes qui n'est pas null, ou null si tous les opérandes ont la valeur null.

Le type de l'expression `a ?? b` dépend des conversions implicites disponibles sur les opérandes. Par ordre de préférence, le type de `a ?? b` est `A0` , `A` ou `B` , où `A` est le type de `a` (fourni qui `a` a un type), `B` est le type de `b` (fourni qui `b` a un type) et `A0` est le type sous-jacent de `A` si `A` est un type Nullable, ou dans le `A` cas contraire. Plus précisément, `a ?? b` est traité comme suit :

- Si `A` Exists et n'est pas un type Nullable ou un type référence, une erreur de compilation se produit.
- Si `b` est une expression dynamique, le type de résultat est `dynamic` . Au moment de l'exécution, `a` est évaluée pour la première fois. Si `a` n'a pas la valeur null, `a` est converti en dynamique, ce qui devient le résultat. Dans le cas contraire, `b` est évaluée et devient le résultat.
- Sinon, si `A` Exists et est un type Nullable et si une conversion implicite existe de `b` en `A0` , le type de résultat est `A0` . Au moment de l'exécution, `a` est évaluée pour la première fois. Si `a` n'a pas la valeur null, `a` est désencapsulée dans `A0` le type et devient le résultat. Sinon, `b` est évalué et converti en type `A0` , ce qui devient le résultat.

- Sinon, si `A` Exists et qu'une conversion implicite existe de `b` en `A` , le type de résultat est `A` . Au moment de l'exécution, `a` est évaluée pour la première fois. Si `a` n'a pas la valeur null, `a` devient le résultat. Sinon, `b` est évalué et converti en type `A` , ce qui devient le résultat.
- Sinon, si `b` a un type `B` et qu'il existe une conversion implicite de `a` en `B` , le type de résultat est `B` . Au moment de l'exécution, `a` est évaluée pour la première fois. Si `a` n'a pas la valeur null, `a` est désencapsulé dans `A0` le type (s' `A` il existe et est Nullable) et converti en type `B` , ce qui devient le résultat. Sinon, `b` est évalué et devient le résultat.
- Sinon, `a` et `b` sont incompatibles, et une erreur de compilation se produit.

Opérateur conditionnel

L' `?:` opérateur est appelé opérateur conditionnel. Il est parfois également appelé opérateur ternaire.

antlr

```
conditional_expression
    : null_coalescing_expression
    | null_coalescing_expression '?' expression ':' expression
    ;
```

Une expression conditionnelle de la forme `b ? x : y` évalue d'abord la condition `b` . Ensuite, si `b` a `true` la valeur, `x` est évaluée et devient le résultat de l'opération. Sinon, `y` est évalué et devient le résultat de l'opération. Une expression conditionnelle n'évalue jamais `x` et `y` .

L'opérateur conditionnel est associatif à droite, ce qui signifie que les opérations sont regroupées de droite à gauche. Par exemple, une expression de la forme `a ? b : c ? d : e` est évaluée comme `a ? b : (c ? d : e)` .

Le premier opérande de l' `?:` opérateur doit être une expression qui peut être implicitement convertie en `bool` ou une expression d'un type qui implémente `operator true` . Si aucune de ces exigences n'est satisfaite, une erreur de compilation se produit.

Le deuxième et le troisième opérandes, `x` et `y` , de l' `?:` opérateur contrôlent le type de l'expression conditionnelle.

- Si `x` a le type `X` et `y` a le type `Y` ,

    - Si une conversion implicite (conversions implicites) existe de `X` à `Y` , mais pas de à `Y` `X` , `Y` est le type de l'expression conditionnelle.
    - Si une conversion implicite (conversions implicites) existe de `Y` à `X` , mais pas de à `X` `Y` , `X` est le type de l'expression conditionnelle.
    - Dans le cas contraire, aucun type d'expression ne peut être déterminé et une erreur de compilation se produit.
- Si seul un de `x` et `y` a un type, et `x` que et `y` , de sont implicitement convertibles en ce type, alors il s'agit du type de l'expression conditionnelle.
- Dans le cas contraire, aucun type d'expression ne peut être déterminé et une erreur de compilation se produit.

Le traitement au moment de l'exécution d'une expression conditionnelle de la forme `b ? x : y` se compose des étapes suivantes :

- Tout d'abord, `b` est évaluée et la `bool` valeur de `b` est déterminée :
    - Si une conversion implicite du type de `b` en `bool` existe, cette conversion implicite est effectuée pour produire une `bool` valeur.
    - Sinon, le `operator true` défini par le type de `b` est appelé pour produire une `bool` valeur.
- Si la `bool` valeur produite par l'étape ci-dessus est `true` , `x` est évaluée et convertie en type de l'expression conditionnelle, et devient le résultat de l'expression conditionnelle.
- Sinon, `y` est évalué et converti en type de l'expression conditionnelle, et devient le résultat de l'expression conditionnelle.

Expressions de fonctions anonymes

Une ***fonction anonyme*** est une expression qui représente une définition de méthode « in-line ». Une fonction anonyme n'a pas de valeur ou de type dans et de elle-même, mais elle peut être convertie en un délégué compatible ou un type d'arborescence d'expression. L'évaluation d'une conversion de fonction anonyme dépend du type de cible de la conversion : s'il s'agit d'un type délégué, la conversion correspond à une valeur de délégué qui référence la méthode définie par la fonction anonyme. S'il s'agit d'un type d'arborescence d'expression, la conversion prend la valeur d'une arborescence d'expressions qui représente la structure de la méthode sous la forme d'une structure d'objet.

Pour des raisons historiques, il existe deux versions syntaxiques des fonctions anonymes, à savoir *lambda_expression* s et *anonymous_method_expression* s. Pour presque tous les besoins, *lambda_expression* s sont plus concises et plus expressifs que

anonymous_method_expression s, qui restent dans le langage à des fins de compatibilité descendante.

antlr

```
lambda_expression
    : anonymous_function_signature '=>' anonymous_function_body
    ;

anonymous_method_expression
    : 'delegate' explicit_anonymous_function_signature? block
    ;

anonymous_function_signature
    : explicit_anonymous_function_signature
    | implicit_anonymous_function_signature
    ;

explicit_anonymous_function_signature
    : '(' explicit_anonymous_function_parameter_list? ')'
    ;

explicit_anonymous_function_parameter_list
    : explicit_anonymous_function_parameter (','
explicit_anonymous_function_parameter)*
    ;

explicit_anonymous_function_parameter
    : anonymous_function_parameter_modifier? type identifier
    ;

anonymous_function_parameter_modifier
    : 'ref'
    | 'out'
    ;

implicit_anonymous_function_signature
    : '(' implicit_anonymous_function_parameter_list? ')'
    | implicit_anonymous_function_parameter
    ;

implicit_anonymous_function_parameter_list
    : implicit_anonymous_function_parameter (','
implicit_anonymous_function_parameter)*
    ;

implicit_anonymous_function_parameter
    : identifier
    ;

anonymous_function_body
    : expression
    | block
    ;
```

L' `=>` opérateur a la même priorité que l'assignation (`=`) et est associatif à droite.

Une fonction anonyme avec le `async` modificateur est une fonction Async et suit les règles décrites dans fonctions Async.

Les paramètres d'une fonction anonyme sous la forme d'un *lambda_expression* peuvent être explicitement ou implicitement typés. Dans une liste de paramètres typés explicitement, le type de chaque paramètre est défini explicitement. Dans une liste de paramètres typée implicitement, les types des paramètres sont déduits du contexte dans lequel la fonction anonyme se produit, en particulier lorsque la fonction anonyme est convertie en un type délégué compatible ou un type d'arborescence d'expression, ce type fournit les types de paramètres (conversions de fonctions anonymes).

Dans une fonction anonyme avec un seul paramètre typé implicitement, les parenthèses peuvent être omises dans la liste de paramètres. En d'autres termes, une fonction anonyme de la forme

```
C#

( param ) => expr
```

peut être abrégé en

```
C#

param => expr
```

La liste des paramètres d'une fonction anonyme sous la forme d'un *anonymous_method_expression* est facultative. Si elles sont spécifiées, les paramètres doivent être explicitement typés. Si ce n'est pas le cas, la fonction anonyme est convertible en un délégué avec une liste de paramètres qui ne contient pas de `out` paramètres.

Un corps de *bloc* d'une fonction anonyme est accessible (points de terminaison et accessibilité) sauf si la fonction anonyme se produit à l'intérieur d'une instruction inaccessible.

Voici quelques exemples de fonctions anonymes :

```
C#

x => x + 1                          // Implicitly typed, expression body
x => { return x + 1; }              // Implicitly typed, statement body
```

```
(int x) => x + 1                        // Explicitly typed, expression body
(int x) => { return x + 1; }            // Explicitly typed, statement body
(x, y) => x * y                         // Multiple parameters
() => Console.WriteLine()               // No parameters
async (t1,t2) => await t1 + await t2    // Async
delegate (int x) { return x + 1; }      // Anonymous method expression
delegate { return 1 + 1; }              // Parameter list omitted
```

Le comportement de *lambda_expression* s et *anonymous_method_expression* s est le même, à l'exception des points suivants :

- *anonymous_method_expression* s permettent l'omission complète de la liste de paramètres, ce qui donne convertibilité aux types délégués d'une liste de paramètres de valeur.
- *lambda_expression* s autorisent l'omission et la déduction des types de paramètres, tandis que les types de paramètres doivent être déclarés explicitement *anonymous_method_expression* s.
- Le corps d'un *lambda_expression* peut être une expression ou un bloc d'instructions, tandis que le corps d'un *anonymous_method_expression* doit être un bloc d'instructions.
- Seuls les *lambda_expression* s ont des conversions en types d'arborescences d'expressions compatibles (types d'arborescence d'expression).

Signatures de fonctions anonymes

La *anonymous_function_signature* facultative d'une fonction anonyme définit les noms et éventuellement les types des paramètres formels pour la fonction anonyme. La portée des paramètres de la fonction anonyme est le *anonymous_function_body*. (Étendues) Avec la liste de paramètres (si elle est donnée), le corps de la méthode anonyme constitue un espace de déclaration (déclarations). Il s'agit donc d'une erreur de compilation pour le nom d'un paramètre de la fonction anonyme qui correspond au nom d'une variable locale, d'une constante locale ou d'un paramètre dont l'étendue comprend le *anonymous_method_expression* ou *lambda_expression*.

Si une fonction anonyme a un *explicit_anonymous_function_signature*, l'ensemble de types de délégués compatibles et de types d'arborescence d'expression est limité à ceux qui ont les mêmes types de paramètres et modificateurs dans le même ordre. Contrairement aux conversions de groupe de méthodes (conversions de groupe de méthodes), la contre-variance des types de paramètre de fonction anonyme n'est pas prise en charge. Si une fonction anonyme n'a pas de *anonymous_function_signature*, l'ensemble des types de délégués compatibles et des types d'arborescence d'expression est limité à ceux qui n'ont pas de `out` paramètres.

Notez qu'un *anonymous_function_signature* ne peut pas inclure d'attributs ou un tableau de paramètres. Toutefois, un *anonymous_function_signature* peut être compatible avec un type délégué dont la liste de paramètres contient un tableau de paramètres.

Notez également que la conversion en un type d'arborescence d'expression, même si elle est compatible, peut encore échouer au moment de la compilation (types d'arborescence d'expression).

Corps des fonctions anonymes

Le corps (*expression* ou *bloc*) d'une fonction anonyme est soumis aux règles suivantes :

- Si la fonction anonyme comprend une signature, les paramètres spécifiés dans la signature sont disponibles dans le corps. Si la fonction anonyme n'a pas de signature, elle peut être convertie en type délégué ou type d'expression ayant des paramètres (conversions de fonctions anonymes), mais les paramètres ne sont pas accessibles dans le corps.
- À l'exception des `ref` `out` paramètres ou spécifiés dans la signature (le cas échéant) de la fonction anonyme englobante la plus proche, il s'agit d'une erreur au moment de la compilation pour que le corps accède à un `ref` `out` paramètre ou.
- Quand le type de `this` est un type struct, il s'agit d'une erreur au moment de la compilation pour que le corps accède `this` . Cela est vrai, que l'accès soit explicite (comme dans `this.x`) ou implicite (comme dans `x` où `x` est un membre d'instance du struct). Cette règle interdit simplement cet accès et n'affecte pas si la recherche de membres est un membre du struct.
- Le corps a accès aux variables externes (variables externes) de la fonction anonyme. L'accès à une variable externe fait référence à l'instance de la variable qui est active au moment où le *lambda_expression* ou *anonymous_method_expression* est évalué (évaluation des expressions de fonction anonymes).
- Il s'agit d'une erreur au moment de la compilation pour que le corps contienne une `goto` instruction, une `break` instruction ou une `continue` instruction dont la cible est en dehors du corps ou dans le corps d'une fonction anonyme contenue.
- Une `return` instruction dans le corps retourne le contrôle à partir d'un appel de la fonction anonyme englobante la plus proche, et non du membre de la fonction englobante. Une expression spécifiée dans une `return` instruction doit être implicitement convertible en type de retour du type délégué ou du type d'arborescence d'expression vers lequel le *lambda_expression* ou le *anonymous_method_expression* englobant le plus proche est converti (conversions de fonctions anonymes).

Elle n'est pas explicitement spécifiée, qu'il existe un moyen d'exécuter le bloc d'une fonction anonyme autre que par le biais de l'évaluation et de l'appel du *lambda_expression* ou *anonymous_method_expression*. En particulier, le compilateur peut choisir d'implémenter une fonction anonyme en synthétisant une ou plusieurs méthodes ou types nommés. Les noms de ces éléments synthétisés doivent être d'une forme réservée à l'utilisation du compilateur.

Résolution de surcharge et fonctions anonymes

Les fonctions anonymes dans une liste d'arguments participent à l'inférence de type et à la résolution de surcharge. Reportez-vous à l' inférence de type et à la résolution de surcharge pour les règles exactes.

L'exemple suivant illustre l'effet des fonctions anonymes sur la résolution de surcharge.

C#

```
class ItemList<T>: List<T>
{
    public int Sum(Func<T,int> selector) {
        int sum = 0;
        foreach (T item in this) sum += selector(item);
        return sum;
    }

    public double Sum(Func<T,double> selector) {
        double sum = 0;
        foreach (T item in this) sum += selector(item);
        return sum;
    }
}
```

La `ItemList<T>` classe a deux `Sum` méthodes. Chacun accepte un `selector` argument, qui extrait la valeur à additionner d'un élément de liste. La valeur extraite peut être un `int` ou un `double` et la somme résultante est également un `int` ou un `double` .

Les `Sum` méthodes peuvent par exemple être utilisées pour calculer des sommes à partir d'une liste de lignes de détails dans un ordre.

C#

```
class Detail
{
    public int UnitCount;
    public double UnitPrice;
    ...
}
```

```
void ComputeSums() {
    ItemList<Detail> orderDetails = GetOrderDetails(...);
    int totalUnits = orderDetails.Sum(d => d.UnitCount);
    double orderTotal = orderDetails.Sum(d => d.UnitPrice * d.UnitCount);
    ...
}
```

Lors du premier appel de `orderDetails.Sum` , les deux `Sum` méthodes sont applicables, car la fonction anonyme `d => d. UnitCount` est compatible avec `Func<Detail,int>` et `Func<Detail,double>` . Toutefois, la résolution de surcharge choisit la première `Sum` méthode, car la conversion vers `Func<Detail,int>` est meilleure que la conversion en `Func<Detail,double>` .

Dans le deuxième appel de `orderDetails.Sum` , seule la deuxième `Sum` méthode est applicable, car la fonction anonyme `d => d.UnitPrice * d.UnitCount` produit une valeur de type `double` . Par conséquent, la résolution de surcharge choisit la deuxième `Sum` méthode pour cet appel.

Fonctions anonymes et liaison dynamique

Une fonction anonyme ne peut pas être un récepteur, un argument ou un opérande d'une opération liée dynamiquement.

Variables externes

Toute variable locale, paramètre de valeur ou tableau de paramètres dont la portée comprend la *lambda_expression* ou *anonymous_method_expression* est appelée une ***variable externe*** de la fonction anonyme. Dans une fonction membre d'instance d'une classe, la `this` valeur est considérée comme un paramètre de valeur et est une variable externe de toute fonction anonyme contenue dans la fonction membre.

Variables externes capturées

Lorsqu'une variable externe est référencée par une fonction anonyme, la variable externe est dite ***capturée*** par la fonction anonyme. En règle générale, la durée de vie d'une variable locale est limitée à l'exécution du bloc ou de l'instruction à laquelle elle est associée (variables locales). Toutefois, la durée de vie d'une variable externe capturée est étendue au moins jusqu'à ce que le délégué ou l'arborescence d'expressions créé à partir de la fonction anonyme devienne éligible pour garbage collection.

Dans l'exemple

C#

```
using System;

delegate int D();

class Test
{
    static D F() {
        int x = 0;
        D result = () => ++x;
        return result;
    }

    static void Main() {
        D d = F();
        Console.WriteLine(d());
        Console.WriteLine(d());
        Console.WriteLine(d());
    }
}
```

la variable locale `x` est capturée par la fonction anonyme, et la durée de vie de `x` est étendue au moins jusqu'à ce que le délégué retourné par `F` devienne éligible pour garbage collection (ce qui ne se produit pas jusqu'à la fin du programme). Étant donné que chaque appel de la fonction anonyme opère sur la même instance de `x` , la sortie de l'exemple est la suivante :

Console

```
1
2
3
```

Lorsqu'une variable locale ou un paramètre de valeur est capturé par une fonction anonyme, la variable locale ou le paramètre n'est plus considéré comme une variable fixe (variables fixes et déplaçables), mais est considéré comme une variable déplaçable. Ainsi, tout `unsafe` code qui prend l'adresse d'une variable externe capturée doit d'abord utiliser l' `fixed` instruction pour corriger la variable.

Notez que contrairement à une variable non capturée, une variable locale capturée peut être exposée simultanément à plusieurs threads d'exécution.

Instanciation de variables locales

Une variable locale est considérée comme ***instanciée*** lorsque l'exécution entre dans l'étendue de la variable. Par exemple, lorsque la méthode suivante est appelée, la variable locale `x` est instanciée et initialisée trois fois, une fois pour chaque itération de la boucle.

C#

```
static void F() {
    for (int i = 0; i < 3; i++) {
        int x = i * 2 + 1;
        ...
    }
}
```

Toutefois, le déplacement de la déclaration de `x` l'extérieur de la boucle entraîne une seule instanciation de `x` :

C#

```
static void F() {
    int x;
    for (int i = 0; i < 3; i++) {
        x = i * 2 + 1;
        ...
    }
}
```

Lorsqu'elles ne sont pas capturées, il n'existe aucun moyen d'observer exactement la fréquence à laquelle une variable locale est instanciée. étant donné que les durées de vie des instanciations sont disjointes, il est possible que chaque instanciation utilise simplement le même emplacement de stockage. Toutefois, lorsqu'une fonction anonyme capture une variable locale, les effets de l'instanciation deviennent évidents.

L'exemple

C#

```
using System;

delegate void D();

class Test
{
    static D[] F() {
        D[] result = new D[3];
        for (int i = 0; i < 3; i++) {
            int x = i * 2 + 1;
            result[i] = () => { Console.WriteLine(x); };
```

```
        }
        return result;
    }

    static void Main() {
        foreach (D d in F()) d();
    }
}
```

génère cette sortie :

Console

```
1
3
5
```

Toutefois, lorsque la déclaration de `x` est déplacée en dehors de la boucle :

C#

```
static D[] F() {
    D[] result = new D[3];
    int x;
    for (int i = 0; i < 3; i++) {
        x = i * 2 + 1;
        result[i] = () => { Console.WriteLine(x); };
    }
    return result;
}
```

la sortie est la suivante :

Console

```
5
5
5
```

Si une boucle for déclare une variable d'itération, cette variable est considérée comme déclarée en dehors de la boucle. Ainsi, si l'exemple est modifié pour capturer la variable d'itération elle-même :

C#

```
static D[] F() {
    D[] result = new D[3];
    for (int i = 0; i < 3; i++) {
```

```
        result[i] = () => { Console.WriteLine(i); };
    }
    return result;
}
```

une seule instance de la variable d'itération est capturée, ce qui génère la sortie :

Console

```
3
3
3
```

Il est possible que les délégués de fonction anonymes partagent des variables capturées, mais des instances distinctes d'autres. Par exemple, si la valeur `F` de est remplacée par

C#

```
static D[] F() {
    D[] result = new D[3];
    int x = 0;
    for (int i = 0; i < 3; i++) {
        int y = 0;
        result[i] = () => { Console.WriteLine("{0} {1}", ++x, ++y); };
    }
    return result;
}
```

les trois délégués capturent la même instance de `x` , mais des instances distinctes de `y` , et la sortie est :

Console

```
1 1
2 1
3 1
```

Des fonctions anonymes distinctes peuvent capturer la même instance d'une variable externe. Dans l'exemple :

C#

```
using System;

delegate void Setter(int value);
```

```
delegate int Getter();

class Test
{
    static void Main() {
        int x = 0;
        Setter s = (int value) => { x = value; };
        Getter g = () => { return x; };
        s(5);
        Console.WriteLine(g());
        s(10);
        Console.WriteLine(g());
    }
}
```

les deux fonctions anonymes capturent la même instance de la variable locale `x` et peuvent donc « communiquer » par le biais de cette variable. La sortie de l'exemple est la suivante :

Console

```
5
10
```

Évaluation des expressions de fonction anonymes

Une fonction anonyme `F` doit toujours être convertie en un type délégué `D` ou un type d'arborescence d'expression `E` , soit directement, soit par le biais de l'exécution d'une expression de création de délégué `new D(F)` . Cette conversion détermine le résultat de la fonction anonyme, comme décrit dans conversions de fonctions anonymes.

Expressions de requête

Les ***expressions de requête*** fournissent une syntaxe intégrée au langage pour les requêtes qui est semblable aux langages de requête relationnels et hiérarchiques tels que SQL et XQuery.

antlr

```
query_expression
    : from_clause query_body
    ;

from_clause
    : 'from' type? identifier 'in' expression
    ;
```

```
query_body
    : query_body_clauses? select_or_group_clause query_continuation?
    ;

query_body_clauses
    : query_body_clause
    | query_body_clauses query_body_clause
    ;

query_body_clause
    : from_clause
    | let_clause
    | where_clause
    | join_clause
    | join_into_clause
    | orderby_clause
    ;

let_clause
    : 'let' identifier '=' expression
    ;

where_clause
    : 'where' boolean_expression
    ;

join_clause
    : 'join' type? identifier 'in' expression 'on' expression 'equals'
expression
    ;

join_into_clause
    : 'join' type? identifier 'in' expression 'on' expression 'equals'
expression 'into' identifier
    ;

orderby_clause
    : 'orderby' orderings
    ;

orderings
    : ordering (',' ordering)*
    ;

ordering
    : expression ordering_direction?
    ;

ordering_direction
    : 'ascending'
    | 'descending'
    ;

select_or_group_clause
```

```
    : select_clause
    | group_clause
    ;

select_clause
    : 'select' expression
    ;

group_clause
    : 'group' expression 'by' expression
    ;

query_continuation
    : 'into' identifier query_body
    ;
```

Une expression de requête commence par une `from` clause et se termine par une `select` `group` clause ou. La `from` clause initiale peut être suivie par zéro, une ou plusieurs `from` `let` `where` `join` `orderby` clauses,, ou. Chaque `from` clause est un générateur qui introduit une **variable de portée***, qui s'étend sur les éléments d'une *séquence* _ *. Chaque `let` clause introduit une variable de portée qui représente une valeur calculée au moyen de variables de plage précédentes. Chaque `where` clause est un filtre qui exclut des éléments du résultat. Chaque `join` clause compare les clés spécifiées de la séquence source avec les clés d'une autre séquence, ce qui génère des paires correspondantes. Chaque `orderby` clause réorganise les éléments en fonction des critères spécifiés. La `select` clause final ou `group` spécifie la forme du résultat en termes de variables de plage. Enfin, une `into` clause peut être utilisée pour « épisser » des requêtes en traitant les résultats d'une requête sous la forme d'un générateur dans une requête suivante.

Ambiguïtés dans les expressions de requête

Les expressions de requête contiennent un certain nombre de « mots-clés contextuels », c'est-à-dire des identificateurs qui ont une signification particulière dans un contexte donné. En particulier, il s'agit de,, `from` `where` `join` , `on` , `equals` , `into` , `let` , `orderby` , `ascending` , `descending` , `select` `group` et `by` . Afin d'éviter les ambiguïtés dans les expressions de requête provoquées par l'utilisation mixte de ces identificateurs en tant que Mots clés ou noms simples, ces identificateurs sont considérés comme des mots clés lorsqu'ils se trouvent n'importe où dans une expression de requête.

À cet effet, une expression de requête est une expression qui commence par « `from` `identifier` » suivi de n'importe quel jeton, à l'exception de « `;` », « `=` » ou « `,` ».

Pour pouvoir utiliser ces mots comme identificateurs dans une expression de requête, ils peuvent être précédés de « `@` » (identificateurs).

Traduction d'expression de requête

Le langage C# ne spécifie pas la sémantique d'exécution des expressions de requête. Au lieu de cela, les expressions de requête sont traduites en appels de méthodes qui adhèrent au *modèle d'expression de requête* (le modèle d'expression de requête). Plus précisément, les expressions de requête sont traduites en appels de méthodes nommées `Where` , `Select` , `SelectMany` , `Join` , `GroupJoin` , `OrderBy` , `OrderByDescending` , `ThenBy` ,, `ThenByDescending` `GroupBy` et `Cast` . Ces méthodes sont supposées avoir des signatures et des types de résultats particuliers, comme décrit dans le modèle d'expression de requête. Ces méthodes peuvent être des méthodes d'instance de l'objet interrogé ou des méthodes d'extension qui sont externes à l'objet, et elles implémentent l'exécution réelle de la requête.

La traduction des expressions de requête en appels de méthode est un mappage syntaxique qui se produit avant l'exécution d'une quelconque liaison de type ou de surcharge. La traduction est garantie pour être syntaxiquement correcte, mais il n'est pas garanti qu'elle produise du code C# correct sémantiquement. À la suite de la traduction des expressions de requête, les appels de méthode résultants sont traités comme des appels de méthode normaux, ce qui peut à son tour dévoiler les erreurs, par exemple si les méthodes n'existent pas, si les arguments ont des types incorrects, ou si les méthodes sont génériques et que l'inférence de type échoue.

Une expression de requête est traitée en appliquant à plusieurs reprises les traductions suivantes jusqu'à ce qu'aucune réduction supplémentaire ne soit possible. Les traductions sont répertoriées dans l'ordre de l'application : chaque section suppose que les traductions dans les sections précédentes ont été effectuées de façon exhaustive, et une fois épuisées, une section n'est pas revisitée ultérieurement dans le traitement de la même expression de requête.

L'assignation à des variables de portée n'est pas autorisée dans les expressions de requête. Toutefois, une implémentation C# est autorisée à ne pas toujours appliquer cette restriction, car cela peut parfois ne pas être possible avec le schéma de traduction syntaxique présenté ici.

Certaines traductions injectent des variables de portée avec des identificateurs transparents dénotés par `*` . Les propriétés spéciales des identificateurs transparents sont décrites plus en détail dans identificateurs transparents.

Clauses SELECT et GroupBy avec continuations

Expression de requête avec une continuation

```
C#
```

```
from ... into x ...
```

est traduit en

```
C#
```

```
from x in ( from ... ) ...
```

Les traductions dans les sections suivantes supposent que les requêtes n'ont pas de `into` continuations.

L'exemple

```
C#
```

```
from c in customers
group c by c.Country into g
select new { Country = g.Key, CustCount = g.Count() }
```

est traduit en

```
C#
```

```
from g in
    from c in customers
    group c by c.Country
select new { Country = g.Key, CustCount = g.Count() }
```

traduction finale de qui est

```
C#
```

```
customers.
GroupBy(c => c.Country).
Select(g => new { Country = g.Key, CustCount = g.Count() })
```

Types de variable de portée explicite

`from`Clause qui spécifie explicitement un type de variable de portée

```C#
from T x in e
```

est traduit en

```C#
from x in ( e ) . Cast < T > ( )
```

`join`Clause qui spécifie explicitement un type de variable de portée

```C#
join T x in e on k1 equals k2
```

est traduit en

```C#
join x in ( e ) . Cast < T > ( ) on k1 equals k2
```

Les traductions dans les sections suivantes supposent que les requêtes n'ont pas de types de variable de portée explicite.

L'exemple

```C#
from Customer c in customers
where c.City == "London"
select c
```

est traduit en

```C#
from c in customers.Cast<Customer>()
where c.City == "London"
select c
```

traduction finale de qui est

```
C#

customers.
Cast<Customer>().
Where(c => c.City == "London")
```

Les types de variable de portée explicite sont utiles pour interroger des collections qui implémentent l'interface non générique `IEnumerable` , mais pas l' `IEnumerable<T>` interface générique. Dans l'exemple ci-dessus, ce serait le cas si `customers` était de type `ArrayList` .

Dégénérer des expressions de requête

Expression de requête de la forme

```
C#

from x in e select x
```

est traduit en

```
C#

( e ) . Select ( x => x )
```

L'exemple

```
C#

from c in customers
select c
```

est traduit en

```
C#

customers.Select(c => c)
```

Une expression de requête dégénérée est une expression qui sélectionne de façon triviale les éléments de la source. Une phase ultérieure de la traduction supprime les requêtes dégénérées introduites par d'autres étapes de traduction en les remplaçant par leur source. Toutefois, il est important de s'assurer que le résultat d'une expression de requête n'est jamais l'objet source lui-même, car cela révélerait le type et l'identité de la

source au client de la requête. Par conséquent, cette étape protège les requêtes dégénérées écrites directement dans le code source en appelant explicitement `Select` sur la source. Il s'agit ensuite des implémenteurs de `Select` et d'autres opérateurs de requête pour garantir que ces méthodes ne retournent jamais l'objet source lui-même.

Clauses from, Let, Where, Join et OrderBy

Une expression de requête avec une seconde `from` clause suivie d'une `select` clause

```
C#

from x1 in e1
from x2 in e2
select v
```

est traduit en

```
C#

( e1 ) . SelectMany( x1 => e2 , ( x1 , x2 ) => v )
```

Une expression de requête avec une seconde `from` clause suivie d'une autre chose qu'une `select` clause :

```
C#

from x1 in e1
from x2 in e2
...
```

est traduit en

```
C#

from * in ( e1 ) . SelectMany( x1 => e2 , ( x1 , x2 ) => new { x1 , x2 } )
...
```

Expression de requête avec une `let` clause

```
C#

from x in e
let y = f
...
```

est traduit en

```C#
from * in ( e ) . Select ( x => new { x , y = f } )
...
```

Expression de requête avec une `where` clause

```C#
from x in e
where f
...
```

est traduit en

```C#
from x in ( e ) . Where ( x => f )
...
```

Expression de requête avec une `join` clause sans `into` suivi d'une `select` clause

```C#
from x1 in e1
join x2 in e2 on k1 equals k2
select v
```

est traduit en

```C#
( e1 ) . Join( e2 , x1 => k1 , x2 => k2 , ( x1 , x2 ) => v )
```

Expression de requête avec une `join` clause sans suivi d'un `into` autre nom qu'une `select` clause

```C#
from x1 in e1
join x2 in e2 on k1 equals k2
...
```

est traduit en

```
C#
```

```
from * in ( e1 ) . Join( e2 , x1 => k1 , x2 => k2 , ( x1 , x2 ) => new { x1
, x2 })
...
```

Expression de requête avec une `join` clause avec une `into` clause suivie d'une `select` clause

```
C#
```

```
from x1 in e1
join x2 in e2 on k1 equals k2 into g
select v
```

est traduit en

```
C#
```

```
( e1 ) . GroupJoin( e2 , x1 => k1 , x2 => k2 , ( x1 , g ) => v )
```

Une expression de requête avec une `join` clause avec un `into` suivi d'un autre nom qu'une `select` clause

```
C#
```

```
from x1 in e1
join x2 in e2 on k1 equals k2 into g
...
```

est traduit en

```
C#
```

```
from * in ( e1 ) . GroupJoin( e2 , x1 => k1 , x2 => k2 , ( x1 , g ) => new {
x1 , g })
...
```

Expression de requête avec une `orderby` clause

```
C#
```

```
from x in e
orderby k1 , k2 , ..., kn
...
```

est traduit en

C#

```
from x in ( e ) .
OrderBy ( x => k1 ) .
ThenBy ( x => k2 ) .
... .
ThenBy ( x => kn )
...
```

Si une clause de classement spécifie un `descending` indicateur de direction, un appel de `OrderByDescending` ou `ThenByDescending` est produit à la place.

Les traductions suivantes supposent qu'il n'y a pas `let` `where` `join` `orderby` de clauses, ou, et pas plus que la `from` clause initiale de chaque expression de requête.

L'exemple

C#

```
from c in customers
from o in c.Orders
select new { c.Name, o.OrderID, o.Total }
```

est traduit en

C#

```
customers.
SelectMany(c => c.Orders,
     (c,o) => new { c.Name, o.OrderID, o.Total }
)
```

L'exemple

C#

```
from c in customers
from o in c.Orders
orderby o.Total descending
select new { c.Name, o.OrderID, o.Total }
```

est traduit en

```C#
from * in customers.
    SelectMany(c => c.Orders, (c,o) => new { c, o })
orderby o.Total descending
select new { c.Name, o.OrderID, o.Total }
```

traduction finale de qui est

```C#
customers.
SelectMany(c => c.Orders, (c,o) => new { c, o }).
OrderByDescending(x => x.o.Total).
Select(x => new { x.c.Name, x.o.OrderID, x.o.Total })
```

où `x` est un identificateur généré par le compilateur qui est autrement invisible et inaccessible.

L'exemple

```C#
from o in orders
let t = o.Details.Sum(d => d.UnitPrice * d.Quantity)
where t >= 1000
select new { o.OrderID, Total = t }
```

est traduit en

```C#
from * in orders.
    Select(o => new { o, t = o.Details.Sum(d => d.UnitPrice * d.Quantity) })
where t >= 1000
select new { o.OrderID, Total = t }
```

traduction finale de qui est

```C#
orders.
Select(o => new { o, t = o.Details.Sum(d => d.UnitPrice * d.Quantity) }).
```

```
Where(x => x.t >= 1000).
Select(x => new { x.o.OrderID, Total = x.t })
```

où `x` est un identificateur généré par le compilateur qui est autrement invisible et inaccessible.

L'exemple

C#

```
from c in customers
join o in orders on c.CustomerID equals o.CustomerID
select new { c.Name, o.OrderDate, o.Total }
```

est traduit en

C#

```
customers.Join(orders, c => c.CustomerID, o => o.CustomerID,
    (c, o) => new { c.Name, o.OrderDate, o.Total })
```

L'exemple

C#

```
from c in customers
join o in orders on c.CustomerID equals o.CustomerID into co
let n = co.Count()
where n >= 10
select new { c.Name, OrderCount = n }
```

est traduit en

C#

```
from * in customers.
    GroupJoin(orders, c => c.CustomerID, o => o.CustomerID,
        (c, co) => new { c, co })
let n = co.Count()
where n >= 10
select new { c.Name, OrderCount = n }
```

traduction finale de qui est

C#

```
customers.
GroupJoin(orders, c => c.CustomerID, o => o.CustomerID,
    (c, co) => new { c, co }).
Select(x => new { x, n = x.co.Count() }).
Where(y => y.n >= 10).
Select(y => new { y.x.c.Name, OrderCount = y.n)
```

où `x` et `y` sont des identificateurs générés par le compilateur qui, sinon, sont inaccessibles et inaccessibles.

L'exemple

C#

```
from o in orders
orderby o.Customer.Name, o.Total descending
select o
```

a la traduction finale

C#

```
orders.
OrderBy(o => o.Customer.Name).
ThenByDescending(o => o.Total)
```

Clauses SELECT

Expression de requête de la forme

C#

```
from x in e select v
```

est traduit en

C#

```
( e ) . Select ( x => v )
```

sauf lorsque v est l'identificateur x, la traduction est simplement

C#

```
( e )
```

Par exemple

C#

```
from c in customers.Where(c => c.City == "London")
select c
```

est simplement traduit en

C#

```
customers.Where(c => c.City == "London")
```

Clauses GROUPBY

Expression de requête de la forme

C#

```
from x in e group v by k
```

est traduit en

C#

```
( e ) . GroupBy ( x => k , x => v )
```

sauf lorsque v est l'identificateur x, la traduction est

C#

```
( e ) . GroupBy ( x => k )
```

L'exemple

C#

```
from c in customers
group c.Name by c.Country
```

est traduit en

```
C#

customers.
GroupBy(c => c.Country, c => c.Name)
```

Identificateurs transparents

Certaines traductions injectent des variables de portée avec * des **identificateurs transparents** _ dénotés par `_` . Les identificateurs transparents ne sont pas une fonctionnalité de langage appropriée. ils existent uniquement comme étape intermédiaire dans le processus de traduction d'expression de requête.

Quand une traduction de requête injecte un identificateur transparent, d'autres étapes de traduction propagent l'identificateur transparent dans des fonctions anonymes et des initialiseurs d'objets anonymes. Dans ces contextes, les identificateurs transparents ont le comportement suivant :

- Lorsqu'un identificateur transparent se produit en tant que paramètre dans une fonction anonyme, les membres du type anonyme associé sont automatiquement dans la portée dans le corps de la fonction anonyme.
- Lorsqu'un membre avec un identificateur transparent est dans la portée, les membres de ce membre sont également dans la portée.
- Lorsqu'un identificateur transparent se produit comme déclarateur de membre dans un initialiseur d'objet anonyme, il introduit un membre avec un identificateur transparent.
- Dans les étapes de traduction décrites ci-dessus, les identificateurs transparents sont toujours introduits en même temps que les types anonymes, dans le but de capturer plusieurs variables de portée en tant que membres d'un seul objet. Une implémentation de C# est autorisée à utiliser un autre mécanisme que les types anonymes pour regrouper plusieurs variables de portée. Les exemples de traduction suivants supposent que les types anonymes sont utilisés et montrent comment les identificateurs transparents peuvent être traduits.

L'exemple

```
C#

from c in customers
from o in c.Orders
orderby o.Total descending
select new { c.Name, o.Total }
```

est traduit en

```C#
from * in customers.
    SelectMany(c => c.Orders, (c,o) => new { c, o })
orderby o.Total descending
select new { c.Name, o.Total }
```

qui est ensuite traduite en

```C#
customers.
SelectMany(c => c.Orders, (c,o) => new { c, o }).
OrderByDescending(* => o.Total).
Select(* => new { c.Name, o.Total })
```

qui, lorsque les identificateurs transparents sont effacés, est équivalent à

```C#
customers.
SelectMany(c => c.Orders, (c,o) => new { c, o }).
OrderByDescending(x => x.o.Total).
Select(x => new { x.c.Name, x.o.Total })
```

où `x` est un identificateur généré par le compilateur qui est autrement invisible et inaccessible.

L'exemple

```C#
from c in customers
join o in orders on c.CustomerID equals o.CustomerID
join d in details on o.OrderID equals d.OrderID
join p in products on d.ProductID equals p.ProductID
select new { c.Name, o.OrderDate, p.ProductName }
```

est traduit en

```C#
from * in customers.
    Join(orders, c => c.CustomerID, o => o.CustomerID,
        (c, o) => new { c, o })
```

```
join d in details on o.OrderID equals d.OrderID
join p in products on d.ProductID equals p.ProductID
select new { c.Name, o.OrderDate, p.ProductName }
```

ce qui est encore plus réduit à

C#

```
customers.
Join(orders, c => c.CustomerID, o => o.CustomerID, (c, o) => new { c, o }).
Join(details, * => o.OrderID, d => d.OrderID, (*, d) => new { *, d }).
Join(products, * => d.ProductID, p => p.ProductID, (*, p) => new { *, p }).
Select(* => new { c.Name, o.OrderDate, p.ProductName })
```

traduction finale de qui est

C#

```
customers.
Join(orders, c => c.CustomerID, o => o.CustomerID,
    (c, o) => new { c, o }).
Join(details, x => x.o.OrderID, d => d.OrderID,
    (x, d) => new { x, d }).
Join(products, y => y.d.ProductID, p => p.ProductID,
    (y, p) => new { y, p }).
Select(z => new { z.y.x.c.Name, z.y.x.o.OrderDate, z.p.ProductName })
```

où `x` , `y` et `z` sont des identificateurs générés par le compilateur qui, sinon, sont inaccessibles et inaccessibles.

Modèle d'expression de requête

Le ***modèle d'expression de requête*** établit un modèle de méthodes que les types peuvent implémenter pour prendre en charge les expressions de requête. Étant donné que les expressions de requête sont traduites en appels de méthode au moyen d'un mappage syntaxique, les types ont une flexibilité considérable dans la manière dont ils implémentent le modèle d'expression de requête. Par exemple, les méthodes du modèle peuvent être implémentées en tant que méthodes d'instance ou en tant que méthodes d'extension, car les deux ont la même syntaxe d'appel, et les méthodes peuvent demander des délégués ou des arborescences d'expressions, car les fonctions anonymes sont convertibles dans les deux.

La forme recommandée d'un type générique `C<T>` qui prend en charge le modèle d'expression de requête est indiquée ci-dessous. Un type générique est utilisé pour

illustrer les relations appropriées entre les types de paramètres et de résultats, mais il est possible d'implémenter le modèle pour les types non génériques également.

C#

```
delegate R Func<T1,R>(T1 arg1);

delegate R Func<T1,T2,R>(T1 arg1, T2 arg2);

class C
{
    public C<T> Cast<T>();
}

class C<T> : C
{
    public C<T> Where(Func<T,bool> predicate);

    public C<U> Select<U>(Func<T,U> selector);

    public C<V> SelectMany<U,V>(Func<T,C<U>> selector,
        Func<T,U,V> resultSelector);

    public C<V> Join<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,
        Func<U,K> innerKeySelector, Func<T,U,V> resultSelector);

    public C<V> GroupJoin<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,
        Func<U,K> innerKeySelector, Func<T,C<U>,V> resultSelector);

    public O<T> OrderBy<K>(Func<T,K> keySelector);

    public O<T> OrderByDescending<K>(Func<T,K> keySelector);

    public C<G<K,T>> GroupBy<K>(Func<T,K> keySelector);

    public C<G<K,E>> GroupBy<K,E>(Func<T,K> keySelector,
        Func<T,E> elementSelector);
}

class O<T> : C<T>
{
    public O<T> ThenBy<K>(Func<T,K> keySelector);

    public O<T> ThenByDescending<K>(Func<T,K> keySelector);
}

class G<K,T> : C<T>
{
    public K Key { get; }
}
```

Les méthodes ci-dessus utilisent les types délégués génériques `Func<T1,R>` et `Func<T1,T2,R>` , mais elles peuvent également avoir utilisé d'autres types de délégué ou d'arborescence d'expression avec les mêmes relations dans les types de paramètres et de résultats.

Notez la relation recommandée entre `C<T>` et `O<T>` qui garantit que les `ThenBy` `ThenByDescending` méthodes et sont disponibles uniquement sur le résultat d'un `OrderBy` ou d'un `OrderByDescending` . Notez également la forme recommandée du résultat d' `GroupBy` une séquence de séquences, où chaque séquence interne a une `Key` propriété supplémentaire.

L' `System.Linq` espace de noms fournit une implémentation du modèle d'opérateur de requête pour tout type qui implémente l' `System.Collections.Generic.IEnumerable<T>` interface.

Opérateurs d'assignation

Les opérateurs d'assignation affectent une nouvelle valeur à une variable, une propriété, un événement ou un élément d'indexeur.

antlr

```
assignment
    : unary_expression assignment_operator expression
    ;

assignment_operator
    : '='
    | '+='
    | '-='
    | '*='
    | '/='
    | '%='
    | '&='
    | '|='
    | '^='
    | '<<='
    | right_shift_assignment
    ;
```

L'opérande gauche d'une assignation doit être une expression classifiée comme une variable, un accès à une propriété, un accès à un indexeur ou un accès à un événement.

L' `=` opérateur est appelé ***opérateur d'assignation simple***. Elle assigne la valeur de l'opérande droit à la variable, à la propriété ou à l'élément d'indexeur donné par

l'opérande gauche. L'opérande gauche de l'opérateur d'assignation simple ne peut pas être un accès à un événement (sauf s'il est décrit dans événements de type champ). L'opérateur d'assignation simple est décrit dans assignation simple.

Les opérateurs d'assignation autres que l' `=` opérateur sont appelés des ***opérateurs d'assignation composée***. Ces opérateurs effectuent l'opération indiquée sur les deux opérandes, puis assignent la valeur résultante à la variable, à la propriété ou à l'élément d'indexeur donné par l'opérande gauche. Les opérateurs d'assignation composée sont décrits dans assignation composée.

Les `+=` `-=` opérateurs et avec une expression d'accès à l'événement comme opérande gauche sont appelés des *opérateurs d'assignation d'événement*. Aucun autre opérateur d'assignation n'est valide avec un accès aux événements comme opérande de gauche. Les opérateurs d'assignation d'événement sont décrits dans assignation d'événement.

Les opérateurs d'assignation sont associatifs à droite, ce qui signifie que les opérations sont regroupées de droite à gauche. Par exemple, une expression de la forme `a = b = c` est évaluée comme `a = (b = c)` .

Assignation simple

L' `=` opérateur est appelé opérateur d'assignation simple.

Si l'opérande gauche d'une assignation simple est au format `E.P` ou `E[Ei]` où `E` a le type au moment de la compilation `dynamic` , l'assignation est liée dynamiquement (liaison dynamique). Dans ce cas, le type au moment de la compilation de l'expression d'assignation est `dynamic` , et la résolution décrite ci-dessous se produit au moment de l'exécution en fonction du type d'exécution de `E` .

Dans une assignation simple, l'opérande de droite doit être une expression implicitement convertible en type de l'opérande de gauche. L'opération assigne la valeur de l'opérande droit à la variable, à la propriété ou à l'élément d'indexeur donné par l'opérande gauche.

Le résultat d'une expression d'assignation simple est la valeur assignée à l'opérande de gauche. Le résultat a le même type que l'opérande de gauche et est toujours classifié comme une valeur.

Si l'opérande de gauche est un accès à une propriété ou un indexeur, la propriété ou l'indexeur doit avoir un `set` accesseur. Si ce n'est pas le cas, une erreur de temps de liaison se produit.

Le traitement au moment de l'exécution d'une assignation simple du formulaire `x = y` se compose des étapes suivantes :

- Si `x` est classé en tant que variable :
  - `x` est évalué pour produire la variable.
  - `y` est évalué et, si nécessaire, est converti en type de `x` via une conversion implicite (conversions implicites).
  - Si la variable fournie par `x` est un élément de tableau d'un *reference_type*, un contrôle au moment de l'exécution est effectué pour garantir que la valeur calculée pour `y` est compatible avec l'instance de tableau de qui `x` est un élément. La vérification s'effectue correctement si `y` a la valeur `null` , ou si une conversion de référence implicite (conversions de référence implicites) existe à partir du type réel de l'instance référencée par `y` vers le type d'élément réel de l'instance de tableau contenant `x` . Sinon, une exception `System.ArrayTypeMismatchException` est levée.
  - La valeur résultant de l'évaluation et de la conversion de `y` est stockée dans l'emplacement donné par l'évaluation de `x` .
- Si `x` est classifié comme un accès à une propriété ou un indexeur :
  - L'expression d'instance (si `x` n'est pas `static`) et la liste d'arguments (si `x` est un accès à un indexeur) associée à `x` sont évaluées, et les résultats sont utilisés dans l' `set` appel d'accesseur suivant.
  - `y` est évalué et, si nécessaire, est converti en type de `x` via une conversion implicite (conversions implicites).
  - L' `set` accesseur de `x` est appelé avec la valeur calculée pour `y` comme `value` argument.

Les règles de covariance de tableau (covariance de tableau) permettent à une valeur d'un type de tableau `A[]` d'être une référence à une instance d'un type de tableau `B[]` , à condition qu'il existe une conversion de référence implicite de `B` en `A` . En raison de ces règles, l'assignation à un élément de tableau d'un *reference_type* nécessite un contrôle à l'exécution pour garantir que la valeur assignée est compatible avec l'instance de tableau. Dans l'exemple

C#

```
string[] sa = new string[10];
object[] oa = sa;

oa[0] = null;               // Ok
oa[1] = "Hello";            // Ok
oa[2] = new ArrayList();    // ArrayTypeMismatchException
```

la dernière assignation provoque la `System.ArrayTypeMismatchException` levée d'une, car une instance de `ArrayList` ne peut pas être stockée dans un élément d'un `string[]` .

Lorsqu'une propriété ou un indexeur déclaré dans un *struct_type* est la cible d'une assignation, l'expression d'instance associée à l'accès à la propriété ou à l'indexeur doit être classée en tant que variable. Si l'expression d'instance est classée en tant que valeur, une erreur de liaison au moment de la liaison se produit. En raison de l' accès aux membres, la même règle s'applique également aux champs.

Compte tenu des déclarations :

C#

```
struct Point
{
    int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int X {
        get { return x; }
        set { x = value; }
    }

    public int Y {
        get { return y; }
        set { y = value; }
    }
}

struct Rectangle
{
    Point a, b;

    public Rectangle(Point a, Point b) {
        this.a = a;
        this.b = b;
    }

    public Point A {
        get { return a; }
        set { a = value; }
    }

    public Point B {
        get { return b; }
        set { b = value; }
    }
}
```

dans l'exemple

```
C#

Point p = new Point();
p.X = 100;
p.Y = 100;
Rectangle r = new Rectangle();
r.A = new Point(10, 10);
r.B = p;
```

les assignations à `p.X` , `p.Y` , `r.A` et `r.B` sont autorisées, car `p` et `r` sont des variables. Toutefois, dans l'exemple

```
C#

Rectangle r = new Rectangle();
r.A.X = 10;
r.A.Y = 10;
r.B.X = 100;
r.B.Y = 100;
```

les attributions ne sont pas valides, car `r.A` et ne `r.B` sont pas des variables.

Assignation composée

Si l'opérande gauche d'une assignation composée est au format `E.P` ou `E[Ei]` où `E` a le type au moment de la compilation `dynamic` , l'assignation est liée dynamiquement (liaison dynamique). Dans ce cas, le type au moment de la compilation de l'expression d'assignation est `dynamic` , et la résolution décrite ci-dessous se produit au moment de l'exécution en fonction du type d'exécution de `E` .

Une opération du formulaire `x op= y` est traitée en appliquant la résolution de surcharge d'opérateur binaire (résolution de surcharge d'opérateur binaire) comme si l'opération était écrite `x op y` . Ainsi,

- Si le type de retour de l'opérateur sélectionné est implicitement convertible dans le type de `x` , l'opération est évaluée comme `x = x op y` , sauf qu' `x` elle est évaluée une seule fois.
- Sinon, si l'opérateur sélectionné est un opérateur prédéfini, si le type de retour de l'opérateur sélectionné est explicitement convertible en type de `x` , et si `y` est implicitement convertible en type de `x` ou si l'opérateur est un opérateur de

décalage, l'opération est évaluée comme `x = (T)(x op y)`, où `T` est le type de `x`, sauf qu' `x` elle est évaluée une seule fois.
- Dans le cas contraire, l'assignation composée n'est pas valide et une erreur de liaison s'est produite.

Le terme « évalué une seule fois » signifie que, dans l'évaluation de `x op y`, les résultats de toutes les expressions constitutives de `x` sont temporairement enregistrés et réutilisés lors de l'exécution de l'assignation à `x`. Par exemple, dans l'assignation `A()[B()] += C()`, où `A` est une méthode qui retourne `int[]`, et `B` et sont des méthodes qui `C` retournent `int`, les méthodes sont appelées une seule fois, dans l'ordre `A`, `B`, `C`.

Lorsque l'opérande gauche d'une assignation composée est un accès à une propriété ou un indexeur, la propriété ou l'indexeur doit avoir à la fois un `get` accesseur et un `set` accesseur. Si ce n'est pas le cas, une erreur de temps de liaison se produit.

La deuxième règle ci-dessus permet `x op= y` d'évaluer comme `x = (T)(x op y)` dans certains contextes. La règle existe de sorte que les opérateurs prédéfinis peuvent être utilisés comme opérateurs composés lorsque l'opérande de gauche est de type,,, `sbyte` `byte` `short` `ushort` ou `char`. Même si les deux arguments sont de l'un de ces types, les opérateurs prédéfinis produisent un résultat de type `int`, comme décrit dans promotions numériques binaires. Ainsi, sans cast, il n'est pas possible d'assigner le résultat à l'opérande gauche.

L'effet intuitif de la règle pour les opérateurs prédéfinis est simplement `x op= y` autorisé si `x op y` et `x = y` sont autorisés. Dans l'exemple

C#

```
byte b = 0;
char ch = '\0';
int i = 0;

b += 1;             // Ok
b += 1000;          // Error, b = 1000 not permitted
b += i;             // Error, b = i not permitted
b += (byte)i;       // Ok

ch += 1;            // Error, ch = 1 not permitted
ch += (char)1;      // Ok
```

la raison la plus intuitive pour chaque erreur est qu'une assignation simple correspondante aurait également été une erreur.

Cela signifie également que les opérations d'assignation composée prennent en charge les opérations levées. Dans l'exemple

```C#
int? i = 0;
i += 1;             // Ok
```

l'opérateur levé `+(int?,int?)` est utilisé.

Assignation d'événement

Si l'opérande gauche d'un `+=` `-=` opérateur or est classifié comme un accès aux événements, l'expression est évaluée comme suit :

- L'expression d'instance, le cas échéant, de l'accès à l'événement est évaluée.
- L'opérande droit de l' `+=` `-=` opérateur or est évalué et, si nécessaire, est converti en type de l'opérande gauche via une conversion implicite (conversions implicites).
- Un accesseur d'événement de l'événement est appelé, avec la liste d'arguments composée de l'opérande de droite, après l'évaluation et, si nécessaire, la conversion. Si l'opérateur était `+=` , l' `add` accesseur est appelé ; si l'opérateur était `-=` , l' `remove` accesseur est appelé.

Une expression d'assignation d'événement ne produit pas de valeur. Ainsi, une expression d'assignation d'événement est valide uniquement dans le contexte d'une *statement_expression* (instructions d'expression).

Expression

Une *expression* est soit une *non_assignment_expression* , soit une *assignation*.

```antlr
expression
    : non_assignment_expression
    | assignment
    ;

non_assignment_expression
    : conditional_expression
    | lambda_expression
    | query_expression
    ;
```

Expressions de constantes

Une *constant_expression* est une expression qui peut être complètement évaluée au moment de la compilation.

antlr

```
constant_expression
    : expression
    ;
```

Une expression constante doit être le `null` littéral ou une valeur avec l'un des types suivants : `sbyte` , `byte` , `short` , `ushort` , `int` , `uint` , `long` , `ulong` , `char` , `float` , `double` , `decimal` , `bool` , `object` , `string` ou tout type énumération. Seules les constructions suivantes sont autorisées dans les expressions constantes :

- Littéraux (y compris le `null` littéral).
- Références aux `const` membres de types classe et struct.
- Références aux membres de types énumération.
- Références à des `const` paramètres ou à des variables locales
- Sous-expressions entre parenthèses, qui sont elles-mêmes des expressions constantes.
- Expressions de cast, à condition que le type cible soit l'un des types indiqués ci-dessus.
- `checked``unchecked`expressions et
- Expressions de valeur par défaut
- Expressions Nameof
- Opérateurs unaires prédéfinis, `+` `-` , `!` et `~` .
- Les opérateurs prédéfinis,,,,,, `+` `-` ,,,, `*` `/` `%` `<<` `>>` `&` `|` `^` `&&` , `||` , `==` , `!=` , `<` , `>` , `<=` et `>=` binaires, à condition que chaque opérande soit d'un type indiqué ci-dessus.
- `?:`Opérateur conditionnel.

Les conversions suivantes sont autorisées dans les expressions constantes :

- Conversions d'identité
- Conversions numériques
- Conversions d'énumération
- Conversions d'expressions constantes
- Conversions de référence implicites et explicites, à condition que la source des conversions soit une expression constante qui prend la valeur null.

Les autres conversions, y compris les conversions boxing, unboxing et implicites des valeurs non null, ne sont pas autorisées dans les expressions constantes. Par exemple :

```
C#
```

```
class C {
    const object i = 5;         // error: boxing conversion not permitted
    const object str = "hello"; // error: implicit reference conversion
}
```

l'initialisation de i est une erreur, car une conversion boxing est requise. L'initialisation de str est une erreur, car une conversion de référence implicite à partir d'une valeur non null est requise.

Chaque fois qu'une expression répond aux spécifications mentionnées ci-dessus, l'expression est évaluée au moment de la compilation. Cela est vrai même si l'expression est une sous-expression d'une expression plus grande qui contient des constructions non constantes.

L'évaluation au moment de la compilation des expressions constantes utilise les mêmes règles que l'évaluation au moment de l'exécution des expressions non constantes, à la différence près que lorsque l'évaluation au moment de l'exécution a levé une exception, l'évaluation au moment de la compilation provoque une erreur au moment de la compilation.

À moins qu'une expression constante ne soit explicitement placée dans un `unchecked` contexte, les débordements qui se produisent dans les opérations arithmétiques de type intégral et les conversions pendant l'évaluation de l'expression au moment de la compilation provoquent toujours des erreurs au moment de la compilation (expressions constantes).

Les expressions constantes se trouvent dans les contextes listés ci-dessous. Dans ces contextes, une erreur de compilation se produit si une expression ne peut pas être complètement évaluée au moment de la compilation.

- Déclarations de constantes (constantes).
- Déclarations de membre d'énumération (membres enum).
- Arguments par défaut des listes de paramètres formels (paramètres de méthode)
- `case` étiquettes d'une `switch` instruction (instruction switch).
- `goto case` instructions (instruction goto).
- Longueurs de dimension dans une expression de création de tableau (expressions de création de tableau) qui comprend un initialiseur.
- Attributs (attributs).

Une conversion d'expression constante implicite (conversions d'expressions constantes implicites) permet de convertir une expression constante de type en `int` `sbyte` , `byte` ,, `short` `ushort` , `uint` ou `ulong` , à condition que la valeur de l'expression constante soit comprise dans la plage du type de destination.

Expressions booléennes

Une *Boolean_expression* est une expression qui produit un résultat de type `bool` , soit directement, soit par le biais de l'application de `operator true` dans certains contextes, comme spécifié dans l'exemple suivant.

antlr

```
boolean_expression
    : expression
    ;
```

L'expression conditionnelle de contrôle d' *un if_statement* (instruction if), *while_statement* (l'instruction while) , *do_statement* (instruction do) ou *for_Statement* (l'instruction for) est un *Boolean_expression*. L'expression conditionnelle de contrôle de l' `?:` opérateur (opérateur conditionnel) suit les mêmes règles qu'un *Boolean_expression*, mais pour des raisons de priorité d'opérateur est classée comme *conditional_or_expression*.

Une *Boolean_expression* `E` est requise pour pouvoir produire une valeur de type `bool` , comme suit :

- Si `E` est implicitement convertible en `bool` au moment de l'exécution, la conversion implicite est appliquée.
- Dans le cas contraire, la résolution de surcharge d'opérateur unaire (résolution de surcharge d'opérateur unaire) est utilisée pour rechercher une implémentation optimale unique de l'opérateur `true` sur `E` , et cette implémentation est appliquée au moment de l'exécution.
- Si aucun opérateur de ce type n'est trouvé, une erreur au moment de la liaison se produit.

Le type `DBBool` struct du type booléen de base de données fournit un exemple de type qui implémente `operator true` et `operator false` .

Instructions

Article • 13/05/2021

C# fournit une série d'instructions. La plupart de ces instructions seront familières aux développeurs qui ont programmé en C et C++.

antlr

```
statement
    : labeled_statement
    | declaration_statement
    | embedded_statement
    ;

embedded_statement
    : block
    | empty_statement
    | expression_statement
    | selection_statement
    | iteration_statement
    | jump_statement
    | try_statement
    | checked_statement
    | unchecked_statement
    | lock_statement
    | using_statement
    | yield_statement
    | embedded_statement_unsafe
    ;
```

Le *embedded_statement* non terminal est utilisé pour les instructions qui s'affichent dans d'autres instructions. L'utilisation de *embedded_statement* plutôt que l' *instruction* exclut l'utilisation d'instructions de déclaration et d'instructions étiquetées dans ces contextes. L'exemple

C#

```
void F(bool b) {
    if (b)
        int i = 44;
}
```

génère une erreur au moment de la compilation, car une `if` instruction requiert un *embedded_statement* plutôt qu'une *instruction* pour sa branche if. Si ce code est autorisé, la variable `i` est déclarée, mais elle n'a jamais été utilisée. Notez, toutefois, qu'en plaçant `i` la déclaration de dans un bloc, l'exemple est valide.

Points de terminaison et accessibilité

Chaque instruction a un ***point de terminaison***. En termes intuitifs, le point de terminaison d'une instruction correspond à l'emplacement qui suit immédiatement l'instruction. Les règles d'exécution des instructions composites (instructions qui contiennent des instructions incorporées) spécifient l'action qui est effectuée lorsque le contrôle atteint le point de terminaison d'une instruction incorporée. Par exemple, lorsque le contrôle atteint le point de terminaison d'une instruction dans un bloc, le contrôle est transféré à l'instruction suivante dans le bloc.

Si une instruction peut être atteinte par l'exécution, l'instruction est dite ***accessible** _. À l'inverse, s'il n'est pas possible qu'une instruction soit exécutée, l'instruction est dite _ *inaccessible* *.

Dans l'exemple

```
C#

void F() {
    Console.WriteLine("reachable");
    goto Label;
    Console.WriteLine("unreachable");
    Label:
    Console.WriteLine("reachable");
}
```

le deuxième appel de `Console.WriteLine` est inaccessible, car il n'est pas possible que l'instruction soit exécutée.

Un avertissement est signalé si le compilateur détermine qu'une instruction est inaccessible. Il ne s'agit pas d'une erreur spécifique pour qu'une instruction soit inaccessible.

Pour déterminer si une instruction ou un point de terminaison particulier est accessible, le compilateur effectue une analyse de workflow en fonction des règles d'accessibilité définies pour chaque instruction. L'analyse du workflow prend en compte les valeurs des expressions constantes (expressions constantes) qui contrôlent le comportement des instructions, mais les valeurs possibles des expressions non constantes ne sont pas prises en compte. En d'autres termes, à des fins d'analyse de workflow de contrôle, une expression non constante d'un type donné est considérée comme ayant n'importe quelle valeur possible de ce type.

Dans l'exemple

```
```

C#

```
void F() {
    const int i = 1;
    if (i == 2) Console.WriteLine("unreachable");
}
```

l'expression booléenne de l' `if` instruction est une expression constante, car les deux opérandes de l' `==` opérateur sont des constantes. Étant donné que l'expression constante est évaluée au moment de la compilation, produisant la valeur `false` , l' `Console.WriteLine` appel est considéré comme inaccessible. Toutefois, si `i` est remplacé par une variable locale

C#

```
void F() {
    int i = 1;
    if (i == 2) Console.WriteLine("reachable");
}
```

l' `Console.WriteLine` appel est considéré comme accessible, même si, en réalité, il ne sera jamais exécuté.

Le *bloc* d'un membre de fonction est toujours considéré comme accessible. En évaluant successivement les règles d'accessibilité de chaque instruction dans un bloc, vous pouvez déterminer l'accessibilité d'une instruction donnée.

Dans l'exemple

C#

```
void F(int x) {
    Console.WriteLine("start");
    if (x < 0) Console.WriteLine("negative");
}
```

l'accessibilité de la seconde `Console.WriteLine` est déterminée comme suit :

- La première `Console.WriteLine` instruction d'expression est accessible car le bloc de la `F` méthode est accessible.
- Le point de terminaison de la première `Console.WriteLine` instruction d'expression est accessible car cette instruction est accessible.
- L' `if` instruction est accessible parce que le point de terminaison de la première `Console.WriteLine` instruction d'expression est accessible.

- La deuxième `Console.WriteLine` instruction d'expression est accessible parce que l'expression booléenne de l' `if` instruction n'a pas la valeur de constante `false` .

Il existe deux situations dans lesquelles il s'agit d'une erreur de compilation pour que le point de terminaison d'une instruction soit accessible :

- Étant donné que l' `switch` instruction ne permet pas à une section switch de passer à la section switch suivante, il s'agit d'une erreur de compilation pour le point de terminaison de la liste d'instructions d'une section switch à atteindre. Si cette erreur se produit, cela indique généralement qu'une `break` instruction est manquante.
- Il s'agit d'une erreur au moment de la compilation pour le point de terminaison du bloc d'un membre de fonction qui calcule une valeur à atteindre. Si cette erreur se produit, cela indique généralement qu'une `return` instruction est manquante.

Blocs

Un *bloc* autorise l'écriture de plusieurs instructions dans les contextes où une seule instruction est autorisée.

antlr

```
block
    : '{' statement_list? '}'
    ;
```

Un *bloc* se compose d'une *statement_list* facultative (listes d'instructions), placées entre accolades. Si la liste d'instructions est omise, le bloc est dit vide.

Un bloc peut contenir des instructions de déclaration (instructions de déclaration). La portée d'une variable locale ou d'une constante déclarée dans un bloc est le bloc.

Un bloc est exécuté comme suit :

- Si le bloc est vide, le contrôle est transféré vers le point de terminaison du bloc.
- Si le bloc n'est pas vide, le contrôle est transféré à la liste d'instructions. Quand et si le contrôle atteint le point de terminaison de la liste d'instructions, le contrôle est transféré vers le point de terminaison du bloc.

La liste d'instructions d'un bloc est accessible si le bloc lui-même est accessible.

Le point de terminaison d'un bloc est accessible si le bloc est vide ou si le point de terminaison de la liste d'instructions est accessible.

Un *bloc* qui contient une ou plusieurs `yield` instructions (l'instruction yield) est appelé un bloc itérateur. Les blocs itérateurs sont utilisés pour implémenter des membres de fonction en tant qu'itérateurs (itérateurs). Certaines restrictions supplémentaires s'appliquent aux blocs Iterator :

- Il s'agit d'une erreur de compilation pour qu'une `return` instruction apparaisse dans un bloc itérateur (`yield return` les instructions sont autorisées).
- Il s'agit d'une erreur au moment de la compilation pour qu'un bloc itérateur contienne un contexte non sécurisé (contextes non sécurisés). Un bloc itérateur définit toujours un contexte sécurisé, même quand sa déclaration est imbriquée dans un contexte unsafe.

Listes d'instructions

Une **instruction * List** contient une ou plusieurs instructions écrites en séquence. Les listes d'instructions se trouvent dans _block * s (blocs) et dans *switch_block* s (instruction switch).

antlr

```
statement_list
    : statement+
    ;
```

Une liste d'instructions est exécutée en transférant le contrôle à la première instruction. Quand et si le contrôle atteint le point de terminaison d'une instruction, le contrôle est transféré à l'instruction suivante. Quand et si le contrôle atteint le point de terminaison de la dernière instruction, le contrôle est transféré vers le point de terminaison de la liste d'instructions.

Une instruction dans une liste d'instructions est accessible si au moins l'une des conditions suivantes est vraie :

- L'instruction est la première instruction et la liste d'instructions elle-même est accessible.
- Le point de terminaison de l'instruction précédente est accessible.
- L'instruction est une instruction étiquetée et l'étiquette est référencée par une `goto` instruction accessible.

Le point de terminaison d'une liste d'instructions est accessible si le point de terminaison de la dernière instruction de la liste est accessible.

Instruction vide

Un *empty_statement* ne fait rien.

antlr

```
empty_statement
    : ';'
    ;
```

Une instruction vide est utilisée lorsqu'il n'y a aucune opération à effectuer dans un contexte où une instruction est requise.

L'exécution d'une instruction vide transfère simplement le contrôle au point de terminaison de l'instruction. Ainsi, le point de terminaison d'une instruction vide est accessible si l'instruction vide est accessible.

Une instruction vide peut être utilisée lors de l'écriture d'une `while` instruction avec un corps NULL :

C#

```
bool ProcessMessage() {...}

void ProcessMessages() {
    while (ProcessMessage())
        ;
}
```

En outre, une instruction vide peut être utilisée pour déclarer une étiquette juste avant la fermeture « `}` » d'un bloc :

C#

```
void F() {
    ...
    if (done) goto exit;
    ...
    exit: ;
}
```

Instructions étiquetées

Une *labeled_statement* permet à une instruction d'être précédée d'une étiquette. Les instructions étiquetées sont autorisées dans des blocs, mais ne sont pas autorisées en

tant qu'instructions incorporées.

```antlr
labeled_statement
    : identifier ':' statement
    ;
```

Une instruction étiquetée déclare une étiquette portant le nom donné par l'*identificateur*. L'étendue d'une étiquette est le bloc entier dans lequel l'étiquette est déclarée, y compris les blocs imbriqués. Il s'agit d'une erreur au moment de la compilation pour deux étiquettes portant le même nom pour lesquelles des étendues se chevauchent.

Une étiquette peut être référencée à partir d' `goto` instructions (l'instruction goto) dans la portée de l'étiquette. Cela signifie que `goto` les instructions peuvent transférer le contrôle dans les blocs et hors des blocs, mais jamais dans des blocs.

Les étiquettes ont leur propre espace de déclaration et n'interfèrent pas avec d'autres identificateurs. L'exemple

```C#
int F(int x) {
    if (x >= 0) goto x;
    x = -x;
    x: return x;
}
```

est valide et utilise le nom `x` en tant que paramètre et étiquette.

L'exécution d'une instruction étiquetée correspond exactement à l'exécution de l'instruction qui suit l'étiquette.

Outre l'accessibilité fournie par le workflow normal de contrôle, une instruction étiquetée est accessible si l'étiquette est référencée par une `goto` instruction accessible. (Exception : si une `goto` instruction se trouve dans un `try` qui comprend un `finally` bloc et que l'instruction étiquetée est en dehors de la `try` , et que le point de terminaison du `finally` bloc est inaccessible, l'instruction étiquetée n'est pas accessible à partir de cette `goto` instruction.)

Instructions de déclaration

Une *declaration_statement* déclare une constante ou une variable locale. Les instructions de déclaration sont autorisées dans des blocs, mais ne sont pas autorisées en tant qu'instructions incorporées.

antlr

```
declaration_statement
    : local_variable_declaration ';'
    | local_constant_declaration ';'
    ;
```

Déclarations de variables locales

Une *local_variable_declaration* déclare une ou plusieurs variables locales.

antlr

```
local_variable_declaration
    : local_variable_type local_variable_declarators
    ;

local_variable_type
    : type
    | 'var'
    ;

local_variable_declarators
    : local_variable_declarator
    | local_variable_declarators ',' local_variable_declarator
    ;

local_variable_declarator
    : identifier
    | identifier '=' local_variable_initializer
    ;

local_variable_initializer
    : expression
    | array_initializer
    | local_variable_initializer_unsafe
    ;
```

La *local_variable_type* d'une *local_variable_declaration* spécifie directement le type des variables introduites par la déclaration, ou indique avec l'identificateur `var` que le type doit être déduit en fonction d'un initialiseur. Le type est suivi d'une liste de *local_variable_declarator* s, chacun d'entre eux introduisant une nouvelle variable. Un *local_variable_declarator* se compose d'un *identificateur* qui nomme la variable,

éventuellement suivie d'un jeton « `=` » et d'un *local_variable_initializer* qui donne la valeur initiale de la variable.

Dans le contexte d'une déclaration de variable locale, l'identificateur var agit comme un mot clé contextuel (Mots clés). Lorsque le *local_variable_type* est spécifié comme `var` et qu'aucun type nommé `var` n'est dans la portée, la déclaration est une ***déclaration de variable locale implicitement typée***, dont le type est déduit du type de l'expression d'initialiseur associée. Les déclarations de variables locales implicitement typées sont soumises aux restrictions suivantes :

- La *local_variable_declaration* ne peut pas inclure plusieurs *local_variable_declarator* s.
- Le *local_variable_declarator* doit inclure une *local_variable_initializer*.
- Le *local_variable_initializer* doit être une *expression*.
- L' *expression* d'initialiseur doit avoir un type au moment de la compilation.
- L' *expression* d'initialiseur ne peut pas faire référence à la variable déclarée elle-même

Voici des exemples de déclarations de variables locales implicitement typées incorrectes :

C#

```
var x;               // Error, no initializer to infer type from
var y = {1, 2, 3};   // Error, array initializer not permitted
var z = null;        // Error, null does not have a type
var u = x => x + 1;  // Error, anonymous functions do not have a type
var v = v++;         // Error, initializer cannot refer to variable itself
```

La valeur d'une variable locale est obtenue dans une expression à l'aide d'un *simple_name* (noms simples) et la valeur d'une variable locale est modifiée à l'aide d'une *assignation* (opérateurs d'affectation). Une variable locale doit être assignée définitivement (assignation définie) à chaque emplacement où sa valeur est obtenue.

La portée d'une variable locale déclarée dans un *local_variable_declaration* est le bloc dans lequel la déclaration se produit. Il est erroné de faire référence à une variable locale dans une position textuelle qui précède la *local_variable_declarator* de la variable locale. Dans l'étendue d'une variable locale, il s'agit d'une erreur de compilation pour déclarer une autre constante ou variable locale portant le même nom.

Une déclaration de variable locale qui déclare plusieurs variables est équivalente à plusieurs déclarations de variables uniques de même type. En outre, un initialiseur de variable dans une déclaration de variable locale correspond exactement à une instruction d'assignation qui est insérée immédiatement après la déclaration.

L'exemple

```C#
void F() {
    int x = 1, y, z = x * 2;
}
```

correspond exactement à

```C#
void F() {
    int x; x = 1;
    int y;
    int z; z = x * 2;
}
```

Dans une déclaration de variable locale implicitement typée, le type de la variable locale déclarée est considéré comme étant le même que le type de l'expression utilisée pour initialiser la variable. Par exemple :

```C#
var i = 5;
var s = "Hello";
var d = 1.0;
var numbers = new int[] {1, 2, 3};
var orders = new Dictionary<int,Order>();
```

Les déclarations de variables locales implicitement typées ci-dessus sont équivalentes précisément aux déclarations explicitement typées suivantes :

```C#
int i = 5;
string s = "Hello";
double d = 1.0;
int[] numbers = new int[] {1, 2, 3};
Dictionary<int,Order> orders = new Dictionary<int,Order>();
```

Déclarations de constantes locales

Une *local_constant_declaration* déclare une ou plusieurs constantes locales.

```
```

antlr

```
local_constant_declaration
    : 'const' type constant_declarators
    ;

constant_declarators
    : constant_declarator (',' constant_declarator)*
    ;

constant_declarator
    : identifier '=' constant_expression
    ;
```

Le *type* d'un *local_constant_declaration* spécifie le type des constantes introduites par la déclaration. Le type est suivi d'une liste de *constant_declarator* s, chacun d'entre eux introduisant une nouvelle constante. Un *constant_declarator* se compose d'un *identificateur* qui nomme la constante, suivi d'un `=` jeton «», suivi d'un *constant_expression* (expressions constantes) qui donne la valeur de la constante.

Le *type* et la *constant_expression* d'une déclaration de constante locale doivent suivre les mêmes règles que celles d'une déclaration de membre constante (constantes).

La valeur d'une constante locale est obtenue dans une expression à l'aide d'un *simple_name* (noms simples).

La portée d'une constante locale est le bloc dans lequel la déclaration se produit. La référence à une constante locale dans une position textuelle précédant son *constant_declarator* est une erreur. Dans l'étendue d'une constante locale, il s'agit d'une erreur de compilation pour déclarer une autre constante ou variable locale portant le même nom.

Une déclaration de constante locale qui déclare plusieurs constantes est équivalente à plusieurs déclarations de constantes uniques avec le même type.

Instructions d'expression

Une *expression_statement* évalue une expression donnée. La valeur calculée par l'expression, le cas échéant, est ignorée.

antlr

```
expression_statement
    : statement_expression ';'
    ;
```

```
statement_expression
    : invocation_expression
    | null_conditional_invocation_expression
    | object_creation_expression
    | assignment
    | post_increment_expression
    | post_decrement_expression
    | pre_increment_expression
    | pre_decrement_expression
    | await_expression
    ;
```

Toutes les expressions ne sont pas autorisées en tant qu'instructions. En particulier, les expressions telles que `x + y` et `x == 1` qui calculent simplement une valeur (qui sera ignorée) ne sont pas autorisées en tant qu'instructions.

L'exécution d'un *expression_statement* évalue l'expression contenue, puis transfère le contrôle au point de terminaison de l' *expression_statement*. Le point de terminaison d'un *expression_statement* est accessible si ce *expression_statement* est accessible.

Instructions de sélection

Instructions de sélection sélectionnez l'une des nombreuses instructions possibles pour l'exécution en fonction de la valeur d'une expression.

antlr

```
selection_statement
    : if_statement
    | switch_statement
    ;
```

L'instruction if

L' `if` instruction sélectionne une instruction à exécuter en fonction de la valeur d'une expression booléenne.

antlr

```
if_statement
    : 'if' '(' boolean_expression ')' embedded_statement
    | 'if' '(' boolean_expression ')' embedded_statement 'else'
embedded_statement
    ;
```

Un `else` composant est associé au précédent lexicalement le plus proche `if` qui est autorisé par la syntaxe. Ainsi, une `if` instruction de la forme

```
C#

if (x) if (y) F(); else G();
```

équivaut à :

```
C#

if (x) {
    if (y) {
        F();
    }
    else {
        G();
    }
}
```

Une `if` instruction est exécutée comme suit :

- Le *Boolean_expression* (expressions booléennes) est évalué.
- Si l'expression booléenne génère `true` , le contrôle est transféré à la première instruction incorporée. Quand et si le contrôle atteint le point de terminaison de cette instruction, le contrôle est transféré au point de terminaison de l' `if` instruction.
- Si l'expression booléenne génère `false` et si une `else` partie est présente, le contrôle est transféré à la deuxième instruction incorporée. Quand et si le contrôle atteint le point de terminaison de cette instruction, le contrôle est transféré au point de terminaison de l' `if` instruction.
- Si l'expression booléenne génère `false` et si une `else` partie n'est pas présente, le contrôle est transféré au point de terminaison de l' `if` instruction.

La première instruction incorporée d'une `if` instruction est accessible si l' `if` instruction est accessible et que l'expression booléenne n'a pas la valeur de constante `false` .

La deuxième instruction incorporée d'une `if` instruction, si elle est présente, est accessible si l' `if` instruction est accessible et si l'expression booléenne n'a pas la valeur de constante `true` .

Le point de terminaison d'une `if` instruction est accessible si le point de terminaison d'au moins une de ses instructions incorporées est accessible. En outre, le point de

terminaison d'une `if` instruction sans `else` partie est accessible si l' `if` instruction est accessible et que l'expression booléenne n'a pas la valeur de constante `true` .

Instruction switch

L'instruction switch sélectionne pour l'exécution une liste d'instructions ayant une étiquette Switch associée qui correspond à la valeur de l'expression Switch.

antlr

```
switch_statement
    : 'switch' '(' expression ')' switch_block
    ;

switch_block
    : '{' switch_section* '}'
    ;

switch_section
    : switch_label+ statement_list
    ;

switch_label
    : 'case' constant_expression ':'
    | 'default' ':'
    ;
```

Une *switch_Statement* se compose du mot clé `switch` , suivi d'une expression entre parenthèses (appelée expression de commutateur), suivie d'une *switch_block*. Le *switch_block* se compose de zéro, un ou plusieurs *switch_section*, entre accolades. Chaque *switch_section* se compose d'un ou plusieurs *switch_label* s suivis d'un *statement_list* (listes d'instructions).

Le ***type gouvernant*** d'une `switch` instruction est établi par l'expression de switch.

- Si le type de l'expression de commutateur est `sbyte` , `byte` , `short` , `ushort` , `int` , `uint` , `long` , `ulong` , `bool` , `char` , `string` ou un *enum_type*, ou s'il s'agit du type Nullable correspondant à l'un de ces types, alors il s'agit du type gouvernant de l' `switch` instruction.
- Dans le cas contraire, une seule conversion implicite définie par l'utilisateur (conversions définies par l'utilisateur) doit exister du type de l'expression de commutateur à l'un des types régis possibles suivants : `sbyte` , `byte` , `short` , `ushort` , `int` , `uint` , `long` , `ulong` ,, `char` `string` ou, un type Nullable correspondant à l'un de ces types.

- Sinon, si aucune conversion implicite n'existe, ou s'il existe plusieurs conversions implicites, une erreur se produit au moment de la compilation.

L'expression constante de chaque `case` étiquette doit indiquer une valeur implicitement convertible (conversions implicites) en type gouvernant de l' `switch` instruction. Une erreur de compilation se produit si au moins deux `case` étiquettes dans la même `switch` instruction spécifient la même valeur de constante.

Il peut y avoir au plus une `default` étiquette dans une instruction switch.

Une `switch` instruction est exécutée comme suit :

- L'expression de commutateur est évaluée et convertie en type gouvernant.
- Si l'une des constantes spécifiées dans une `case` étiquette dans la même `switch` instruction est égale à la valeur de l'expression de commutateur, le contrôle est transféré à la liste d'instructions à la suite de l' `case` étiquette correspondante.
- Si aucune des constantes spécifiées dans `case` les étiquettes de la même `switch` instruction n'est égale à la valeur de l'expression de commutateur, et si une `default` étiquette est présente, le contrôle est transféré à la liste d'instructions après l' `default` étiquette.
- Si aucune des constantes spécifiées dans `case` les étiquettes de la même `switch` instruction n'est égale à la valeur de l'expression de commutateur, et si aucune `default` étiquette n'est présente, le contrôle est transféré au point de terminaison de l' `switch` instruction.

Si le point de terminaison de la liste d'instructions d'une section switch est accessible, une erreur de compilation se produit. C'est ce que l'on appelle la règle « aucune chute ». L'exemple

C#

```
switch (i) {
case 0:
    CaseZero();
    break;
case 1:
    CaseOne();
    break;
default:
    CaseOthers();
    break;
}
```

est valide, car aucune section switch n'a un point de terminaison accessible. Contrairement à C et C++, l'exécution d'une section switch n'est pas autorisée à passer à la section switch suivante, et l'exemple

```
C#

switch (i) {
case 0:
    CaseZero();
case 1:
    CaseZeroOrOne();
default:
    CaseAny();
}
```

génère une erreur au moment de la compilation. Lorsque l'exécution d'une section switch doit être suivie par l'exécution d'une autre section switch, `goto case` une `goto default` instruction explicite ou doit être utilisée :

```
C#

switch (i) {
case 0:
    CaseZero();
    goto case 1;
case 1:
    CaseZeroOrOne();
    goto default;
default:
    CaseAny();
    break;
}
```

Plusieurs étiquettes sont autorisées dans un *switch_section*. L'exemple

```
C#

switch (i) {
case 0:
    CaseZero();
    break;
case 1:
    CaseOne();
    break;
case 2:
default:
    CaseTwo();
    break;
}
```

est valide. L'exemple ne viole pas la règle « aucune chute », car les étiquettes `case 2:` et `default:` font partie du même *switch_section*.

La règle « aucune chute » empêche une classe courante de bogues qui se produisent en C et C++ lorsque `break` les instructions sont omises par erreur. En outre, en raison de cette règle, les sections Switch d'une `switch` instruction peuvent être réorganisées arbitrairement sans affecter le comportement de l'instruction. Par exemple, les sections de l' `switch` instruction ci-dessus peuvent être inversées sans affecter le comportement de l'instruction :

C#

```
switch (i) {
default:
    CaseAny();
    break;
case 1:
    CaseZeroOrOne();
    goto default;
case 0:
    CaseZero();
    goto case 1;
}
```

La liste d'instructions d'une section switch se termine généralement par une `break` `goto case` instruction,, ou `goto default` , mais toute construction qui restitue le point de terminaison de la liste d'instructions inaccessible est autorisée. Par exemple, une `while` instruction contrôlée par l'expression booléenne `true` est connue pour ne jamais atteindre son point de terminaison. De même, `throw` une `return` instruction ou transfère toujours le contrôle ailleurs et n'atteint jamais son point de terminaison. Ainsi, l'exemple suivant est valide :

C#

```
switch (i) {
case 0:
    while (true) F();
case 1:
    throw new ArgumentException();
case 2:
    return;
}
```

Le type gouvernant d'une `switch` instruction peut être le type `string` . Par exemple :

C#

```
void DoCommand(string command) {
    switch (command.ToLower()) {
    case "run":
        DoRun();
        break;
    case "save":
        DoSave();
        break;
    case "quit":
        DoQuit();
        break;
    default:
        InvalidCommand(command);
        break;
    }
}
```

À l'instar des opérateurs d'égalité de chaînes (opérateurs d'égalité de chaînes), l' `switch` instruction respecte la casse et exécute une section switch donnée uniquement si la chaîne d'expression de commutateur correspond exactement à une `case` constante d'étiquette.

Lorsque le type gouvernant d'une `switch` instruction est `string` , la valeur `null` est autorisée comme constante d'étiquette case.

Le *statement_list* s d'une *switch_block* peut contenir des instructions de déclaration (instructions de déclaration). La portée d'une variable locale ou d'une constante déclarée dans un bloc switch est le bloc switch.

La liste d'instructions d'une section switch donnée est accessible si l' `switch` instruction est accessible et qu'au moins l'une des conditions suivantes est vraie :

- L'expression de commutateur est une valeur non constante.
- L'expression de commutateur est une valeur constante qui correspond à une `case` étiquette dans la section switch.
- L'expression de commutateur est une valeur constante qui ne correspond à aucune `case` étiquette, et la section Switch contient l' `default` étiquette.
- Une étiquette de commutateur de la section switch est référencée par une `goto case` instruction ou accessible `goto default` .

Le point de terminaison d'une `switch` instruction est accessible si au moins l'une des conditions suivantes est vraie :

- L' `switch` instruction contient une instruction accessible `break` qui quitte l' `switch` instruction.
- L' `switch` instruction est accessible, l'expression de commutateur est une valeur non constante et aucune `default` étiquette n'est présente.
- L' `switch` instruction est accessible, l'expression de commutateur est une valeur constante qui ne correspond à aucune `case` étiquette et aucune `default` étiquette n'est présente.

Instructions d'itération

Les instructions d'itération exécutent plusieurs fois une instruction incorporée.

antlr

```
iteration_statement
    : while_statement
    | do_statement
    | for_statement
    | foreach_statement
    ;
```

Instruction while

L' `while` instruction exécute de manière conditionnelle une instruction incorporée zéro ou plusieurs fois.

antlr

```
while_statement
    : 'while' '(' boolean_expression ')' embedded_statement
    ;
```

Une `while` instruction est exécutée comme suit :

- Le *Boolean_expression* (expressions booléennes) est évalué.
- Si l'expression booléenne génère `true` , le contrôle est transféré à l'instruction incorporée. Quand et si le contrôle atteint le point de terminaison de l'instruction incorporée (peut-être à partir de l'exécution d'une `continue` instruction), le contrôle est transféré au début de l' `while` instruction.
- Si l'expression booléenne génère `false` , le contrôle est transféré au point de terminaison de l' `while` instruction.

Dans l'instruction incorporée d'une `while` instruction, une `break` instruction (instruction Break) peut être utilisée pour transférer le contrôle au point de terminaison de l' `while` instruction (qui termine donc l'itération de l'instruction incorporée), et une `continue` instruction (instruction continue) peut être utilisée pour transférer le contrôle au point de terminaison de l'instruction incorporée (ce qui effectue une autre itération de l' `while` instruction).

L'instruction incorporée d'une `while` instruction est accessible si l' `while` instruction est accessible et que l'expression booléenne n'a pas la valeur de constante `false` .

Le point de terminaison d'une `while` instruction est accessible si au moins l'une des conditions suivantes est vraie :

- L' `while` instruction contient une instruction accessible `break` qui quitte l' `while` instruction.
- L' `while` instruction est accessible et l'expression booléenne n'a pas la valeur de constante `true` .

Instruction do

L' `do` instruction exécute de manière conditionnelle une instruction incorporée une ou plusieurs fois.

antlr

```
do_statement
    : 'do' embedded_statement 'while' '(' boolean_expression ')' ';'
    ;
```

Une `do` instruction est exécutée comme suit :

- Le contrôle est transféré à l'instruction incorporée.
- Quand et si le contrôle atteint le point de terminaison de l'instruction incorporée (peut-être à partir de l'exécution d'une `continue` instruction), le *Boolean_expression* (expressions booléennes) est évalué. Si l'expression booléenne génère `true` , le contrôle est transféré au début de l' `do` instruction. Sinon, le contrôle est transféré au point de terminaison de l' `do` instruction.

Dans l'instruction incorporée d'une `do` instruction, une `break` instruction (instruction Break) peut être utilisée pour transférer le contrôle au point de terminaison de l' `do` instruction (qui termine donc l'itération de l'instruction incorporée), et une `continue`

instruction (instruction continue) peut être utilisée pour transférer le contrôle au point de terminaison de l'instruction incorporée.

L'instruction incorporée d'une `do` instruction est accessible si l' `do` instruction est accessible.

Le point de terminaison d'une `do` instruction est accessible si au moins l'une des conditions suivantes est vraie :

- L' `do` instruction contient une instruction accessible `break` qui quitte l' `do` instruction.
- Le point de terminaison de l'instruction incorporée est accessible et l'expression booléenne n'a pas la valeur de constante `true` .

Instruction for

L' `for` instruction évalue une séquence d'expressions d'initialisation, puis, alors qu'une condition a la valeur true, exécute à plusieurs reprises une instruction incorporée et évalue une séquence d'expressions d'itération.

antlr

```
for_statement
    : 'for' '(' for_initializer? ';' for_condition? ';' for_iterator? ')'
embedded_statement
    ;

for_initializer
    : local_variable_declaration
    | statement_expression_list
    ;

for_condition
    : boolean_expression
    ;

for_iterator
    : statement_expression_list
    ;

statement_expression_list
    : statement_expression (',' statement_expression)*
    ;
```

Le *for_initializer*, le cas échéant, se compose soit d'une *local_variable_declaration* (déclarations de variables locales), soit d'une liste d' *statement_expression* s (instructions d'expression) séparées par des virgules. La portée d'une variable locale déclarée par un

for_initializer commence à la *local_variable_declarator* pour la variable et s'étend jusqu'à la fin de l'instruction incorporée. L'étendue comprend les *for_condition* et les *for_iterator*.

Le *for_condition*, le cas échéant, doit être un *Boolean_expression* (expressions booléennes).

Le *for_iterator*, le cas échéant, se compose d'une liste d' *statement_expression* s (instructions d'expression) séparées par des virgules.

Une instruction for est exécutée comme suit :

- Si un *for_initializer* est présent, les initialiseurs de variable ou les expressions d'instruction sont exécutés dans l'ordre dans lequel ils sont écrits. Cette étape n'est effectuée qu'une seule fois.
- Si un *for_condition* est présent, il est évalué.
- Si le *for_condition* n'est pas présent ou si l'évaluation génère `true` , le contrôle est transféré à l'instruction incorporée. Quand et si le contrôle atteint le point de terminaison de l'instruction incorporée (peut-être à partir de l'exécution d'une `continue` instruction), les expressions du *for_iterator*, le cas échéant, sont évaluées dans l'ordre, puis une autre itération est effectuée, en commençant par l'évaluation du *for_condition* à l'étape ci-dessus.
- Si le *for_condition* est présent et que l'évaluation génère `false` , le contrôle est transféré au point de terminaison de l' `for` instruction.

Dans l'instruction incorporée d'une `for` instruction, une `break` instruction (instruction Break) peut être utilisée pour transférer le contrôle au point de terminaison de l' `for` instruction (qui termine donc l'itération de l'instruction incorporée), et une `continue` instruction (l'instruction continue) peut être utilisée pour transférer le contrôle au point de terminaison de l'instruction incorporée (en exécutant ainsi l' *for_iterator* et en exécutant une autre itération de l' `for` instruction, en commençant par la *for_condition*)

L'instruction incorporée d'une `for` instruction est accessible si l'une des conditions suivantes est remplie :

- L' `for` instruction est accessible et aucune *for_condition* n'est présente.
- L' `for` instruction est accessible et un *for_condition* est présent et n'a pas la valeur de constante `false` .

Le point de terminaison d'une `for` instruction est accessible si au moins l'une des conditions suivantes est vraie :

- L' `for` instruction contient une instruction accessible `break` qui quitte l' `for` instruction.

- L' `for` instruction est accessible et un *for_condition* est présent et n'a pas la valeur de constante `true` .

Instruction foreach

L' `foreach` instruction énumère les éléments d'une collection, en exécutant une instruction incorporée pour chaque élément de la collection.

```antlr
foreach_statement
    : 'foreach' '(' local_variable_type identifier 'in' expression ')'
embedded_statement
    ;
```

Le *type* et l' *identificateur* d'une `foreach` instruction déclarent la **variable d'itération*** _ de l'instruction. Si l' `var` identificateur est fourni en tant que _local_variable_type * et qu'aucun type nommé `var` n'est dans la portée, la variable d'itération est dite une ***variable d'itération implicitement typée*** et son type est considéré comme étant le type d'élément de l' `foreach` instruction, comme indiqué ci-dessous. La variable d'itération correspond à une variable locale en lecture seule avec une portée qui s'étend sur l'instruction incorporée. Pendant l'exécution d'une `foreach` instruction, la variable d'itération représente l'élément de collection pour lequel une itération est en cours d'exécution. Une erreur de compilation se produit si l'instruction incorporée tente de modifier la variable d'itération (par le biais de l'assignation ou des `++` `--` opérateurs et) ou de passer la variable d'itération en tant que `ref` `out` paramètre ou.

Dans ce qui suit, par souci de concision,, `IEnumerable` `IEnumerator` `IEnumerable<T>` et `IEnumerator<T>` font référence aux types correspondants dans les espaces de noms `System.Collections` et `System.Collections.Generic` .

Le traitement au moment de la compilation d'une instruction foreach détermine d'abord le * type de **collection** , *le type d' *énumérateur** et *le type d'élément _ de l'expression.* and _ *element type** of the expression. Cette détermination se déroule comme suit :

- Si le type `X` d' *expression* est un type tableau, il existe une conversion de référence implicite de `X` en `IEnumerable` interface (puisque `System.Array` implémente cette interface). Le ***type de collection*** _ *est l'* `IEnumerable` *interface, le _type d'énumérateur_ est l'* `IEnumerator` *interface et le type d'élément** _ est le type d'élément du type tableau `X` .

- Si le type `X` d' *expression* est `dynamic` , il y a une conversion implicite d' *expression* en `IEnumerable` interface (conversions dynamiques implicites). Le ***type de collection*** *_ est l'* `IEnumerable` *interface et le _type d'énumérateur*_* est l' `IEnumerator` interface. Si l' `var` identificateur est fourni en tant que _local_variable_type *, le ***type d'élément*** est `dynamic` , sinon, `object` .

- Sinon, déterminez si le type `X` a une `GetEnumerator` méthode appropriée :
  - Effectuez une recherche de membre sur le type `X` avec l'identificateur `GetEnumerator` et aucun argument de type. Si la recherche de membre ne produit pas de correspondance, si elle génère une ambiguïté ou si elle produit une correspondance qui n'est pas un groupe de méthodes, recherchez une interface énumérable comme décrit ci-dessous. Il est recommandé d'émettre un avertissement si la recherche de membres produit tout sauf un groupe de méthodes ou aucune correspondance.
  - Exécutez la résolution de surcharge à l'aide du groupe de méthodes résultant et d'une liste d'arguments vide. Si la résolution de surcharge entraîne l'absence de méthodes applicables, provoque une ambiguïté ou produit une seule meilleure méthode, mais que cette méthode est statique ou non publique, recherchez une interface énumérable comme décrit ci-dessous. Il est recommandé d'émettre un avertissement si la résolution de surcharge produit tout sauf une méthode d'instance publique non ambiguë ou aucune méthode applicable.
  - Si le type `E` de retour de la `GetEnumerator` méthode n'est pas un type de classe, de struct ou d'interface, une erreur est générée et aucune autre étape n'est prise.
  - La recherche de membres est effectuée sur `E` avec l'identificateur `Current` et aucun argument de type. Si la recherche de membre ne produit aucune correspondance, le résultat est une erreur, ou le résultat est tout sauf une propriété d'instance publique qui autorise la lecture, une erreur est générée et aucune autre étape n'est effectuée.
  - La recherche de membres est effectuée sur `E` avec l'identificateur `MoveNext` et aucun argument de type. Si la recherche de membre ne produit aucune correspondance, le résultat est une erreur, ou le résultat est tout sauf un groupe de méthodes, une erreur est générée et aucune autre étape n'est prise.
  - La résolution de surcharge est effectuée sur le groupe de méthodes avec une liste d'arguments vide. Si la résolution de surcharge entraîne l'absence de méthodes applicables, entraîne une ambiguïté ou produit une seule meilleure méthode, mais cette méthode est statique ou non publique, ou son type de retour n'est pas `bool` , une erreur est générée et aucune autre étape n'est prise.
  - Le ***type de collection*** *_ est* `X` *, le _type d'énumérateur_ est* `E` *, et le type d'élément** _ est le type de la `Current` propriété.

- Sinon, recherchez une interface énumérable :
  - Si parmi tous les types `Ti` pour lesquels il existe une conversion implicite de `X` en `IEnumerable<Ti>` , il existe un type unique `T` , qui n' `T` est pas `dynamic` et pour tout l'autre, `Ti` il existe une conversion implicite de `IEnumerable<T>` en `IEnumerable<Ti>` , puis le * type de **collection** _ est l'interface `IEnumerable<T>` , le type d' **énumérateur** est l'interface `IEnumerator<T>` et le type d'*élément** _ *est* `T` ._ is the interface `IEnumerator<T>`, and the _ *element type** is `T`.
  - Sinon, s'il existe plusieurs types de ce type `T` , une erreur est générée et aucune autre étape n'est effectuée.
  - Sinon, s'il existe une conversion implicite de `X` en `System.Collections.IEnumerable` interface, le ***type de collection*** _ *est cette interface, le _type d'énumérateur*_* est l'interface `System.Collections.IEnumerator` et le *type d'élément* _ est `object` .
  - Dans le cas contraire, une erreur est générée et aucune autre étape n'est prise.

Les étapes ci-dessus, en cas de réussite, produisent sans ambiguïté un type de collection, un type d' `C` énumérateur `E` et un type d'élément `T` . Une instruction foreach de la forme

```
C#

foreach (V v in x) embedded_statement
```

est ensuite développé pour :

```
C#

{
    E e = ((C)(x)).GetEnumerator();
    try {
        while (e.MoveNext()) {
            V v = (V)(T)e.Current;
            embedded_statement
        }
    }
    finally {
        ... // Dispose e
    }
}
```

La variable `e` n'est pas visible ou accessible à l'expression `x` ou à l'instruction incorporée ou à tout autre code source du programme. La variable `v` est en lecture seule dans l'instruction incorporée. S'il n'existe pas de conversion explicite (conversions

explicites) de `T` (le type d'élément) en `V` (le *local_variable_type* dans l'instruction foreach), une erreur est générée et aucune autre étape n'est effectuée. Si `x` a la valeur `null` , une `System.NullReferenceException` est levée au moment de l'exécution.

Une implémentation est autorisée à implémenter une instruction foreach donnée différemment, par exemple pour des raisons de performances, tant que le comportement est cohérent avec l'expansion ci-dessus.

La position `v` à l'intérieur de la boucle while est importante pour la façon dont elle est capturée par toute fonction anonyme se produisant dans le *embedded_statement*.

Par exemple :

```C#
int[] values = { 7, 9, 13 };
Action f = null;

foreach (var value in values)
{
    if (f == null) f = () => Console.WriteLine("First value: " + value);
}

f();
```

Si `v` a été déclaré en dehors de la boucle while, il est partagé entre toutes les itérations, et sa valeur après la boucle for est la valeur finale, `13` , qui correspond à l'impression de l'appel `f` . Au lieu de cela, étant donné que chaque itération a sa propre variable `v` , celle capturée par `f` dans la première itération continue à contenir la valeur `7` , qui est celle qui sera imprimée. (Remarque : les versions antérieures de C# ont déclaré `v` en dehors de la boucle while.)

Le corps du bloc finally est construit selon les étapes suivantes :

- S'il existe une conversion implicite de `E` en `System.IDisposable` interface,
  - Si `E` est un type valeur qui n'autorise pas les valeurs NULL, la clause finally est développée avec l'équivalent sémantique de :

```C#
finally {
    ((System.IDisposable)e).Dispose();
}
```

- Dans le cas contraire, la clause finally est étendue à l'équivalent sémantique de :

```
C#

finally {
    if (e != null) ((System.IDisposable)e).Dispose();
}
```

Hormis `E` le fait que si est un type valeur ou un paramètre de type instancié à un type valeur, le cast de `e` en n' `System.IDisposable` entraîne pas de conversion boxing.

- Sinon, si `E` est un type sealed, la clause finally est développée en bloc vide :

```
C#

finally {
}
```

- Dans le cas contraire, la clause finally est développée en :

```
C#

finally {
    System.IDisposable d = e as System.IDisposable;
    if (d != null) d.Dispose();
}
```

La variable locale `d` n'est pas visible ou accessible à un code utilisateur. En particulier, il n'est pas en conflit avec une autre variable dont la portée comprend le bloc finally.

L'ordre dans lequel `foreach` traverse les éléments d'un tableau est le suivant : pour les tableaux unidimensionnels, les éléments sont parcourus dans l'ordre d'index de plus en plus important, en commençant par l'index `0` et en terminant par l'index `Length - 1` . Pour les tableaux multidimensionnels, les éléments sont parcourus de telle sorte que les index de la dimension la plus à droite soient augmentés en premier, puis la dimension gauche suivante, et ainsi de suite vers la gauche.

L'exemple suivant imprime chaque valeur dans un tableau à deux dimensions, dans l'ordre des éléments :

```
C#
```

```
using System;

class Test
{
    static void Main() {
        double[,] values = {
            {1.2, 2.3, 3.4, 4.5},
            {5.6, 6.7, 7.8, 8.9}
        };

        foreach (double elementValue in values)
            Console.Write("{0} ", elementValue);

        Console.WriteLine();
    }
}
```

La sortie produite est la suivante :

Console

```
1.2 2.3 3.4 4.5 5.6 6.7 7.8 8.9
```

Dans l'exemple

C#

```
int[] numbers = { 1, 3, 5, 7, 9 };
foreach (var n in numbers) Console.WriteLine(n);
```

le type de `n` est déduit comme étant `int` , le type d'élément de `numbers` .

Instructions de saut

Les instructions de saut transfèrent le contrôle de manière inconditionnelle.

antlr

```
jump_statement
    : break_statement
    | continue_statement
    | goto_statement
    | return_statement
    | throw_statement
    ;
```

L'emplacement vers lequel une instruction de saut transfère le contrôle est appelé la ***cible*** de l'instruction de saut.

Lorsqu'une instruction de saut se produit dans un bloc et que la cible de cette instruction de saut est en dehors de ce bloc, l'instruction de saut est dite pour ***quitter*** le bloc. Alors qu'une instruction de saut peut transférer le contrôle hors d'un bloc, elle ne peut jamais transférer le contrôle dans un bloc.

L'exécution d'instructions de saut est compliquée par la présence d' `try` instructions intermédiaires. En l'absence de ces `try` instructions, une instruction de saut transfère de manière inconditionnelle le contrôle de l'instruction de saut à sa cible. En présence de ces instructions intervenues `try` , l'exécution est plus complexe. Si l'instruction de saut quitte un ou plusieurs `try` blocs avec des `finally` blocs associés, le contrôle est initialement transféré au `finally` bloc de l'instruction la plus profonde `try` . Quand et si le contrôle atteint le point de terminaison d'un `finally` bloc, le contrôle est transféré au `finally` bloc de l'instruction englobante suivante `try` . Ce processus est répété jusqu'à ce que les `finally` blocs de toutes les `try` instructions intermédiaires aient été exécutés.

Dans l'exemple

C#

```
using System;

class Test
{
    static void Main() {
        while (true) {
            try {
                try {
                    Console.WriteLine("Before break");
                    break;
                }
                finally {
                    Console.WriteLine("Innermost finally block");
                }
            }
            finally {
                Console.WriteLine("Outermost finally block");
            }
        }
        Console.WriteLine("After break");
    }
}
```

les `finally` blocs associés à deux `try` instructions sont exécutés avant que le contrôle soit transféré à la cible de l'instruction de saut.

La sortie produite est la suivante :

Console

```
Before break
Innermost finally block
Outermost finally block
After break
```

Instruction Break

L' `break` instruction quitte l'instruction,,, ou englobante la plus proche `switch` `while` `do` `for` `foreach` .

antlr

```
break_statement
    : 'break' ';'
    ;
```

La cible d'une `break` instruction est le point de terminaison de l'instruction,,, ou englobante la plus proche `switch` `while` `do` `for` `foreach` . Si une `break` instruction n'est pas entourée d'une `switch` `while` instruction,, `do` , `for` ou `foreach` , une erreur de compilation se produit.

Lorsque plusieurs `switch` `while` instructions,,, `do` `for` , ou `foreach` sont imbriquées les unes dans les autres, une `break` instruction s'applique uniquement à l'instruction la plus profonde. Pour transférer le contrôle entre plusieurs niveaux d'imbrication, une `goto` instruction (instruction goto) doit être utilisée.

Une `break` instruction ne peut pas quitter un `finally` bloc (instruction try). Lorsqu'une `break` instruction se produit dans un `finally` bloc, la cible de l' `break` instruction doit se trouver dans le même `finally` bloc ; sinon, une erreur de compilation se produit.

Une `break` instruction est exécutée comme suit :

- Si l' `break` instruction quitte un ou plusieurs `try` blocs avec des `finally` blocs associés, le contrôle est initialement transféré au `finally` bloc de l'instruction la plus profonde `try` . Quand et si le contrôle atteint le point de terminaison d'un `finally` bloc, le contrôle est transféré au `finally` bloc de l'instruction englobante

suivante `try` . Ce processus est répété jusqu'à ce que les `finally` blocs de toutes les `try` instructions intermédiaires aient été exécutés.
- Le contrôle est transféré à la cible de l' `break` instruction.

Étant donné qu'une `break` instruction transfère sans condition le contrôle ailleurs, le point de terminaison d'une `break` instruction n'est jamais accessible.

Instruction continue

L' `continue` instruction démarre une nouvelle itération de l' `while` `do` instruction,, ou englobante la plus proche `for` `foreach` .

antlr

```
continue_statement
    : 'continue' ';'
    ;
```

La cible d'une `continue` instruction est le point de terminaison de l'instruction incorporée de l'instruction englobante `while` ,, ou englobante la plus proche `do` `for` `foreach` . Si une `continue` instruction n'est pas entourée d'une `while` `do` instruction,, `for` ou `foreach` , une erreur de compilation se produit.

Lorsque plusieurs `while` `do` instructions,, `for` , ou `foreach` sont imbriquées les unes dans les autres, une `continue` instruction s'applique uniquement à l'instruction la plus profonde. Pour transférer le contrôle entre plusieurs niveaux d'imbrication, une `goto` instruction (instruction goto) doit être utilisée.

Une `continue` instruction ne peut pas quitter un `finally` bloc (instruction try). Lorsqu'une `continue` instruction se produit dans un `finally` bloc, la cible de l' `continue` instruction doit se trouver dans le même `finally` bloc ; sinon, une erreur de compilation se produit.

Une `continue` instruction est exécutée comme suit :

- Si l' `continue` instruction quitte un ou plusieurs `try` blocs avec des `finally` blocs associés, le contrôle est initialement transféré au `finally` bloc de l'instruction la plus profonde `try` . Quand et si le contrôle atteint le point de terminaison d'un `finally` bloc, le contrôle est transféré au `finally` bloc de l'instruction englobante suivante `try` . Ce processus est répété jusqu'à ce que les `finally` blocs de toutes les `try` instructions intermédiaires aient été exécutés.

- Le contrôle est transféré à la cible de l' `continue` instruction.

Étant donné qu'une `continue` instruction transfère sans condition le contrôle ailleurs, le point de terminaison d'une `continue` instruction n'est jamais accessible.

Instruction goto

L' `goto` instruction transfère le contrôle à une instruction qui est marquée par une étiquette.

antlr

```
goto_statement
    : 'goto' identifier ';'
    | 'goto' 'case' constant_expression ';'
    | 'goto' 'default' ';'
    ;
```

La cible d'une `goto` instruction d' *identificateur* est l'instruction étiquetée avec l'étiquette donnée. Si une étiquette portant le nom spécifié n'existe pas dans le membre de fonction actuel, ou si l' `goto` instruction n'est pas dans la portée de l'étiquette, une erreur de compilation se produit. Cette règle autorise l'utilisation d'une `goto` instruction pour transférer le contrôle hors d'une portée imbriquée, mais pas dans une portée imbriquée. Dans l'exemple

C#

```
using System;

class Test
{
    static void Main(string[] args) {
        string[,] table = {
            {"Red", "Blue", "Green"},
            {"Monday", "Wednesday", "Friday"}
        };

        foreach (string str in args) {
            int row, colm;
            for (row = 0; row <= 1; ++row)
                for (colm = 0; colm <= 2; ++colm)
                    if (str == table[row,colm])
                        goto done;

            Console.WriteLine("{0} not found", str);
            continue;
    done:
            Console.WriteLine("Found {0} at [{1}][{2}]", str, row, colm);
```

```
        }
    }
}
```

une `goto` instruction est utilisée pour transférer le contrôle hors d'une portée imbriquée.

La cible d'une `goto case` instruction est la liste d'instructions dans l'instruction immédiatement englobante `switch` (l'instruction switch), qui contient une `case` étiquette avec la valeur de constante donnée. Si l' `goto case` instruction n'est pas entourée d'une `switch` instruction, si la *constant_expression* n'est pas implicitement convertible (conversion implicite) en type gouvernant de l'instruction englobante la plus proche `switch` , ou si l'instruction englobante la plus proche `switch` ne contient pas d' `case` étiquette avec la valeur de constante donnée, une erreur de compilation se produit.

La cible d'une `goto default` instruction est la liste d'instructions dans l'instruction immédiatement englobante `switch` (l'instruction switch), qui contient une `default` étiquette. Si l' `goto default` instruction n'est pas entourée d'une `switch` instruction, ou si l'instruction englobante la plus proche `switch` ne contient pas d' `default` étiquette, une erreur de compilation se produit.

Une `goto` instruction ne peut pas quitter un `finally` bloc (instruction try). Lorsqu'une `goto` instruction se produit dans un `finally` bloc, la cible de l' `goto` instruction doit se trouver dans le même `finally` bloc, sinon une erreur de compilation se produit.

Une `goto` instruction est exécutée comme suit :

- Si l' `goto` instruction quitte un ou plusieurs `try` blocs avec des `finally` blocs associés, le contrôle est initialement transféré au `finally` bloc de l'instruction la plus profonde `try` . Quand et si le contrôle atteint le point de terminaison d'un `finally` bloc, le contrôle est transféré au `finally` bloc de l'instruction englobante suivante `try` . Ce processus est répété jusqu'à ce que les `finally` blocs de toutes les `try` instructions intermédiaires aient été exécutés.
- Le contrôle est transféré à la cible de l' `goto` instruction.

Étant donné qu'une `goto` instruction transfère sans condition le contrôle ailleurs, le point de terminaison d'une `goto` instruction n'est jamais accessible.

Instruction return

L' `return` instruction retourne le contrôle à l'appelant actuel de la fonction dans laquelle l' `return` instruction apparaît.

antlr

```
return_statement
    : 'return' expression? ';'
    ;
```

Une `return` instruction sans expression peut être utilisée uniquement dans un membre de fonction qui ne calcule pas de valeur, autrement dit, une méthode avec le type de résultat (corps de méthode) `void` , l' `set` accesseur d'une propriété ou d'un indexeur, les `add` `remove` accesseurs et d'un événement, un constructeur d'instance, un constructeur statique ou un destructeur.

Une `return` instruction avec une expression ne peut être utilisée que dans un membre de fonction qui calcule une valeur, autrement dit, une méthode avec un type de résultat non void, l' `get` accesseur d'une propriété ou un indexeur ou un opérateur défini par l'utilisateur. Une conversion implicite (conversions implicites) doit exister du type de l'expression vers le type de retour de la fonction membre conteneur.

Les instructions Return peuvent également être utilisées dans le corps d'expressions de fonction anonymes (expressions de fonction anonymes) et participer à la détermination des conversions qui existent pour ces fonctions.

Il s'agit d'une erreur de compilation pour qu'une `return` instruction apparaisse dans un `finally` bloc (instruction try).

Une `return` instruction est exécutée comme suit :

- Si l' `return` instruction spécifie une expression, l'expression est évaluée et la valeur résultante est convertie dans le type de retour de la fonction conteneur par une conversion implicite. Le résultat de la conversion devient la valeur de résultat produite par la fonction.
- Si l' `return` instruction est entourée d'un ou de plusieurs `try` `catch` blocs ou avec des `finally` blocs associés, le contrôle est initialement transféré au `finally` bloc de l'instruction la plus profonde `try` . Quand et si le contrôle atteint le point de terminaison d'un `finally` bloc, le contrôle est transféré au `finally` bloc de l'instruction englobante suivante `try` . Ce processus est répété jusqu'à ce que les `finally` blocs de toutes les `try` instructions englobantes aient été exécutés.
- Si la fonction conteneur n'est pas une fonction Async, le contrôle est retourné à l'appelant de la fonction conteneur avec la valeur de résultat, le cas échéant.

- Si la fonction conteneur est une fonction Async, le contrôle est retourné à l'appelant actuel et la valeur de résultat, le cas échéant, est enregistrée dans la tâche de retour, comme décrit dans (interfaces d'énumération).

Étant donné qu'une `return` instruction transfère sans condition le contrôle ailleurs, le point de terminaison d'une `return` instruction n'est jamais accessible.

Instruction throw

L'instruction `throw`lève une exception.

antlr

```
throw_statement
    : 'throw' expression? ';'
    ;
```

Une `throw` instruction avec une expression génère la valeur produite par l'évaluation de l'expression. L'expression doit indiquer une valeur du type de classe `System.Exception` , d'un type de classe qui dérive de `System.Exception` ou d'un type de paramètre de type ayant `System.Exception` (ou une sous-classe) comme classe de base effective. Si l'évaluation de l'expression produit `null` , une `System.NullReferenceException` exception est levée à la place.

Une `throw` instruction sans expression ne peut être utilisée que dans un `catch` bloc, auquel cas cette instruction lève à nouveau l'exception qui est actuellement gérée par ce `catch` bloc.

Étant donné qu'une `throw` instruction transfère sans condition le contrôle ailleurs, le point de terminaison d'une `throw` instruction n'est jamais accessible.

Quand une exception est levée, le contrôle est transféré à la première `catch` clause dans une instruction englobante `try` qui peut gérer l'exception. Le processus qui a lieu à partir du point de l'exception levée jusqu'au point de transfert du contrôle à un gestionnaire d'exceptions approprié est appelé *__propagation des exceptions__ _. La propagation d'une exception consiste à évaluer à plusieurs reprises les étapes suivantes jusqu'à ce qu'une `catch` clause qui corresponde à l'exception soit trouvée. Dans cette description, le _ *point levé** est initialement l'emplacement où l'exception est levée.

- Dans le membre de fonction actuel, chaque `try` instruction qui encadre le point d'exception est examinée. Pour chaque instruction `S` , en commençant par

l'instruction la plus profonde `try` et en terminant par l'instruction la plus extérieure `try` , les étapes suivantes sont évaluées :

- Si le `try` bloc de `S` encadre le point de levée et si S a une ou plusieurs `catch` clauses, les `catch` clauses sont examinées par ordre d'apparition pour trouver un gestionnaire approprié pour l'exception, conformément aux règles spécifiées dans la section de l'instruction try. Si une `catch` clause correspondante est trouvée, la propagation de l'exception est terminée en transférant le contrôle au bloc de cette `catch` clause.

- Sinon, si le `try` bloc ou un `catch` bloc de `S` englobe le point de levée et si `S` a un `finally` bloc, le contrôle est transféré au `finally` bloc. Si le `finally` bloc lève une autre exception, le traitement de l'exception actuelle est terminé. Dans le cas contraire, lorsque le contrôle atteint le point de terminaison du `finally` bloc, le traitement de l'exception actuelle est poursuivi.

- Si un gestionnaire d'exceptions n'a pas été trouvé dans l'appel de la fonction en cours, l'appel de la fonction se termine et l'un des éléments suivants se produit :

  - Si la fonction active est non Async, les étapes ci-dessus sont répétées pour l'appelant de la fonction avec un point d'exception correspondant à l'instruction à partir de laquelle la fonction membre a été appelée.

  - Si la fonction active est Async et retourne une tâche, l'exception est enregistrée dans la tâche de retour, qui est placée dans un état Faulted ou Cancelled, comme décrit dans interfaces d'énumérateur.

  - Si la fonction active est Async et void-retournant, le contexte de synchronisation du thread actuel est notifié comme décrit dans interfaces Enumerable.

- Si le traitement des exceptions met fin à tous les appels de membre de fonction dans le thread actuel, ce qui indique que le thread n'a aucun gestionnaire pour l'exception, le thread se termine lui-même. L'impact d'une telle terminaison est défini par l'implémentation.

Instruction try

L' `try` instruction fournit un mécanisme permettant d'intercepter les exceptions qui se produisent pendant l'exécution d'un bloc. En outre, l' `try` instruction offre la possibilité de spécifier un bloc de code qui est toujours exécuté lorsque le contrôle quitte l' `try` instruction.

antlr

```
try_statement
    : 'try' block catch_clause+
    | 'try' block finally_clause
    | 'try' block catch_clause+ finally_clause
    ;

catch_clause
    : 'catch' exception_specifier? exception_filter?  block
    ;

exception_specifier
    : '(' type identifier? ')'
    ;

exception_filter
    : 'when' '(' expression ')'
    ;

finally_clause
    : 'finally' block
    ;
```

Il existe trois formes d' `try` instructions possibles :

- `try`Bloc suivi d'un ou de plusieurs `catch` blocs.
- `try`Bloc suivi d'un `finally` bloc.
- `try`Bloc suivi d'un ou de plusieurs `catch` blocs suivis d'un `finally` bloc.

Quand une `catch` clause spécifie un *exception_specifier*, le type doit être `System.Exception` , un type qui dérive de `System.Exception` ou un type de paramètre de type qui a `System.Exception` (ou une sous-classe de ce type) comme classe de base effective.

Quand une `catch` clause spécifie à la fois un *exception_specifier* avec un *identificateur*, une ***variable d'exception** _ du nom et du type spécifiés est déclarée. La variable d'exception correspond à une variable locale avec une étendue qui s'étend sur la `catch` clause. Pendant l'exécution du _exception_filter * et du *bloc*, la variable d'exception représente l'exception actuellement gérée. À des fins de vérification de l'attribution définie, la variable d'exception est considérée comme assignée de manière définitive dans toute son étendue.

À moins qu'une `catch` clause comprenne un nom de variable d'exception, il est impossible d'accéder à l'objet d'exception dans le filtre et le `catch` bloc.

Une `catch` clause qui ne spécifie pas de *exception_specifier* est appelée une `catch` clause générale.

Certains langages de programmation peuvent prendre en charge des exceptions qui ne peuvent pas être représentées en tant qu'objet dérivé de `System.Exception` , bien que ces exceptions ne puissent jamais être générées par du code C#. Une `catch` clause générale peut être utilisée pour intercepter ces exceptions. Par conséquent, une `catch` clause générale est sémantiquement différente d'une clause qui spécifie le type `System.Exception` , dans le cas où la première peut également intercepter des exceptions d'autres langages.

Afin de localiser un gestionnaire pour une exception, `catch` les clauses sont examinées dans l'ordre lexical. Si une `catch` clause spécifie un type mais aucun filtre d'exception, il s'agit d'une erreur de compilation pour une `catch` clause ultérieure dans la même `try` instruction pour spécifier un type qui est le même que, ou qui est dérivé de ce type. Si une `catch` clause ne spécifie aucun type et aucun filtre, il doit s'agir de la dernière `catch` clause pour cette `try` instruction.

Dans un `catch` bloc, une `throw` instruction (instruction throw) sans expression peut être utilisée pour lever à nouveau l'exception interceptée par le `catch` bloc. Les assignations à une variable d'exception ne modifient pas l'exception qui est levée une nouvelle fois.

Dans l'exemple

C#

```
using System;

class Test
{
    static void F() {
        try {
            G();
        }
        catch (Exception e) {
            Console.WriteLine("Exception in F: " + e.Message);
            e = new Exception("F");
            throw;                // re-throw
        }
    }

    static void G() {
        throw new Exception("G");
    }

    static void Main() {
        try {
```

```
            F();
        }
        catch (Exception e) {
            Console.WriteLine("Exception in Main: " + e.Message);
        }
    }
}
```

la méthode `F` intercepte une exception, écrit des informations de diagnostic dans la console, modifie la variable d'exception et lève à nouveau l'exception. L'exception qui est levée une nouvelle fois est l'exception d'origine, de sorte que la sortie produite est :

Console

```
Exception in F: G
Exception in Main: G
```

Si le premier bloc catch avait été levé `e` au lieu de lever à nouveau l'exception actuelle, la sortie produite serait la suivante :

Console

```
Exception in F: G
Exception in Main: F
```

Il s'agit d'une erreur de compilation pour `break` une `continue` instruction, ou `goto` pour transférer le contrôle hors d'un `finally` bloc. Quand une `break` `continue` instruction, ou `goto` se trouve dans un `finally` bloc, la cible de l'instruction doit se trouver dans le même `finally` bloc, sinon une erreur de compilation se produit.

Il s'agit d'une erreur de compilation pour qu'une `return` instruction se produise dans un `finally` bloc.

Une `try` instruction est exécutée comme suit :

- Le contrôle est transféré au `try` bloc.
- Quand et si le contrôle atteint le point de terminaison du `try` bloc :
  - Si l' `try` instruction contient un `finally` bloc, le `finally` bloc est exécuté.
  - Le contrôle est transféré au point de terminaison de l' `try` instruction.
- Si une exception est propagée à l' `try` instruction pendant l'exécution du `try` bloc :

- Les `catch` clauses, le cas échéant, sont examinées par ordre d'apparition pour trouver un gestionnaire approprié pour l'exception. Si une `catch` clause ne spécifie pas de type, ou spécifie le type d'exception ou un type de base du type d'exception :
  - Si la `catch` clause déclare une variable d'exception, l'objet exception est assigné à la variable d'exception.
  - Si la `catch` clause déclare un filtre d'exception, le filtre est évalué. Si `false` la valeur est, la clause catch n'est pas une correspondance, et la recherche se poursuit à travers toutes les `catch` clauses suivantes pour un gestionnaire approprié.
  - Dans le cas contraire, la `catch` clause est considérée comme une correspondance et le contrôle est transféré au `catch` bloc correspondant.
  - Quand et si le contrôle atteint le point de terminaison du `catch` bloc :
    - Si l' `try` instruction contient un `finally` bloc, le `finally` bloc est exécuté.
    - Le contrôle est transféré au point de terminaison de l' `try` instruction.
  - Si une exception est propagée à l' `try` instruction pendant l'exécution du `catch` bloc :
    - Si l' `try` instruction contient un `finally` bloc, le `finally` bloc est exécuté.
    - L'exception est propagée à l'instruction englobante suivante `try` .
- Si l' `try` instruction n'a pas de `catch` clauses ou si aucune `catch` clause ne correspond à l'exception :
  - Si l' `try` instruction contient un `finally` bloc, le `finally` bloc est exécuté.
  - L'exception est propagée à l'instruction englobante suivante `try` .

Les instructions d'un `finally` bloc sont toujours exécutées lorsque le contrôle quitte une `try` instruction. Cela est vrai, que le transfert de contrôle se produise suite à une exécution normale, à la suite de l'exécution d'une `break` instruction,, `continue` `goto` ou `return` , ou suite à la propagation d'une exception à partir de l' `try` instruction.

Si une exception est levée pendant l'exécution d'un `finally` bloc et n'est pas interceptée dans le même bloc finally, l'exception est propagée à l'instruction englobante suivante `try` . Si une autre exception était en cours de propagation, cette exception est perdue. Le processus de propagation d'une exception est abordé plus en détail dans la description de l' `throw` instruction (instruction throw).

Le `try` bloc d'une `try` instruction est accessible si l' `try` instruction est accessible.

Un `catch` bloc d' `try` instruction est accessible si l' `try` instruction est accessible.

Le `finally` bloc d'une `try` instruction est accessible si l' `try` instruction est accessible.

Le point de terminaison d'une `try` instruction est accessible si les deux conditions suivantes sont remplies :

- Le point de terminaison du `try` bloc est accessible ou le point de terminaison d'au moins un `catch` bloc est accessible.
- Si un `finally` bloc est présent, le point de terminaison du `finally` bloc est accessible.

Les instructions checked et unchecked

Les `checked` `unchecked` instructions et sont utilisées pour contrôler le ***contexte*** de contrôle de dépassement de capacité pour les opérations arithmétiques de type intégral et les conversions.

antlr

```
checked_statement
    : 'checked' block
    ;

unchecked_statement
    : 'unchecked' block
    ;
```

L' `checked` instruction entraîne l'évaluation de toutes les expressions du *bloc* dans un contexte vérifié, et l' `unchecked` instruction entraîne l'évaluation de toutes les expressions du *bloc* dans un contexte non vérifié.

Les `checked` `unchecked` instructions et sont équivalentes de manière précise aux `checked` `unchecked` opérateurs et (opérateurs vérifiés etnon vérifiés), sauf qu'ils opèrent sur des blocs au lieu d'expressions.

lock, instruction

L' `lock` instruction obtient le verrou d'exclusion mutuelle pour un objet donné, exécute une instruction, puis libère le verrou.

antlr

```
lock_statement
    : 'lock' '(' expression ')' embedded_statement
    ;
```

L'expression d'une `lock` instruction doit désigner une valeur d'un type connu comme étant un *reference_type*. Aucune conversion boxing implicite (conversions boxing) n'est jamais effectuée pour l'expression d'une `lock` instruction. par conséquent, il s'agit d'une erreur de compilation pour que l'expression désigne une valeur d'un *Value_type*.

Une `lock` déclaration de la forme

C#

```
lock (x) ...
```

où `x` est une expression d'un *reference_type*, équivaut précisément à

C#

```
bool __lockWasTaken = false;
try {
    System.Threading.Monitor.Enter(x, ref __lockWasTaken);
    ...
}
finally {
    if (__lockWasTaken) System.Threading.Monitor.Exit(x);
}
```

sauf que `x` n'est évalué qu'une seule fois.

Lorsqu'un verrou d'exclusion mutuelle est maintenu, le code qui s'exécute dans le même thread d'exécution peut également obtenir et libérer le verrou. Toutefois, le code qui s'exécute dans d'autres threads ne peut pas obtenir le verrou tant que le verrou n'est pas libéré.

Le verrouillage `System.Type` des objets pour synchroniser l'accès aux données statiques n'est pas recommandé. Un autre code peut verrouiller le même type, ce qui peut entraîner un interblocage. Une meilleure approche consiste à synchroniser l'accès aux données statiques en verrouillant un objet statique privé. Par exemple :

C#

```
class Cache
{
    private static readonly object synchronizationObject = new object();

    public static void Add(object x) {
        lock (Cache.synchronizationObject) {
            ...
        }
    }
```

```
    public static void Remove(object x) {
        lock (Cache.synchronizationObject) {
            ...
        }
    }
}
```

Instruction using

L' `using` instruction obtient une ou plusieurs ressources, exécute une instruction, puis libère la ressource.

antlr

```
using_statement
    : 'using' '(' resource_acquisition ')' embedded_statement
    ;

resource_acquisition
    : local_variable_declaration
    | expression
    ;
```

Une ***ressource*** est une classe ou un struct qui implémente `System.IDisposable` , qui comprend une méthode unique sans paramètre nommée `Dispose` . Le code qui utilise une ressource peut appeler `Dispose` pour indiquer que la ressource n'est plus nécessaire. Si `Dispose` n'est pas appelé, la suppression automatique finit par se produire à la suite de garbage collection.

Si la forme de *resource_acquisition* est *local_variable_declaration* le type de la *local_variable_declaration* doit être `dynamic` ou un type qui peut être implicitement converti en `System.IDisposable` . Si la forme de *resource_acquisition* est *expression* , cette expression doit être implicitement convertible en `System.IDisposable` .

Les variables locales déclarées dans un *resource_acquisition* sont en lecture seule et doivent inclure un initialiseur. Une erreur de compilation se produit si l'instruction incorporée tente de modifier ces variables locales (par le biais de l'assignation ou des `++` `--` opérateurs et), d'en prendre l'adresse ou de les passer en tant que `ref` `out` paramètres ou.

Une `using` instruction est traduite en trois parties : l'acquisition, l'utilisation et la suppression. L'utilisation de la ressource est implicitement incluse dans une `try` instruction qui comprend une `finally` clause. Cette `finally` clause supprime la

ressource. Si une `null` ressource est acquise, aucun appel à `Dispose` n'est effectué et aucune exception n'est levée. Si la ressource est de type `dynamic` , elle est convertie dynamiquement via une conversion dynamique implicite (conversions dynamiques implicites) en `IDisposable` lors de l'acquisition afin de garantir la réussite de la conversion avant l'utilisation et la suppression.

Une `using` déclaration de la forme

C#

```
using (ResourceType resource = expression) statement
```

correspond à l'une des trois expansions possibles. Lorsque `ResourceType` est un type valeur qui n'autorise pas les valeurs NULL, l'expansion est

C#

```
{
    ResourceType resource = expression;
    try {
        statement;
    }
    finally {
        ((IDisposable)resource).Dispose();
    }
}
```

Dans le cas contraire, quand `ResourceType` est un type valeur Nullable ou un type référence autre que `dynamic` , le développement est

C#

```
{
    ResourceType resource = expression;
    try {
        statement;
    }
    finally {
        if (resource != null) ((IDisposable)resource).Dispose();
    }
}
```

Dans le cas contraire, lorsque `ResourceType` est `dynamic` , l'expansion est

C#

```
{
    ResourceType resource = expression;
    IDisposable d = (IDisposable)resource;
    try {
        statement;
    }
    finally {
        if (d != null) d.Dispose();
    }
}
```

Dans les deux expansions, la `resource` variable est en lecture seule dans l'instruction incorporée, et la `d` variable est inaccessible dans, et invisible à, l'instruction incorporée.

Une implémentation de est autorisée à implémenter une instruction using donnée différemment, par exemple pour des raisons de performances, tant que le comportement est cohérent avec l'expansion ci-dessus.

Une `using` déclaration de la forme

C#

```
using (expression) statement
```

a les mêmes trois expansions possibles. Dans ce cas `ResourceType` , est implicitement le type au moment de la compilation du `expression` , s'il en a un. Dans le cas contraire, l'interface `IDisposable` elle-même est utilisée comme `ResourceType` . La `resource` variable est inaccessible dans, et invisible à, l'instruction incorporée.

Lorsqu'un *resource_acquisition* prend la forme d'un *local_variable_declaration*, il est possible d'acquérir plusieurs ressources d'un type donné. Une `using` déclaration de la forme

C#

```
using (ResourceType r1 = e1, r2 = e2, ..., rN = eN) statement
```

équivaut précisément à une séquence d'instructions imbriquées `using` :

C#

```
using (ResourceType r1 = e1)
    using (ResourceType r2 = e2)
        ...
```

```
            using (ResourceType rN = eN)
                statement
```

L'exemple ci-dessous crée un fichier nommé `log.txt` et écrit deux lignes de texte dans le fichier. L'exemple ouvre ensuite le même fichier pour la lecture et copie les lignes de texte contenues dans la console.

C#

```
using System;
using System.IO;

class Test
{
    static void Main() {
        using (TextWriter w = File.CreateText("log.txt")) {
            w.WriteLine("This is line one");
            w.WriteLine("This is line two");
        }

        using (TextReader r = File.OpenText("log.txt")) {
            string s;
            while ((s = r.ReadLine()) != null) {
                Console.WriteLine(s);
            }

        }
    }
}
```

Étant donné que les `TextWriter` `TextReader` classes et implémentent l' `IDisposable` interface, l'exemple peut utiliser des `using` instructions pour s'assurer que le fichier sous-jacent est correctement fermé après les opérations d'écriture ou de lecture.

yield, instruction

L' `yield` instruction est utilisée dans un bloc itérateur (blocs) pour produire une valeur à l'objet énumérateur (objets énumérateur) ou à un objet énumérable (objets énumérables) d'un itérateur ou pour signaler la fin de l'itération.

antlr

```
yield_statement
    : 'yield' 'return' expression ';'
    | 'yield' 'break' ';'
    ;
```

`yield` n'est pas un mot réservé ; Il a une signification spéciale uniquement lorsqu'il est utilisé juste avant un `return` `break` mot clé ou. Dans d'autres contextes, `yield` peut être utilisé comme identificateur.

Il existe plusieurs restrictions sur l'emplacement où une `yield` instruction peut apparaître, comme décrit dans l'exemple suivant.

- Une erreur de compilation pour une `yield` instruction (de l'une des deux formes) s'affiche en dehors d'un *method_body*, *operator_body* ou *accessor_body*
- Il s'agit d'une erreur de compilation pour qu'une `yield` instruction (de l'une des deux formes) apparaisse dans une fonction anonyme.
- Il s'agit d'une erreur de compilation pour qu'une `yield` instruction (de l'une des deux formes) apparaisse dans la `finally` clause d'une `try` instruction.
- Il s'agit d'une erreur de compilation pour qu'une `yield return` instruction apparaisse n'importe où dans une `try` instruction qui contient des `catch` clauses.

L'exemple suivant montre des utilisations valides et non valides d' `yield` instructions.

C#

```
delegate IEnumerable<int> D();

IEnumerator<int> GetEnumerator() {
    try {
        yield return 1;        // Ok
        yield break;           // Ok
    }
    finally {
        yield return 2;        // Error, yield in finally
        yield break;           // Error, yield in finally
    }

    try {
        yield return 3;        // Error, yield return in try...catch
        yield break;           // Ok
    }
    catch {
        yield return 4;        // Error, yield return in try...catch
        yield break;           // Ok
    }

    D d = delegate {
        yield return 5;        // Error, yield in an anonymous function
    };
}

int MyMethod() {
    yield return 1;            // Error, wrong return type for an iterator
```

```
block
}
```

Une conversion implicite (conversions implicites) doit exister à partir du type de l'expression dans l' `yield return` instruction vers le type Yield (type yield) de l'itérateur.

Une `yield return` instruction est exécutée comme suit :

- L'expression donnée dans l'instruction est évaluée, implicitement convertie en type yield et assignée à la `Current` propriété de l'objet énumérateur.
- L'exécution du bloc itérateur est suspendue. Si l' `yield return` instruction se trouve dans un ou plusieurs `try` blocs, les `finally` blocs associés ne sont pas exécutés pour l'instant.
- La `MoveNext` méthode de l'objet énumérateur retourne `true` à son appelant, ce qui indique que l'objet énumérateur a réussi à avancer jusqu'à l'élément suivant.

Le prochain appel à la méthode de l'objet énumérateur `MoveNext` reprend l'exécution du bloc itérateur à partir de l'emplacement où il a été suspendu pour la dernière fois.

Une `yield break` instruction est exécutée comme suit :

- Si l' `yield break` instruction est entourée d'un ou de plusieurs `try` blocs avec des `finally` blocs associés, le contrôle est initialement transféré au `finally` bloc de l'instruction la plus profonde `try` . Quand et si le contrôle atteint le point de terminaison d'un `finally` bloc, le contrôle est transféré au `finally` bloc de l'instruction englobante suivante `try` . Ce processus est répété jusqu'à ce que les `finally` blocs de toutes les `try` instructions englobantes aient été exécutés.
- Le contrôle est retourné à l'appelant du bloc Iterator. Il s'agit `MoveNext` de la méthode ou `Dispose` de la méthode de l'objet énumérateur.

Étant donné qu'une `yield break` instruction transfère sans condition le contrôle ailleurs, le point de terminaison d'une `yield break` instruction n'est jamais accessible.

Espaces de noms

Article • 13/05/2021

Les programmes C# sont organisés à l'aide d'espaces de noms. Les espaces de noms sont utilisés à la fois en tant que système d'organisation « interne » pour un programme, et en tant que système d'organisation « externe », ce qui permet de présenter les éléments de programme qui sont exposés à d'autres programmes.

Les directives using (à l'aide de directives) sont fournies pour faciliter l'utilisation des espaces de noms.

Unités de compilation

Un *compilation_unit* définit la structure globale d'un fichier source. Une unité de compilation se compose de zéro, un ou plusieurs *using_directives* suivis de zéro, un ou plusieurs *global_attributes* suivis de zéro, un ou plusieurs *namespace_member_declaration* s.

antlr

```
compilation_unit
    : extern_alias_directive* using_directive* global_attributes?
namespace_member_declaration*
    ;
```

Un programme C# se compose d'une ou de plusieurs unités de compilation, chacune contenue dans un fichier source séparé. Lorsqu'un programme C# est compilé, toutes les unités de compilation sont traitées ensemble. Par conséquent, les unités de compilation peuvent dépendre l'une de l'autre, éventuellement de manière circulaire.

Les *using_directive* s d'une unité de compilation affectent les *global_attributes* et les *namespace_member_declaration* s de cette unité de compilation, mais n'ont aucun effet sur les autres unités de compilation.

Les *global_attributes* (attributs) d'une unité de compilation autorisent la spécification des attributs de l'assembly cible et du module. Les assemblys et les modules jouent le rôle de conteneurs physiques pour les types. Un assembly peut être constitué de plusieurs modules physiquement séparés.

Les *namespace_member_declaration* s de chaque unité de compilation d'un programme contribuent aux membres d'un espace de déclaration unique appelé espace de noms global. Par exemple :

Fichier `A.cs` :

```C#
class A {}
```

Fichier `B.cs` :

```C#
class B {}
```

Les deux unités de compilation contribuent à l'espace de noms global unique, dans ce cas, en déclarant deux classes avec les noms qualifiés complets `A` et `B` . Étant donné que les deux unités de compilation contribuent au même espace de déclaration, il s'agit d'une erreur si chacune contient une déclaration d'un membre portant le même nom.

Déclarations d'espace de noms

Un *namespace_declaration* se compose du mot clé `namespace` , suivi d'un nom et d'un corps d'espace de noms, éventuellement suivi d'un point-virgule.

```antlr
namespace_declaration
    : 'namespace' qualified_identifier namespace_body ';'?
    ;

qualified_identifier
    : identifier ('.' identifier)*
    ;

namespace_body
    : '{' extern_alias_directive* using_directive*
namespace_member_declaration* '}'
    ;
```

Une *namespace_declaration* peut se produire comme une déclaration de niveau supérieur dans un *compilation_unit* ou comme une déclaration de membre dans un autre *namespace_declaration*. Lorsqu'un *namespace_declaration* se produit comme une déclaration de niveau supérieur dans un *compilation_unit*, l'espace de noms devient un membre de l'espace de noms global. Lorsqu'un *namespace_declaration* se produit dans une autre *namespace_declaration*, l'espace de noms interne devient un membre de

l'espace de noms externe. Dans les deux cas, le nom d'un espace de noms doit être unique au sein de l'espace de noms conteneur.

Les espaces de noms sont implicitement `public` et la déclaration d'un espace de noms ne peut pas inclure de modificateurs d'accès.

Dans un *namespace_body*, la *using_directive* facultatives importent les noms d'autres espaces de noms, types et membres, ce qui leur permet d'être référencés directement plutôt que par le biais de noms qualifiés. Les *namespace_member_declaration* facultatives apportent des membres à l'espace de déclaration de l'espace de noms. Notez que tous les *using_directive* s doivent apparaître avant les déclarations de membre.

La *qualified_identifier* d'un *namespace_declaration* peut être un identificateur unique ou une séquence d'identificateurs séparés par des `.` jetons «». Cette dernière forme permet à un programme de définir un espace de noms imbriqué sans imbrication lexicale de plusieurs déclarations d'espaces de noms. Par exemple,

C#

```
namespace N1.N2
{
    class A {}

    class B {}
}
```

est sémantiquement équivalent à

C#

```
namespace N1
{
    namespace N2
    {
        class A {}

        class B {}
    }
}
```

Les espaces de noms sont ouverts et les deux déclarations d'espaces de noms portant le même nom qualifié complet participent au même espace de déclaration (déclarations). Dans l'exemple

C#

```
namespace N1.N2
{
    class A {}
}

namespace N1.N2
{
    class B {}
}
```

les deux déclarations d'espaces de noms ci-dessus contribuent au même espace de déclaration, dans ce cas, en déclarant deux classes avec les noms qualifiés complets `N1.N2.A` et `N1.N2.B` . Étant donné que les deux déclarations contribuent au même espace de déclaration, il s'agit d'une erreur si chacune contient une déclaration d'un membre portant le même nom.

Alias externes

Une *extern_alias_directive* introduit un identificateur qui sert d'alias pour un espace de noms. La spécification de l'espace de noms alias est externe au code source du programme et s'applique également aux espaces de noms imbriqués de l'espace de noms avec alias.

antlr

```
extern_alias_directive
    : 'extern' 'alias' identifier ';'
    ;
```

La portée d'un *extern_alias_directive* s'étend sur le *using_directive* s, *global_attributes* et *namespace_member_declaration* s de son conteneur d'unité de compilation ou d'espace de noms qui le contient immédiatement.

Dans une unité de compilation ou un corps d'espace de noms qui contient un *extern_alias_directive*, l'identificateur introduit par l' *extern_alias_directive* peut être utilisé pour référencer l'espace de noms avec alias. Il s'agit d'une erreur au moment de la compilation pour que l' *identificateur* soit le mot `global` .

Un *extern_alias_directive* rend un alias disponible dans une unité de compilation ou un corps d'espace de noms particulier, mais il ne fournit pas de nouveaux membres à l'espace de déclaration sous-jacent. En d'autres termes, un *extern_alias_directive* n'est pas transitif, mais affecte uniquement l'unité de compilation ou le corps de l'espace de noms dans lequel il se produit.

Le programme suivant déclare et utilise deux alias extern `X` et `Y` , chacun représentant la racine d'une hiérarchie d'espaces de noms distincte :

C#

```
extern alias X;
extern alias Y;

class Test
{
    X::N.A a;
    X::N.B b1;
    Y::N.B b2;
    Y::N.C c;
}
```

Le programme déclare l'existence des alias extern `X` et `Y` , mais les définitions réelles des alias sont externes au programme. Les classes portant le même nom `N.B` peuvent désormais être référencées en tant que `X.N.B` et `Y.N.B` , ou, à l'aide du qualificateur d'alias d'espace de noms, `X::N.B` et `Y::N.B` . Une erreur se produit si un programme déclare un alias extern pour lequel aucune définition externe n'est fournie.

using, directives

*Les **directives using** _ facilitent l'utilisation d'espaces de noms et de types définis dans d'autres espaces de noms. L'utilisation de directives a un impact sur le processus de résolution de noms de _namespace_or_type_name * s (noms d'espaces de noms et de types) et *simple_name* s (noms simples), mais contrairement aux déclarations, les directives using ne contribuent pas aux nouveaux membres dans les espaces de déclaration sous-jacents des unités de compilation ou des espaces de noms dans lesquels elles sont utilisées.

antlr

```
using_directive
    : using_alias_directive
    | using_namespace_directive
    | using_static_directive
    ;
```

Une *using_alias_directive* (à l'aide de directives d'alias) introduit un alias pour un espace de noms ou un type.

Une *using_namespace_directive* (à l'aide de directives d'espace de noms) importe les membres de type d'un espace de noms.

Une *using_static_directive* (à l'aide de directives statiques) importe les types imbriqués et les membres statiques d'un type.

La portée d'un *using_directive* s'étend sur le *namespace_member_declaration* s de son conteneur d'unité de compilation ou d'espace de noms qui le contient immédiatement. La portée d'un *using_directive* n'inclut pas son homologue *using_directive* s. Par conséquent, l'homologue *using_directive* s n'affectent pas les uns les autres et l'ordre dans lequel ils sont écrits n'est pas significatif.

Utilisation de directives d'alias

Une *using_alias_directive* introduit un identificateur qui sert d'alias pour un espace de noms ou un type dans le corps de l'unité de compilation ou de l'espace de noms englobant immédiatement.

antlr

```
using_alias_directive
    : 'using' identifier '=' namespace_or_type_name ';'
    ;
```

Dans les déclarations de membre d'une unité de compilation ou d'un corps d'espace de noms qui contient un *using_alias_directive*, l'identificateur introduit par l' *using_alias_directive* peut être utilisé pour référencer l'espace de noms ou le type donné. Par exemple :

C#

```
namespace N1.N2
{
    class A {}
}

namespace N3
{
    using A = N1.N2.A;

    class B: A {}
}
```

Ci-dessus, dans les déclarations de membre de l' `N3` espace de noms, `A` est un alias pour `N1.N2.A` , et par conséquent, la classe `N3.B` dérive de la classe `N1.N2.A` . Le même

effet peut être obtenu en créant un alias `R` pour `N1.N2` , puis en référençant `R.A` :

C#

```
namespace N3
{
    using R = N1.N2;

    class B: R.A {}
}
```

L' *identificateur* d'un *using_alias_directive* doit être unique dans l'espace de déclaration de l'unité de compilation ou de l'espace de noms qui contient immédiatement le *using_alias_directive*. Par exemple :

C#

```
namespace N3
{
    class A {}
}

namespace N3
{
    using A = N1.N2.A;        // Error, A already exists
}
```

Ci-dessus, `N3` contient déjà un membre `A` . il s'agit donc d'une erreur au moment de la compilation pour qu'un *using_alias_directive* utilise cet identificateur. De même, il s'agit d'une erreur de compilation pour deux ou plusieurs *using_alias_directive* s dans la même unité de compilation ou le même corps d'espace de noms pour déclarer des alias portant le même nom.

Un *using_alias_directive* rend un alias disponible dans une unité de compilation ou un corps d'espace de noms particulier, mais il ne fournit pas de nouveaux membres à l'espace de déclaration sous-jacent. En d'autres termes, un *using_alias_directive* n'est pas transitif, mais affecte uniquement l'unité de compilation ou le corps de l'espace de noms dans lequel il se produit. Dans l'exemple

C#

```
namespace N3
{
    using R = N1.N2;
}

namespace N3
```

```
{
    class B: R.A {}            // Error, R unknown
}
```

la portée de la *using_alias_directive* qui introduit `R` uniquement s'étend aux déclarations de membre dans le corps de l'espace de noms dans lequel elle est contenue, donc `R` est inconnu dans la deuxième déclaration d'espace de noms. Toutefois, le fait de placer les *using_alias_directive* dans l'unité de compilation conteneur provoque la disponibilité de l'alias dans les deux déclarations d'espaces de noms :

C#

```
using R = N1.N2;

namespace N3
{
    class B: R.A {}
}

namespace N3
{
    class C: R.A {}
}
```

Tout comme les membres normaux, les noms introduits par *using_alias_directive* s sont masqués par des membres portant le même nom dans des portées imbriquées. Dans l'exemple

C#

```
using R = N1.N2;

namespace N3
{
    class R {}

    class B: R.A {}        // Error, R has no member A
}
```

la référence à `R.A` dans la déclaration de `B` provoque une erreur au moment de la compilation, car `R` fait référence à `N3.R` , et non à `N1.N2` .

L'ordre dans lequel les *using_alias_directive* s sont écrits n'a pas d'importance, et la résolution des *namespace_or_type_name* référencés par un *using_alias_directive* n'est pas affectée par le *using_alias_directive* lui-même ou par d'autres *using_directive* dans le corps de l'unité de compilation ou de l'espace de noms qui le contient immédiatement.

En d'autres termes, la *namespace_or_type_name* d'un *using_alias_directive* est résolue comme si le corps de l'unité de compilation ou de l'espace de noms contenant immédiatement n'avait aucun *using_directive* s. Un *using_alias_directive* peut toutefois être affecté par *extern_alias_directive* s dans le corps de l'unité de compilation ou de l'espace de noms contenant immédiatement. Dans l'exemple

C#

```
namespace N1.N2 {}

namespace N3
{
    extern alias E;

    using R1 = E.N;         // OK

    using R2 = N1;          // OK

    using R3 = N1.N2;       // OK

    using R4 = R2.N2;       // Error, R2 unknown
}
```

la dernière *using_alias_directive* génère une erreur de compilation, car elle n'est pas affectée par le premier *using_alias_directive*. La première *using_alias_directive* ne génère pas d'erreur, car l'étendue de l'alias extern `E` comprend l' *using_alias_directive*.

Un *using_alias_directive* pouvez créer un alias pour un espace de noms ou un type, y compris l'espace de noms dans lequel il apparaît et tout espace de noms ou type imbriqué dans cet espace de noms.

L'accès à un espace de noms ou à un type via un alias produit exactement le même résultat que l'accès à cet espace de noms ou à un type via son nom déclaré. Par exemple, donné

C#

```
namespace N1.N2
{
    class A {}
}

namespace N3
{
    using R1 = N1;
    using R2 = N1.N2;

    class B
```

```
    {
        N1.N2.A a;              // refers to N1.N2.A
        R1.N2.A b;              // refers to N1.N2.A
        R2.A c;                 // refers to N1.N2.A
    }
}
```

les noms `N1.N2.A` , `R1.N2.A` et `R2.A` sont équivalents et tous font référence à la classe dont le nom qualifié complet est `N1.N2.A` .

L'utilisation d'alias peut nommer un type construit fermé, mais ne peut pas nommer une déclaration de type générique indépendante sans fournir d'arguments de type. Par exemple :

C#

```
namespace N1
{
    class A<T>
    {
        class B {}
    }
}

namespace N2
{
    using W = N1.A;          // Error, cannot name unbound generic type

    using X = N1.A.B;        // Error, cannot name unbound generic type

    using Y = N1.A<int>;     // Ok, can name closed constructed type

    using Z<T> = N1.A<T>;    // Error, using alias cannot have type
parameters
}
```

Utilisation de directives d'espace de noms

Une *using_namespace_directive* importe les types contenus dans un espace de noms dans l'unité de compilation ou le corps d'espace de noms englobant immédiatement, ce qui permet l'utilisation de l'identificateur de chaque type sans qualification.

antlr

```
using_namespace_directive
    : 'using' namespace_name ';'
    ;
```

Dans les déclarations de membre d'une unité de compilation ou d'un corps d'espace de noms qui contient une *using_namespace_directive*, les types contenus dans l'espace de noms donné peuvent être référencés directement. Par exemple :

C#

```
namespace N1.N2
{
    class A {}
}

namespace N3
{
    using N1.N2;

    class B: A {}
}
```

Dans les déclarations de membre de l'espace de noms ci-dessus, `N3` les membres de type de `N1.N2` sont directement disponibles et, par conséquent, la classe `N3.B` dérive de la classe `N1.N2.A` .

Une *using_namespace_directive* importe les types contenus dans l'espace de noms donné, mais n'importe pas spécifiquement les espaces de noms imbriqués. Dans l'exemple

C#

```
namespace N1.N2
{
    class A {}
}

namespace N3
{
    using N1;

    class B: N2.A {}        // Error, N2 unknown
}
```

le *using_namespace_directive* importe les types contenus dans `N1` , mais pas les espaces de noms imbriqués dans `N1` . Ainsi, la référence à `N2.A` dans la déclaration de `B` génère une erreur de compilation, car aucun membre nommé `N2` n'est dans la portée.

Contrairement à un *using_alias_directive*, un *using_namespace_directive* peut importer des types dont les identificateurs sont déjà définis dans l'unité de compilation ou le corps de l'espace de noms englobant. En effet, les noms importés par un

using_namespace_directive sont masqués par des membres portant le même nom dans l'unité de compilation ou le corps de l'espace de noms englobant. Par exemple :

C#

```
namespace N1.N2
{
    class A {}

    class B {}
}

namespace N3
{
    using N1.N2;

    class A {}
}
```

Ici, dans les déclarations de membre de l' `N3` espace de noms, `A` fait référence à `N3.A` plutôt qu'à `N1.N2.A` .

Quand plusieurs espaces de noms ou types importés par *using_namespace_directive* s ou *using_static_directive* s dans la même unité de compilation ou le même corps d'espace de noms contiennent des types portant le même nom, les références à ce nom en tant que *type_name* sont considérées comme ambiguës. Dans l'exemple

C#

```
namespace N1
{
    class A {}
}

namespace N2
{
    class A {}
}

namespace N3
{
    using N1;

    using N2;

    class B: A {}               // Error, A is ambiguous
}
```

`N1` et contiennent tous deux `N2` un membre `A` , et comme `N3` les importations, le référencement `A` dans `N3` est une erreur de compilation. Dans ce cas, le conflit peut être résolu par la qualification de références à `A` , ou en introduisant une *using_alias_directive* qui choisit un particulier `A` . Par exemple :

C#

```
namespace N3
{
    using N1;

    using N2;

    using A = N1.A;

    class B: A {}               // A means N1.A
}
```

En outre, lorsque plusieurs espaces de noms ou types importés par *using_namespace_directive* s ou *using_static_directive* s dans la même unité de compilation ou le même corps d'espace de noms contiennent des types ou des membres portant le même nom, les références à ce nom en tant que *simple_name* sont considérées comme ambiguës. Dans l'exemple

C#

```
namespace N1
{
    class A {}
}

class C
{
    public static int A;
}

namespace N2
{
    using N1;
    using static C;

    class B
    {
        void M()
        {
            A a = new A();   // Ok, A is unambiguous as a type-name
            A.Equals(2);     // Error, A is ambiguous as a simple-name
        }
    }
}
```

`N1` contient un membre de type `A` et `C` contient un champ statique `A` , et parce que importe à la `N2` fois, la référencement `A` en tant que *simple_name* est ambiguë et une erreur de compilation.

À l'instar d'un *using_alias_directive*, un *using_namespace_directive* ne contribue pas aux nouveaux membres de l'espace de déclaration sous-jacent de l'unité de compilation ou de l'espace de noms, mais affecte uniquement l'unité de compilation ou le corps de l'espace de noms dans lequel il apparaît.

Le *namespace_name* référencé par un *using_namespace_directive* est résolu de la même façon que le *namespace_or_type_name* référencé par un *using_alias_directive*. Ainsi, *using_namespace_directive* s dans la même unité de compilation ou le même corps d'espace de noms n'affectent pas l'un l'autre et peut être écrite dans n'importe quel ordre.

Utilisation de directives statiques

Une *using_static_directive* importe les types imbriqués et les membres statiques contenus directement dans une déclaration de type dans l'unité de compilation ou le corps de l'espace de noms englobant immédiatement, ce qui permet d'utiliser l'identificateur de chaque membre et type sans qualification.

antlr

```
using_static_directive
    : 'using' 'static' type_name ';'
    ;
```

Dans les déclarations de membre d'une unité de compilation ou d'un corps d'espace de noms qui contient une *using_static_directive*, les types imbriqués accessibles et les membres statiques (à l'exception des méthodes d'extension) contenus directement dans la déclaration du type donné peuvent être référencés directement. Par exemple :

C#

```
namespace N1
{
    class A
    {
        public class B{}
        public static B M(){ return new B(); }
    }
}
```

```
namespace N2
{
    using static N1.A;
    class C
    {
        void N() { B b = M(); }
    }
}
```

Ci-dessus, dans les déclarations de membre de l' `N2` espace de noms, les membres statiques et les types imbriqués de `N1.A` sont directement disponibles, et la méthode peut donc `N` référencer à la fois les `B` `M` membres et de `N1.A` .

Un *using_static_directive* spécifiquement n'importe pas les méthodes d'extension directement en tant que méthodes statiques, mais les rend disponibles pour l'appel de méthode d'extension (appels de méthode d'extension). Dans l'exemple

C#

```
namespace N1
{
    static class A
    {
        public static void M(this string s){}
    }
}

namespace N2
{
    using static N1.A;

    class B
    {
        void N()
        {
            M("A");      // Error, M unknown
            "B".M();     // Ok, M known as extension method
            N1.A.M("C"); // Ok, fully qualified
        }
    }
}
```

le *using_static_directive* importe la méthode `M` d'extension contenue dans `N1.A` , mais uniquement en tant que méthode d'extension. Ainsi, la première référence à `M` dans le corps de `B.N` génère une erreur au moment de la compilation, car aucun membre nommé `M` n'est dans la portée.

Un *using_static_directive* importe uniquement les membres et les types déclarés directement dans le type donné, pas les membres et les types déclarés dans les classes de base.

TODO : exemple

Les ambiguïtés entre plusieurs *using_namespace_directives* et *using_static_directives* sont abordées dans utilisation de directives d' espace de noms.

Membres d'espaces de noms

Une *namespace_member_declaration* est une *namespace_declaration* (déclarations d'espace de noms) ou une *type_declaration* (déclarations de type).

antlr

```
namespace_member_declaration
    : namespace_declaration
    | type_declaration
    ;
```

Une unité de compilation ou un corps d'espace de noms peut contenir *namespace_member_declaration* s, et ces déclarations apportent de nouveaux membres à l'espace de déclaration sous-jacent de l'unité de compilation ou du corps de l'espace de noms conteneur.

Déclarations de type

Une *type_declaration* est une *class_declaration* (déclarations de classe), une *struct_declaration* (déclarations de struct), une *interface_declaration* (déclarations d'interface), une *enum_declaration* (déclarations d'énumération) ou une *delegate_declaration* (déclarations de délégué).

antlr

```
type_declaration
    : class_declaration
    | struct_declaration
    | interface_declaration
    | enum_declaration
    | delegate_declaration
    ;
```

Une *type_declaration* peut se produire comme une déclaration de niveau supérieur dans une unité de compilation ou comme une déclaration de membre dans un espace de noms, une classe ou un struct.

Lorsqu'une déclaration de type pour un type `T` se produit comme déclaration de niveau supérieur dans une unité de compilation, le nom qualifié complet du type nouvellement déclaré est simplement `T` . Lorsqu'une déclaration de type pour un type `T` se produit dans un espace de noms, une classe ou un struct, le nom qualifié complet du type nouvellement déclaré est `N.T` , où `N` est le nom qualifié complet de l'espace de noms, de la classe ou de la structure conteneur.

Un type déclaré dans une classe ou un struct est appelé type imbriqué (types imbriqués).

Les modificateurs d'accès autorisés et l'accès par défaut pour une déclaration de type dépendent du contexte dans lequel la déclaration a lieu (accessibilité déclarée) :

- Les types déclarés dans les unités de compilation ou les espaces de noms peuvent avoir `public` ou `internal` accéder à. La valeur par défaut est `internal` Access.
- Les types déclarés dans les classes peuvent avoir un `public` accès,,, `protected internal` `protected` `internal` ou `private` . La valeur par défaut est `private` Access.
- Les types déclarés dans des structs peuvent avoir un `public` `internal` accès, ou `private` . La valeur par défaut est `private` Access.

Qualificateurs d'alias d'espace de noms

Le ***qualificateur d'alias*** `::` d'espace de noms permet de garantir que les recherches de noms de types ne sont pas affectées par l'introduction de nouveaux types et membres. Le qualificateur d'alias d'espace de noms apparaît toujours entre deux identificateurs désignés comme identificateurs de gauche et de droite. Contrairement au `.` qualificateur normal, l'identificateur de gauche du `::` qualificateur est recherché uniquement comme un alias extern ou using.

Une *qualified_alias_member* est définie comme suit :

antlr

```
qualified_alias_member
    : identifier '::' identifier type_argument_list?
    ;
```

Un *qualified_alias_member* peut être utilisé comme *namespace_or_type_name* (noms d'espaces de noms et de types) ou comme opérande gauche dans un *member_access* (accès aux membres).

Un *qualified_alias_member* a l'une des deux formes suivantes :

- `N::I<A1, ..., Ak>`, où `N` et `I` représentent des identificateurs, et `<A1, ..., Ak>` est une liste d'arguments de type. (`K` est toujours au moins un.)
- `N::I`, où `N` et `I` représentent des identificateurs. (Dans ce cas, `K` est considéré comme égal à zéro.)

À l'aide de cette notation, la signification d'un *qualified_alias_member* est déterminée comme suit :

- Si `N` est l'identificateur `global` , l'espace de noms global est recherché `I` :
  - Si l'espace de noms global contient un espace de noms nommé `I` et `K` que est égal à zéro, le *qualified_alias_member* fait référence à cet espace de noms.
  - Sinon, si l'espace de noms global contient un type non générique nommé `I` et `K` est égal à zéro, le *qualified_alias_member* fait référence à ce type.
  - Sinon, si l'espace de noms global contient un type nommé `I` qui a des `K` paramètres de type, alors le *qualified_alias_member* fait référence à ce type construit avec les arguments de type donnés.
  - Dans le cas contraire, la *qualified_alias_member* n'est pas définie et une erreur de compilation se produit.

- Dans le cas contraire, en commençant par la déclaration d'espace de noms (déclarations d'espace de noms) contenant immédiatement le *qualified_alias_member* (le cas échéant), en continuant avec chaque déclaration d'espace de noms englobant (le cas échéant) et en terminant par l'unité de compilation contenant le *qualified_alias_member*, les étapes suivantes sont évaluées jusqu'à ce qu'une entité soit localisée :
  - Si la déclaration d'espace de noms ou l'unité de compilation contient une *using_alias_directive* qui associe `N` à un type, le *qualified_alias_member* n'est pas défini et une erreur de compilation se produit.
  - Sinon, si la déclaration d'espace de noms ou l'unité de compilation contient une *extern_alias_directive* ou *using_alias_directive* qui associe `N` à un espace de noms, procédez comme suit :
    - Si l'espace de noms associé à `N` contient un espace de noms nommé `I` et `K` est égal à zéro, le *qualified_alias_member* fait référence à cet espace de noms.

    - Sinon, si l'espace de noms associé à `N` contient un type non générique nommé `I` et `K` est égal à zéro, le *qualified_alias_member* fait référence à ce type.
    - Sinon, si l'espace de noms associé à `N` contient un type nommé `I` qui a `K` des paramètres de type, alors le *qualified_alias_member* fait référence à ce type construit avec les arguments de type donnés.
    - Dans le cas contraire, la *qualified_alias_member* n'est pas définie et une erreur de compilation se produit.

- Dans le cas contraire, la *qualified_alias_member* n'est pas définie et une erreur de compilation se produit.

Notez que l'utilisation du qualificateur d'alias d'espace de noms avec un alias qui référence un type provoque une erreur au moment de la compilation. Notez également que si l'identificateur `N` est `global` , la recherche est effectuée dans l'espace de noms global, même s'il existe un alias using qui associe `global` à un type ou un espace de noms.

Unicité des alias

Chaque unité de compilation et corps d'espace de noms possède un espace de déclaration distinct pour les alias extern et l'utilisation d'alias. Ainsi, bien que le nom d'un alias extern ou d'un alias doive être unique au sein du jeu d'alias extern et qu'il utilise des alias déclarés dans le corps de l'unité de compilation ou de l'espace de noms qui le contient immédiatement, un alias peut avoir le même nom qu'un type ou un espace de noms, à condition qu'il soit utilisé uniquement avec le `::` qualificateur.

Dans l'exemple

C#

```
namespace N
{
    public class A {}

    public class B {}
}

namespace N
{
    using A = System.IO;

    class X
    {
        A.Stream s1;            // Error, A is ambiguous
```

```
        A::Stream s2;           // Ok
    }
}
```

le nom `A` a deux significations possibles dans le deuxième corps de l'espace de noms, car la classe `A` et l'alias using `A` sont dans la portée. Pour cette raison, l'utilisation de `A` dans le nom qualifié `A.Stream` est ambiguë et provoque une erreur au moment de la compilation. Toutefois, l'utilisation de `A` avec le `::` qualificateur n'est pas une erreur, car `A` est recherché uniquement comme alias d'espace de noms.

Classes

Article • 13/05/2021

Une classe est une structure de données qui peut contenir des données membres (constantes et champs), des fonctions membres (méthodes, propriétés, événements, indexeurs, opérateurs, constructeurs d'instance, destructeurs et constructeurs statiques) et des types imbriqués. Les types de classe prennent en charge l'héritage, mécanisme par lequel une classe dérivée peut étendre et spécialiser une classe de base.

Déclarations de classe

Un *class_declaration* est un *type_declaration* (déclarations de type) qui déclare une nouvelle classe.

antlr

```
class_declaration
    : attributes? class_modifier* 'partial'? 'class' identifier
type_parameter_list?
      class_base? type_parameter_constraints_clause* class_body ';'?
    ;
```

Un *class_declaration* se compose d'un ensemble facultatif d' *attributs* (attributs), suivi d'un ensemble facultatif de *class_modifier* s (modificateurs de classe). suivi d'un `partial` modificateur facultatif, suivi du mot clé `class` et d'un *identificateur* qui nomme la classe, suivi d'un *type_parameter_list* facultatif (paramètres de type), suivi d'une spécification facultative *class_base* (spécification de base de classe), suivi d'un ensemble facultatif de *type_parameter_constraints_clause* s (contraintes de paramètre de type), suivi d'un *class_body* (corps de classe), éventuellement suivi d'un point-virgule.

Une déclaration de classe ne peut pas fournir *type_parameter_constraints_clause* s, sauf si elle fournit également une *type_parameter_list*.

Une déclaration de classe qui fournit un *type_parameter_list* est une ***déclaration de classe générique***. En outre, toute classe imbriquée dans une déclaration de classe générique ou une déclaration de struct générique est elle-même une déclaration de classe générique, car les paramètres de type pour le type conteneur doivent être fournis pour créer un type construit.

Modificateurs de classe

Une *class_declaration* peut éventuellement inclure une séquence de modificateurs de classe :

```antlr
class_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'abstract'
    | 'sealed'
    | 'static'
    | class_modifier_unsafe
    ;
```

Il s'agit d'une erreur de compilation pour que le même modificateur apparaisse plusieurs fois dans une déclaration de classe.

Le `new` modificateur est autorisé sur les classes imbriquées. Il spécifie que la classe masque un membre hérité du même nom, comme décrit dans le nouveau modificateur. Il s'agit d'une erreur de compilation pour `new` que le modificateur apparaisse sur une déclaration de classe qui n'est pas une déclaration de classe imbriquée.

Les `public` `protected` `internal` modificateurs,, et `private` contrôlent l'accessibilité de la classe. Selon le contexte dans lequel la déclaration de classe se produit, certains de ces modificateurs peuvent ne pas être autorisés (accessibilité déclarée).

Les `abstract` `sealed` `static` modificateurs, et sont décrits dans les sections suivantes.

Classes abstraites

Le `abstract` modificateur est utilisé pour indiquer qu'une classe est incomplète et qu'elle est destinée à être utilisée uniquement comme classe de base. Une classe abstraite diffère d'une classe non abstraite des façons suivantes :

- Une classe abstraite ne peut pas être instanciée directement, et il s'agit d'une erreur au moment de la compilation pour utiliser l' `new` opérateur sur une classe abstraite. Bien qu'il soit possible d'avoir des variables et des valeurs dont les types au moment de la compilation sont abstraits, ces variables et valeurs doivent être `null` ou contenir des références à des instances de classes non abstraites dérivées des types abstraits.

- Une classe abstraite est autorisée (mais pas obligatoire) à contenir des membres abstraits.
- Une classe abstraite ne peut pas être sealed.

Lorsqu'une classe non abstraite est dérivée d'une classe abstraite, la classe non abstraite doit inclure des implémentations réelles de tous les membres abstraits hérités, remplaçant ainsi ces membres abstraits. Dans l'exemple

C#

```
abstract class A
{
    public abstract void F();
}

abstract class B: A
{
    public void G() {}
}

class C: B
{
    public override void F() {
        // actual implementation of F
    }
}
```

la classe abstraite `A` introduit une méthode abstraite `F` . La classe `B` introduit une méthode supplémentaire `G` , mais puisqu'elle ne fournit pas d'implémentation de `F` , `B` doit également être déclarée comme abstract. La classe `C` se substitue à `F` et fournit une implémentation réelle. Étant donné qu'il n'y a aucun membre abstrait dans `C` , `C` est autorisé (mais pas obligatoire) à être non abstrait.

Classes sealed

Le `sealed` modificateur est utilisé pour empêcher la dérivation à partir d'une classe. Une erreur de compilation se produit si une classe sealed est spécifiée en tant que classe de base d'une autre classe.

Une classe sealed ne peut pas également être une classe abstraite.

Le `sealed` modificateur est principalement utilisé pour empêcher la dérivation involontaire, mais il permet également certaines optimisations au moment de l'exécution. En particulier, étant donné qu'une classe sealed est connue pour ne jamais

avoir de classes dérivées, il est possible de transformer des appels de membre de fonction virtuelle sur des instances de classe sealed en appels non virtuels.

Classes statiques

Le `static` modificateur est utilisé pour marquer la classe déclarée comme une ***classe statique***. Une classe statique ne peut pas être instanciée, elle ne peut pas être utilisée en tant que type et ne peut contenir que des membres statiques. Seule une classe statique peut contenir des déclarations de méthodes d'extension (méthodes d'extension).

Une déclaration de classe statique est soumise aux restrictions suivantes :

- Une classe statique ne peut pas inclure `sealed` un `abstract` modificateur ou. Notez, toutefois, qu'étant donné qu'une classe statique ne peut pas être instanciée ou dérivée de, elle se comporte comme si elle était à la fois sealed et abstract.
- Une classe statique ne peut pas inclure une spécification *class_base* (spécification de base de classe) et ne peut pas spécifier explicitement une classe de base ou une liste d'interfaces implémentées. Une classe statique hérite implicitement du type `object` .
- Une classe statique ne peut contenir que des membres statiques (membres statiques et d'instance). Notez que les constantes et les types imbriqués sont classés en tant que membres statiques.
- Une classe statique ne peut pas avoir de membres avec `protected` ou `protected internal` d'accessibilité déclarée.

Il s'agit d'une erreur au moment de la compilation pour enfreindre l'une de ces restrictions.

Une classe statique n'a pas de constructeurs d'instance. Il n'est pas possible de déclarer un constructeur d'instance dans une classe statique, et aucun constructeur d'instance par défaut (constructeurs par défaut) n'est fourni pour une classe statique.

Les membres d'une classe statique ne sont pas automatiquement statiques, et les déclarations de membre doivent inclure explicitement un `static` modificateur (à l'exception des constantes et des types imbriqués). Quand une classe est imbriquée dans une classe externe statique, la classe imbriquée n'est pas une classe statique, sauf si elle comprend explicitement un `static` modificateur.

Référencement des types de classes statiques

Une *namespace_or_type_name* (espaces de noms et noms de types) est autorisée à faire référence à une classe statique si

- Le *namespace_or_type_name* est `T` dans un *namespace_or_type_name* de la forme `T.I` , ou
- Le *namespace_or_type_name* est le `T` dans un *typeof_expression* (liste d'arguments1) du formulaire `typeof(T)` .

Une *primary_expression* (fonctions membres) est autorisée à référencer une classe statique si

- Le *primary_expression* est le `E` dans une *member_access* (vérification de la résolution de surcharge dynamique au momentde la compilation) du formulaire `E.I` .

Dans tout autre contexte, il s'agit d'une erreur de compilation pour référencer une classe statique. Par exemple, il s'agit d'une erreur pour une classe statique à utiliser comme classe de base, un type constitutif (types imbriqués) d'un membre, un argument de type générique ou une contrainte de paramètre de type. De même, une classe statique ne peut pas être utilisée dans un type de tableau, un type pointeur, une `new` expression, une expression de cast, une expression, une expression, `is` `as` une `sizeof` expression ou une expression de valeur par défaut.

Modificateur partiel

Le `partial` modificateur est utilisé pour indiquer que ce *class_declaration* est une déclaration de type partielle. Plusieurs déclarations de type partiel portant le même nom dans un espace de noms englobant ou une déclaration de type sont combinées pour former une déclaration de type, suivant les règles spécifiées dans les types partiels.

La déclaration d'une classe distribuée sur des segments distincts de texte de programme peut être utile si ces segments sont produits ou maintenus dans des contextes différents. Par exemple, une partie d'une déclaration de classe peut être générée par l'ordinateur, tandis que l'autre est créée manuellement. La séparation du texte des deux empêche les mises à jour par l'une d'un conflit avec les mises à jour par l'autre.

Paramètres de type

Un paramètre de type est un identificateur simple qui désigne un espace réservé pour un argument de type fourni pour créer un type construit. Un paramètre de type est un espace réservé formel pour un type qui sera fourni ultérieurement. En revanche, un argument de type (arguments de type) est le type réel qui remplace le paramètre de type lors de la création d'un type construit.

```
```

antlr

```
type_parameter_list
    : '<' type_parameters '>'
    ;

type_parameters
    : attributes? type_parameter
    | type_parameters ',' attributes? type_parameter
    ;

type_parameter
    : identifier
    ;
```

Chaque paramètre de type dans une déclaration de classe définit un nom dans l'espace de déclaration (déclarations) de cette classe. Par conséquent, il ne peut pas avoir le même nom qu'un autre paramètre de type ou un membre déclaré dans cette classe. Un paramètre de type ne peut pas avoir le même nom que le type lui-même.

Spécification de base de classe

Une déclaration de classe peut inclure une spécification *class_base* , qui définit la classe de base directe de la classe et les interfaces (interfaces) implémentées directement par la classe.

antlr

```
class_base
    : ':' class_type
    | ':' interface_type_list
    | ':' class_type ',' interface_type_list
    ;

interface_type_list
    : interface_type (',' interface_type)*
    ;
```

La classe de base spécifiée dans une déclaration de classe peut être un type de classe construit (types construits). Une classe de base ne peut pas être un paramètre de type seul, bien qu'il puisse impliquer les paramètres de type qui se trouvent dans la portée.

C#

```
class Extend<V>: V {}           // Error, type parameter used as base class
```

Classes de base

Lorsqu'un *class_type* est inclus dans le *class_base*, il spécifie la classe de base directe de la classe déclarée. Si une déclaration de classe n'a pas de *class_base*, ou si le *class_base* répertorie uniquement les types d'interface, la classe de base directe est supposée être `object` . Une classe hérite des membres de sa classe de base directe, comme décrit dans héritage.

Dans l'exemple

```
C#

class A {}

class B: A {}
```

`A`la classe est considérée comme la classe de base directe de `B` et `B` est dite dérivée de `A` . Étant donné que `A` ne spécifie pas explicitement une classe de base directe, sa classe de base directe est implicitement `object` .

Pour un type de classe construit, si une classe de base est spécifiée dans la déclaration de classe générique, la classe de base du type construit est obtenue en substituant, pour chaque *type_parameter* dans la déclaration de la classe de base, les *type_argument* correspondants du type construit. Compte tenu des déclarations de classes génériques

```
C#

class B<U,V> {...}

class G<T>: B<string,T[]> {...}
```

la classe de base du type construit `G<int>` est `B<string,int[]>` .

La classe de base directe d'un type de classe doit être au moins aussi accessible que le type de classe lui-même (domaines d'accessibilité). Par exemple, il s'agit d'une erreur de compilation pour qu'une `public` classe dérive d'une `private` `internal` classe ou.

La classe de base directe d'un type de classe ne doit pas être de l'un des types suivants : `System.Array` , `System.Delegate` ,, `System.MulticastDelegate` `System.Enum` ou `System.ValueType` . En outre, une déclaration de classe générique ne peut pas être utilisée `System.Attribute` en tant que classe de base directe ou indirecte.

Tout en déterminant la signification de la spécification directe de la classe `A` de base d'une classe `B` , la classe de base directe de `B` est temporairement supposée être

`object` . Intuitivement, cela garantit que la signification d'une spécification de classe de base ne peut pas dépendre de lui-même de manière récursive. L'exemple :

```
C#

class A<T> {
   public class B {}
}

class C : A<C.B> {}
```

est erroné, car dans la spécification de la classe de base `A<C.B>` , la classe de base directe de `C` est considérée comme étant `object` , et par conséquent (par les règles des noms de type et d'espace de noms) `C` n'est pas considérée comme ayant un membre `B` .

Les classes de base d'un type de classe sont la classe de base directe et ses classes de base. En d'autres termes, l'ensemble de classes de base est la fermeture transitive de la relation de la classe de base directe. En vous référant à l'exemple ci-dessus, les classes de base de `B` sont `A` et `object` . Dans l'exemple

```
C#

class A {...}

class B<T>: A {...}

class C<T>: B<IComparable<T>> {...}

class D<T>: C<T[]> {...}
```

les classes de base de `D<int>` sont `C<int[]>` ,, `B<IComparable<int[]>>` `A` et `object` .

À l'exception de `object` la classe, chaque type de classe a une seule classe de base directe. La `object` classe n'a pas de classe de base directe et est la classe de base fondamentale de toutes les autres classes.

Quand une classe `B` dérive d'une classe `A` , il s'agit d'une erreur de compilation pour que `A` dépende de `B` . Une classe ***dépend directement de*** _ *sa classe de base directe (le cas échéant) et _dépend directement de** _ la classe dans laquelle elle est immédiatement imbriquée (le cas échéant). Compte tenu de cette définition, l'ensemble complet des classes dont dépend une classe est la fermeture réflexive et transitive de la relation _ *dépend* de *.

L'exemple

C#

```
class A: A {}
```

est erroné, car la classe dépend elle-même. De même, l'exemple

C#

```
class A: B {}
class B: C {}
class C: A {}
```

est erroné, car les classes dépendent d'elles-mêmes de façon circulaire. Enfin, l'exemple

C#

```
class A: B.C {}

class B: A
{
    public class C {}
}
```

génère une erreur au moment de la compilation, car `A` dépend de `B.C` (sa classe de base directe), qui dépend de `B` (sa classe englobante immédiate), qui dépend de façon circulaire `A` .

Notez qu'une classe ne dépend pas des classes qui sont imbriquées dans celle-ci. Dans l'exemple

C#

```
class A
{
    class B: A {}
}
```

`B` dépend de `A` (car `A` est à la fois sa classe de base directe et sa classe englobante immédiate), mais `A` ne dépend pas de `B` (étant donné qu' `B` il ne s'agit ni d'une classe de base, ni d'une classe englobante de `A`). Ainsi, l'exemple est valide.

Il n'est pas possible de dériver d'une `sealed` classe. Dans l'exemple

C#

```
sealed class A {}

class B: A {}            // Error, cannot derive from a sealed class
```

`B`la classe est erronée, car elle tente de dériver de la `sealed` classe `A` .

Implémentations d'interfaces

Une spécification de *class_base* peut inclure une liste de types d'interfaces, auquel cas la classe est dite qui implémente directement les types d'interfaces donnés. Les implémentations d'interface sont abordées plus en détail dans implémentations d'interface.

Contraintes de paramètre de type

Les déclarations de type et de méthode génériques peuvent éventuellement spécifier des contraintes de paramètre de type en incluant *type_parameter_constraints_clause* s.

antlr

```
type_parameter_constraints_clause
    : 'where' type_parameter ':' type_parameter_constraints
    ;

type_parameter_constraints
    : primary_constraint
    | secondary_constraints
    | constructor_constraint
    | primary_constraint ',' secondary_constraints
    | primary_constraint ',' constructor_constraint
    | secondary_constraints ',' constructor_constraint
    | primary_constraint ',' secondary_constraints ','
constructor_constraint
    ;

primary_constraint
    : class_type
    | 'class'
    | 'struct'
    ;

secondary_constraints
    : interface_type
    | type_parameter
    | secondary_constraints ',' interface_type
    | secondary_constraints ',' type_parameter
    ;
```

```
constructor_constraint
    : 'new' '(' ')'
    ;
```

Chaque *type_parameter_constraints_clause* se compose du jeton `where` , suivi du nom d'un paramètre de type, suivi d'un signe deux-points et de la liste de contraintes pour ce paramètre de type. Il peut y avoir au plus une `where` clause pour chaque paramètre de type, et les `where` clauses peuvent être listées dans n'importe quel ordre. À l'instar des `get` `set` jetons et dans un accesseur de propriété, le `where` jeton n'est pas un mot clé.

La liste de contraintes donnée dans une `where` clause peut inclure l'un des composants suivants, dans cet ordre : une seule contrainte primaire, une ou plusieurs contraintes secondaires et la contrainte de constructeur, `new()` .

Une contrainte principale peut être un type de classe ou la contrainte de **type référence** _ _ `class` ou la *contrainte de type valeur* `struct` . Une contrainte secondaire peut être un _type_parameter * ou *INTERFACE_TYPE*.

La contrainte de type référence spécifie qu'un argument de type utilisé pour le paramètre de type doit être un type référence. Tous les types de classe, d'interface, de délégué, de tableau et de type connu comme étant un type référence (tel que défini ci-dessous) répondent à cette contrainte.

La contrainte de type valeur spécifie qu'un argument de type utilisé pour le paramètre de type doit être un type valeur n'acceptant pas les valeurs NULL. Tous les types struct non Nullable, les types ENUM et les paramètres de type ayant la contrainte de type valeur satisfont cette contrainte. Notez que, bien qu'il soit classifié comme un type valeur, un type Nullable (types Nullable) ne satisfait pas la contrainte de type valeur. Un paramètre de type ayant la contrainte de type valeur ne peut pas également avoir la *constructor_constraint*.

Les types de pointeurs ne sont jamais autorisés à être des arguments de type et ne sont pas considérés comme répondant aux contraintes de type référence ou de type valeur.

Si une contrainte est un type de classe, un type d'interface ou un paramètre de type, ce type spécifie un « type de base » minimal que chaque argument de type utilisé pour ce paramètre de type doit prendre en charge. Chaque fois qu'un type construit ou une méthode générique est utilisé, l'argument de type est comparé aux contraintes sur le paramètre de type au moment de la compilation. L'argument de type fourni doit satisfaire aux conditions décrites dans satisfaire les contraintes.

Une contrainte de *class_type* doit respecter les règles suivantes :

- Le type doit être un type de classe.
- Le type ne doit pas être `sealed` .
- Le type ne doit pas être l'un des types suivants : `System.Array` , `System.Delegate` , `System.Enum` ou `System.ValueType` .
- Le type ne doit pas être `object` . Étant donné que tous les types dérivent de `object` , une telle contrainte n'aura aucun effet si elle était autorisée.
- Au plus une contrainte pour un paramètre de type donné peut être un type de classe.

Un type spécifié en tant que contrainte de *INTERFACE_TYPE* doit respecter les règles suivantes :

- Le type doit être un type d'interface.
- Un type ne doit pas être spécifié plus d'une fois dans une `where` clause donnée.

Dans les deux cas, la contrainte peut impliquer l'un des paramètres de type du type ou de la déclaration de méthode associés dans le cadre d'un type construit et peut impliquer le type déclaré.

Tout type de classe ou d'interface spécifié en tant que contrainte de paramètre de type doit être au moins aussi accessible (contraintes d'accessibilité) que le type ou la méthode générique déclaré (e).

Un type spécifié en tant que contrainte de *type_parameter* doit respecter les règles suivantes :

- Le type doit être un paramètre de type.
- Un type ne doit pas être spécifié plus d'une fois dans une `where` clause donnée.

En outre, il ne doit y avoir aucun cycle dans le graphique de dépendance des paramètres de type, où la dépendance est une relation transitive définie par :

- Si un paramètre `T` de type est utilisé en tant que contrainte pour le paramètre de type `S` , `S` ***dépend de*** `T` .
- Si un paramètre de type `S` dépend d'un paramètre de type `T` et `T` dépend d'un paramètre de type, `U` `S` ***dépend de*** `U` .

Compte tenu de cette relation, il s'agit d'une erreur de compilation pour qu'un paramètre de type dépende lui-même (directement ou indirectement).

Toutes les contraintes doivent être cohérentes entre les paramètres de type dépendant. Si le paramètre `S` de type dépend du paramètre de type `T` , alors :

- `T` ne doit pas avoir la contrainte de type valeur. Dans le cas contraire, `T` est effectivement sealed, ce qui signifie qu'il doit `S` être du même type que, ce qui `T` élimine le besoin de deux paramètres de type.
- Si `S` a la contrainte de type valeur, il `T` ne doit pas avoir de contrainte *class_type* .
- Si `S` a une contrainte de *class_type* `A` et `T` a une contrainte de *class_type* `B` , il doit y avoir une conversion d'identité ou une conversion de référence implicite de `A` en `B` ou une conversion de référence implicite de `B` en `A` .
- Si `S` dépend également du paramètre de type `U` et `U` a une contrainte de *class_type* `A` et `T` a une contrainte de *class_type* , il doit y avoir une conversion d'identité ou une conversion de `B` référence implicite de `A` en `B` ou une conversion de référence implicite de `B` en `A` .

`S`La contrainte de type valeur et `T` la contrainte de type référence sont valides. Cela limite `T` en fait les types `System.Object` , `System.ValueType` , `System.Enum` et n'importe quel type d'interface.

Si la `where` clause d'un paramètre de type inclut une contrainte de constructeur (qui a la forme `new()`), il est possible d'utiliser l' `new` opérateur pour créer des instances du type (expressions de création d'objet). Tout argument de type utilisé pour un paramètre de type avec une contrainte de constructeur doit avoir un constructeur sans paramètre public (ce constructeur existe implicitement pour tout type valeur) ou être un paramètre de type ayant la contrainte de type valeur ou la contrainte de constructeur (pour plus d'informations, consultez contraintes de paramètre de type).

Voici quelques exemples de contraintes :

C#

```
interface IPrintable
{
    void Print();
}

interface IComparable<T>
{
    int CompareTo(T value);
}

interface IKeyProvider<T>
{
    T GetKey();
}

class Printer<T> where T: IPrintable {...}

class SortedList<T> where T: IComparable<T> {...}
```

```
class Dictionary<K,V>
    where K: IComparable<K>
    where V: IPrintable, IKeyProvider<K>, new()
{
    ...
}
```

L'exemple suivant est une erreur, car elle provoque une circularité dans le graphique de dépendance des paramètres de type :

C#

```
class Circular<S,T>
    where S: T
    where T: S                // Error, circularity in dependency graph
{
    ...
}
```

Les exemples suivants illustrent des situations non valides supplémentaires :

C#

```
class Sealed<S,T>
    where S: T
    where T: struct        // Error, T is sealed
{
    ...
}

class A {...}

class B {...}

class Incompat<S,T>
    where S: A, T
    where T: B                // Error, incompatible class-type constraints
{
    ...
}

class StructWithClass<S,T,U>
    where S: struct, T
    where T: U
    where U: A                // Error, A incompatible with struct
{
    ...
}
```

La ***classe de base effective*** d'un paramètre de type `T` est définie comme suit :

- Si n' `T` a pas de contraintes primaires ni de contraintes de paramètre de type, sa classe de base effective est `object` .
- Si `T` a la contrainte de type valeur, sa classe de base effective est `System.ValueType` .
- Si `T` a une contrainte de *class_type* `C` , mais pas de contraintes de *type_parameter* , sa classe de base effective est `C` .
- Si n' `T` a pas de contrainte de *class_type* mais possède une ou plusieurs contraintes *type_parameter* , sa classe de base effective est le type le plus englobé (les opérateurs de conversion levés) dans le jeu de classes de base effectives de ses contraintes de *type_parameter* . Les règles de cohérence garantissent qu'il existe un type le plus englobé.
- Si `T` a à la fois une contrainte de *class_type* et une ou plusieurs contraintes de *type_parameter* , sa classe de base effective est le type le plus englobé (les opérateurs de conversion levés) dans le jeu qui se compose de la contrainte *class_type* de `T` et des classes de base effectives de ses *type_parameter* contraintes. Les règles de cohérence garantissent qu'il existe un type le plus englobé.
- Si `T` a la contrainte de type référence, mais pas de contraintes de *class_type* , sa classe de base effective est `object` .

Dans le cadre de ces règles, si T a une contrainte `V` qui est un *Value_type*, utilisez à la place le type de base le plus spécifique de `V` qui est un *class_type*. Cela ne peut jamais se produire dans une contrainte donnée explicitement, mais peut se produire lorsque les contraintes d'une méthode générique sont implicitement héritées par une déclaration de méthode de substitution ou une implémentation explicite d'une méthode d'interface.

Ces règles garantissent que la classe de base effective est toujours une *class_type*.

L' ***ensemble d'interfaces effectif*** d'un paramètre de type `T` est défini comme suit :

- Si n' `T` a pas de *secondary_constraints*, son ensemble d'interfaces effectif est vide.
- Si `T` a *INTERFACE_TYPE* contraintes, mais pas de contraintes *type_parameter* , son ensemble d'interfaces effectif est son ensemble de contraintes de *INTERFACE_TYPE* .
- Si n' `T` a pas de contraintes de *INTERFACE_TYPE* mais a des contraintes de *type_parameter* , son ensemble d'interfaces effectif est l'Union des jeux d'interfaces effectifs de ses contraintes de *type_parameter* .
- Si `T` a à la fois *INTERFACE_TYPE* contraintes et *type_parameter* contraintes, son ensemble d'interfaces effectif est l'Union de son ensemble de contraintes de

INTERFACE_TYPE et des ensembles d'interfaces effectifs de ses contraintes de *type_parameter* .

Un paramètre de type est ***connu comme un type référence*** s'il a la contrainte de type référence ou sa classe de base effective n'est pas `object` ou `System.ValueType` .

Les valeurs d'un type de paramètre de type restreint peuvent être utilisées pour accéder aux membres de l'instance implicite par les contraintes. Dans l'exemple

C#

```
interface IPrintable
{
    void Print();
}

class Printer<T> where T: IPrintable
{
    void PrintOne(T x) {
        x.Print();
    }
}
```

les méthodes de `IPrintable` peuvent être appelées directement sur `x` , car `T` est restreinte pour toujours implémenter `IPrintable` .

Corps de la classe

La *class_body* d'une classe définit les membres de cette classe.

antlr

```
class_body
    : '{' class_member_declaration* '}'
    ;
```

Types partiels

Une déclaration de type peut être fractionnée entre plusieurs ***déclarations de type partiel***. La déclaration de type est construite à partir de ses parties en suivant les règles de cette section, après quoi elle est traitée comme une déclaration unique pendant le reste du traitement au moment de la compilation et de l'exécution du programme.

Un *class_declaration*, *struct_declaration* ou *interface_declaration* représente une déclaration de type partiel s'il comprend un `partial` modificateur. `partial` n'est pas un mot clé et agit uniquement comme un modificateur s'il apparaît immédiatement avant l'un des mots clés `class` , `struct` ou `interface` dans une déclaration de type, ou avant le type `void` dans une déclaration de méthode. Dans d'autres contextes, il peut être utilisé comme un identificateur normal.

Chaque partie d'une déclaration de type partiel doit inclure un `partial` modificateur. Elle doit avoir le même nom et être déclarée dans le même espace de noms ou déclaration de type que les autres parties. Le `partial` modificateur indique que des parties supplémentaires de la déclaration de type peuvent exister ailleurs, mais que l'existence de ces parties supplémentaires n'est pas une exigence ; il est valide pour un type avec une déclaration unique pour inclure le `partial` modificateur.

Toutes les parties d'un type partiel doivent être compilées ensemble afin que les parties puissent être fusionnées au moment de la compilation dans une seule déclaration de type. En particulier, les types partiels n'autorisent pas l'extension des types déjà compilés.

Les types imbriqués peuvent être déclarés dans plusieurs parties à l'aide du `partial` modificateur. En règle générale, le type conteneur est déclaré à l'aide de `partial` également et chaque partie du type imbriqué est déclarée dans une autre partie du type conteneur.

Le `partial` modificateur n'est pas autorisé sur les déclarations de délégué ou d'énumération.

Attributs

Les attributs d'un type partiel sont déterminés en combinant, dans un ordre non spécifié, les attributs de chacune des parties. Si un attribut est placé sur plusieurs parties, il équivaut à spécifier l'attribut plusieurs fois sur le type. Par exemple, les deux parties :

```
C#
```

```
[Attr1, Attr2("hello")]
partial class A {}

[Attr3, Attr2("goodbye")]
partial class A {}
```

équivalent à une déclaration telle que :

C#

```
[Attr1, Attr2("hello"), Attr3, Attr2("goodbye")]
class A {}
```

Les attributs sur les paramètres de type sont combinés de manière similaire.

Modificateurs

Lorsqu'une déclaration de type partiel inclut une spécification d'accessibilité (les `public` `protected` `internal` modificateurs,, et `private`), elle doit accepter toutes les autres parties qui incluent une spécification d'accessibilité. Si aucune partie d'un type partiel n'intègre une spécification d'accessibilité, le type reçoit l'accessibilité par défaut appropriée (accessibilité déclarée).

Si une ou plusieurs déclarations partielles d'un type imbriqué incluent un `new` modificateur, aucun avertissement n'est signalé si le type imbriqué masque un membre hérité (masquage par héritage).

Si une ou plusieurs déclarations partielles d'une classe incluent un `abstract` modificateur, la classe est considérée comme abstraite (classes abstraites). Dans le cas contraire, la classe est considérée comme non abstraite.

Si une ou plusieurs déclarations partielles d'une classe incluent un `sealed` modificateur, la classe est considérée comme sealed (classes sealed). Dans le cas contraire, la classe est considérée comme non scellée.

Notez qu'une classe ne peut pas être à la fois abstract et sealed.

Quand le `unsafe` modificateur est utilisé sur une déclaration de type partiel, seul ce composant particulier est considéré comme un contexte non sécurisé (contextes non sécurisés).

Paramètres de type et contraintes

Si un type générique est déclaré dans plusieurs parties, chaque partie doit indiquer les paramètres de type. Chaque partie doit avoir le même nombre de paramètres de type et le même nom pour chaque paramètre de type, dans l'ordre.

Lorsqu'une déclaration de type générique partielle inclut des contraintes (`where` clauses), les contraintes doivent être conformes à toutes les autres parties qui incluent des contraintes. Plus précisément, chaque partie qui comprend des contraintes doit

avoir des contraintes pour le même jeu de paramètres de type, et pour chaque paramètre de type, les ensembles de contraintes primaires, secondaires et de constructeurs doivent être équivalents. Deux ensembles de contraintes sont équivalents s'ils contiennent les mêmes membres. Si aucune partie d'un type générique partiel ne spécifie des contraintes de paramètre de type, les paramètres de type sont considérés comme non restreints.

L'exemple

```C#
partial class Dictionary<K,V>
    where K: IComparable<K>
    where V: IKeyProvider<K>, IPersistable
{
    ...
}

partial class Dictionary<K,V>
    where V: IPersistable, IKeyProvider<K>
    where K: IComparable<K>
{
    ...
}

partial class Dictionary<K,V>
{
    ...
}
```

est correct, car les parties qui incluent des contraintes (les deux premières) spécifient effectivement le même jeu de contraintes primaires, secondaires et de constructeur pour le même jeu de paramètres de type, respectivement.

Classe de base

Lorsqu'une déclaration de classe partielle inclut une spécification de classe de base, elle doit accepter toutes les autres parties qui incluent une spécification de classe de base. Si aucune partie d'une classe partielle n'intègre une spécification de classe de base, la classe de base devient `System.Object` (classes de base).

Interfaces de base

L'ensemble d'interfaces de base pour un type déclaré en plusieurs parties est l'Union des interfaces de base spécifiées sur chaque composant. Une interface de base

particulière ne peut être nommée qu'une seule fois sur chaque partie, mais elle est autorisée pour plusieurs parties à nommer les mêmes interfaces de base. Il ne doit exister qu'une seule implémentation des membres d'une interface de base donnée.

Dans l'exemple

```
C#

partial class C: IA, IB {...}

partial class C: IC {...}

partial class C: IA, IB {...}
```

l'ensemble d'interfaces de base pour la classe `C` est `IA` , `IB` et `IC` .

En règle générale, chaque partie fournit une implémentation de l'interface déclarée sur cette partie ; Toutefois, cela n'est pas obligatoire. Un composant peut fournir l'implémentation d'une interface déclarée sur une autre partie :

```
C#

partial class X
{
    int IComparable.CompareTo(object o) {...}
}

partial class X: IComparable
{
    ...
}
```

Membres

À l'exception des méthodes partielles (méthodes partielles), le jeu de membres d'un type déclaré en plusieurs parties est simplement l'Union de l'ensemble des membres déclarés dans chaque partie. Les corps de toutes les parties de la déclaration de type partagent le même espace de déclaration (déclarations) et l'étendue de chaque membre (portées) s'étend aux corps de toutes les parties. Le domaine d'accessibilité de n'importe quel membre comprend toujours toutes les parties du type englobant ; un `private` membre déclaré dans une partie est librement accessible à partir d'une autre partie. Il s'agit d'une erreur au moment de la compilation pour déclarer le même membre dans plusieurs parties du type, sauf si ce membre est un type avec le `partial` modificateur.

```
C#
```

```
partial class A
{
    int x;                      // Error, cannot declare x more than once

    partial class Inner         // Ok, Inner is a partial type
    {
        int y;
    }
}

partial class A
{
    int x;                      // Error, cannot declare x more than once

    partial class Inner         // Ok, Inner is a partial type
    {
        int z;
    }
}
```

L'ordonnancement des membres dans un type est rarement significatif pour le code C#, mais peut être significatif lors de l'interfaçage avec d'autres langages et environnements. Dans ce cas, l'ordre des membres dans un type déclaré dans plusieurs parties n'est pas défini.

Méthodes partielles

Les méthodes partielles peuvent être définies dans une partie d'une déclaration de type et implémentées dans une autre. L'implémentation est facultative ; Si aucune partie n'implémente la méthode partielle, la déclaration de méthode partielle et tous les appels à celle-ci sont supprimés de la déclaration de type résultant de la combinaison des parties.

Les méthodes partielles ne peuvent pas définir de modificateurs d'accès, mais sont implicitement `private` . Leur type de retour doit être `void` , et leurs paramètres ne peuvent pas avoir le `out` modificateur. L'identificateur `partial` est reconnu comme un mot clé spécial dans une déclaration de méthode uniquement s'il apparaît juste avant le `void` type ; sinon, il peut être utilisé comme un identificateur normal. Une méthode partielle ne peut pas implémenter explicitement des méthodes d'interface.

Il existe deux types de déclarations de méthode partielles : si le corps de la déclaration de méthode est un point-virgule, la déclaration est dite « * définition de la **déclaration de méthode partielle** _ ». Si le corps est fourni en tant que _block *, la déclaration est considérée comme étant une ***déclaration de méthode partielle d'implémentation***. Dans les parties d'une déclaration de type, il ne peut y avoir qu'une seule déclaration de

méthode partielle avec une signature donnée, et il ne peut y avoir qu'une seule déclaration de méthode partielle implémentée avec une signature donnée. Si une déclaration de méthode partielle d'implémentation est fournie, une déclaration de méthode partielle de définition correspondante doit exister et les déclarations doivent correspondre comme indiqué dans l'exemple suivant :

- Les déclarations doivent avoir les mêmes modificateurs (mais pas nécessairement dans le même ordre), le nom de la méthode, le nombre de paramètres de type et le nombre de paramètres.
- Les paramètres correspondants dans les déclarations doivent avoir les mêmes modificateurs (mais pas nécessairement dans le même ordre) et les mêmes types (différences modulo dans les noms de paramètres de type).
- Les paramètres de type correspondants dans les déclarations doivent avoir les mêmes contraintes (différences modulo dans les noms de paramètre de type).

Une déclaration de méthode partielle d'implémentation peut apparaître dans la même partie que la déclaration de méthode partielle de définition correspondante.

Seule une méthode partielle de définition participe à la résolution de surcharge. Par conséquent, qu'une déclaration d'implémentation soit donnée ou non, les expressions d'appel peuvent être résolues en appels de la méthode partielle. Comme une méthode partielle retourne toujours `void` , ces expressions d'appel sont toujours des instructions d'expression. En outre, étant donné qu'une méthode partielle est implicitement `private` , ces instructions sont toujours exécutées dans l'une des parties de la déclaration de type dans laquelle la méthode partielle est déclarée.

Si aucune partie d'une déclaration de type partiel ne contient une déclaration d'implémentation pour une méthode partielle donnée, toute instruction d'expression qui l'appelle est simplement supprimée de la déclaration de type combiné. Ainsi, l'expression d'appel, y compris toutes les expressions constitutives, n'a aucun effet au moment de l'exécution. La méthode partielle elle-même est également supprimée et ne sera pas membre de la déclaration de type combiné.

Si une déclaration d'implémentation existe pour une méthode partielle donnée, les appels des méthodes partielles sont conservés. La méthode partielle donne lieu à une déclaration de méthode similaire à la déclaration de méthode partielle d'implémentation, à l'exception des éléments suivants :

- Le `partial` modificateur n'est pas inclus
- Les attributs de la déclaration de méthode résultante sont les attributs combinés de la définition et de la déclaration de méthode partielle d'implémentation dans un ordre non spécifié. Les doublons ne sont pas supprimés.

- Les attributs sur les paramètres de la déclaration de méthode résultante sont les attributs combinés des paramètres correspondants de la définition et de la déclaration de méthode partielle d'implémentation dans un ordre non spécifié. Les doublons ne sont pas supprimés.

Si une déclaration de définition, mais pas une déclaration d'implémentation, est fournie pour une méthode partielle M, les restrictions suivantes s'appliquent :

- Il s'agit d'une erreur au moment de la compilation pour créer un délégué à une méthode (expressions de création de délégué).
- Il s'agit d'une erreur de compilation qui se réfère à `M` l'intérieur d'une fonction anonyme convertie en un type d'arborescence d'expression (évaluation de conversions de fonction anonyme en types d'arborescence d'expression).
- Les expressions qui se trouvent dans le cadre d'un appel de `M` n'affectent pas l'état d'assignation définie (assignation définie), ce qui peut entraîner des erreurs de compilation.
- `M` ne peut pas être le point d'entrée d'une application (démarrage de l'application).

Les méthodes partielles sont utiles pour permettre à une partie d'une déclaration de type de personnaliser le comportement d'une autre partie, par exemple, celle qui est générée par un outil. Considérons la déclaration de classe partielle suivante :

C#

```
partial class Customer
{
    string name;

    public string Name {
        get { return name; }
        set {
            OnNameChanging(value);
            name = value;
            OnNameChanged();
        }

    }

    partial void OnNameChanging(string newName);

    partial void OnNameChanged();
}
```

Si cette classe est compilée sans autre partie, les déclarations de méthode partielle de définition et leurs appels seront supprimés, et la déclaration de classe combinée résultante sera équivalente à ce qui suit :

C#

```
class Customer
{
    string name;

    public string Name {
        get { return name; }
        set { name = value; }
    }
}
```

Supposons qu'une autre partie est fournie, toutefois, qui fournit des déclarations d'implémentation des méthodes partielles :

C#

```
partial class Customer
{
    partial void OnNameChanging(string newName)
    {
        Console.WriteLine("Changing " + name + " to " + newName);
    }

    partial void OnNameChanged()
    {
        Console.WriteLine("Changed to " + name);
    }
}
```

La déclaration de classe combinée résultante est alors équivalente à ce qui suit :

C#

```
class Customer
{
    string name;

    public string Name {
        get { return name; }
        set {
            OnNameChanging(value);
            name = value;
            OnNameChanged();
        }

    }

    void OnNameChanging(string newName)
    {
        Console.WriteLine("Changing " + name + " to " + newName);
```

```
    }

    void OnNameChanged()
    {
        Console.WriteLine("Changed to " + name);
    }
}
```

Liaison de nom

Bien que chaque partie d'un type extensible doive être déclarée dans le même espace de noms, les parties sont généralement écrites dans des déclarations d'espaces de noms différentes. Ainsi, différentes `using` directives (directives using) peuvent être présentes pour chaque partie. Lors de l'interprétation de noms simples (inférence de type) dans une seule partie, seules les `using` directives de la ou des déclarations d'espace de noms englobant cette partie sont prises en compte. Cela peut entraîner un même identificateur ayant des significations différentes dans différentes parties :

C#

```
namespace N
{
    using List = System.Collections.ArrayList;

    partial class A
    {
        List x;                 // x has type System.Collections.ArrayList
    }
}

namespace N
{
    using List = Widgets.LinkedList;

    partial class A
    {
        List y;                 // y has type Widgets.LinkedList
    }
}
```

Membres de classe

Les membres d'une classe se composent des membres introduits par son *class_member_declaration* s et des membres hérités de la classe de base directe.

antlr

```
class_member_declaration
    : constant_declaration
    | field_declaration
    | method_declaration
    | property_declaration
    | event_declaration
    | indexer_declaration
    | operator_declaration
    | constructor_declaration
    | destructor_declaration
    | static_constructor_declaration
    | type_declaration
    ;
```

Les membres d'un type de classe sont répartis dans les catégories suivantes :

- Les constantes, qui représentent des valeurs constantes associées à la classe (constantes).
- Les champs, qui sont les variables de la classe (champs).
- Les méthodes qui implémentent les calculs et les actions qui peuvent être effectuées par la classe (méthodes).
- Propriétés, qui définissent les caractéristiques nommées et les actions associées à la lecture et à l'écriture de ces caractéristiques (Propriétés).
- Les événements, qui définissent des notifications qui peuvent être générées par la classe (événements).
- Les indexeurs, qui autorisent l'indexation des instances de la classe de la même façon (syntaxiquement) en tant que tableaux (indexeurs).
- Les opérateurs, qui définissent les opérateurs d'expression qui peuvent être appliqués aux instances de la classe (opérateurs).
- Constructeurs d'instance, qui implémentent les actions requises pour initialiser les instances de la classe (constructeurs d'instance)
- Les destructeurs, qui implémentent les actions à effectuer avant que les instances de la classe soient définitivement ignorées (destructeurs).
- Constructeurs statiques, qui implémentent les actions requises pour initialiser la classe elle-même (constructeurs statiques).
- Types, qui représentent les types qui sont locaux pour la classe (types imbriqués).

Les membres qui peuvent contenir du code exécutable sont collectivement appelés *membres de fonction* du type de classe. Les fonctions membres d'un type de classe sont les méthodes, les propriétés, les événements, les indexeurs, les opérateurs, les constructeurs d'instance, les destructeurs et les constructeurs statiques de ce type de classe.

Un *class_declaration* crée un nouvel espace de déclaration (déclarations) et le *class_member_declaration* s contenu immédiatement dans l' *class_declaration* introduire de nouveaux membres dans cet espace de déclaration. Les règles suivantes s'appliquent à *class_member_declaration* s :

- Les constructeurs d'instance, les destructeurs et les constructeurs statiques doivent avoir le même nom que la classe immédiatement englobante. Tous les autres membres doivent avoir des noms qui diffèrent du nom de la classe englobante immédiatement.
- Le nom d'une constante, d'un champ, d'une propriété, d'un événement ou d'un type doit différer des noms de tous les autres membres déclarés dans la même classe.
- Le nom d'une méthode doit être différent des noms de toutes les autres méthodes non déclarées dans la même classe. En outre, la signature (signatures et surcharge) d'une méthode doit être différente des signatures de toutes les autres méthodes déclarées dans la même classe, et deux méthodes déclarées dans la même classe ne peuvent pas avoir de signatures qui diffèrent uniquement par `ref` et `out` .
- La signature d'un constructeur d'instance doit être différente des signatures de tous les autres constructeurs d'instance déclarés dans la même classe, et deux constructeurs déclarés dans la même classe ne peuvent pas avoir de signatures qui diffèrent uniquement par `ref` et `out` .
- La signature d'un indexeur doit être différente des signatures de tous les autres indexeurs déclarés dans la même classe.
- La signature d'un opérateur doit être différente des signatures de tous les autres opérateurs déclarés dans la même classe.

Les membres hérités d'un type de classe (héritage) ne font pas partie de l'espace de déclaration d'une classe. Par conséquent, une classe dérivée est autorisée à déclarer un membre avec le même nom ou la même signature qu'un membre hérité (qui, en effet, masque le membre hérité).

Type d'instance

Chaque déclaration de classe a un type dépendant associé (types liés et indépendants), le ***type d'instance***. Pour une déclaration de classe générique, le type d'instance est formé en créant un type construit (types construits) à partir de la déclaration de type, avec chacun des arguments de type fournis étant le paramètre de type correspondant. Étant donné que le type d'instance utilise les paramètres de type, il ne peut être utilisé que lorsque les paramètres de type sont dans la portée ; autrement dit, à l'intérieur de la déclaration de classe. Le type d'instance est le type de `this` pour le code écrit à l'intérieur de la déclaration de classe. Pour les classes non génériques, le type d'instance

est simplement la classe déclarée. L'exemple suivant illustre plusieurs déclarations de classe avec leurs types d'instance :

```
C#

class A<T>                              // instance type: A<T>
{
    class B {}                          // instance type: A<T>.B
    class C<U> {}                       // instance type: A<T>.C<U>
}

class D {}                              // instance type: D
```

Membres des types construits

Les membres non hérités d'un type construit sont obtenus en substituant, pour chaque *type_parameter* dans la déclaration de membre, les *type_argument* correspondants du type construit. Le processus de substitution est basé sur la signification sémantique des déclarations de type et n'est pas simplement une substitution textuelle.

Par exemple, à partir de la déclaration de classe générique

```
C#

class Gen<T,U>
{
    public T[,] a;
    public void G(int i, T t, Gen<U,T> gt) {...}
    public U Prop { get {...} set {...} }
    public int H(double d) {...}
}
```

le type construit `Gen<int[],IComparable<string>>` contient les membres suivants :

```
C#

public int[,][] a;
public void G(int i, int[] t, Gen<IComparable<string>,int[]> gt) {...}
public IComparable<string> Prop { get {...} set {...} }
public int H(double d) {...}
```

Le type du membre `a` dans la déclaration de classe générique `Gen` est « tableau à deux dimensions de `T` ». par conséquent, le type du membre `a` dans le type construit ci-dessus est « tableau à deux dimensions d'un tableau unidimensionnel de `int` » ou `int[,][]` .

Dans les membres de fonction d'instance, le type de `this` est le type d'instance (type d'instance) de la déclaration conteneur.

Tous les membres d'une classe générique peuvent utiliser des paramètres de type de n'importe quelle classe englobante, directement ou dans le cadre d'un type construit. Lorsqu'un type construit fermé particulier (types ouverts et fermés) est utilisé au moment de l'exécution, chaque utilisation d'un paramètre de type est remplacée par l'argument de type réel fourni au type construit. Par exemple :

C#

```
class C<V>
{
    public V f1;
    public C<V> f2 = null;

    public C(V x) {
        this.f1 = x;
        this.f2 = this;
    }
}

class Application
{
    static void Main() {
        C<int> x1 = new C<int>(1);
        Console.WriteLine(x1.f1);        // Prints 1

        C<double> x2 = new C<double>(3.1415);
        Console.WriteLine(x2.f1);        // Prints 3.1415
    }
}
```

Héritage

Une classe ***hérite*** des membres de son type de classe de base directe. L'héritage signifie qu'une classe contient implicitement tous les membres de son type de classe de base directe, à l'exception des constructeurs d'instance, des destructeurs et des constructeurs statiques de la classe de base. Voici quelques aspects importants de l'héritage :

- L'héritage est transitif. Si `C` est dérivé de, `B` et `B` est dérivé de `A` , `C` hérite des membres déclarés dans, ainsi `B` que des membres déclarés dans `A` .
- Une classe dérivée étend sa classe de base directe. Une classe dérivée peut ajouter des membres hérités, mais ne peut pas supprimer la définition d'un membre hérité.

- Les constructeurs d'instance, les destructeurs et les constructeurs statiques ne sont pas hérités, mais tous les autres membres sont, indépendamment de leur accessibilité déclarée (accès aux membres). Toutefois, selon l'accessibilité déclarée, les membres hérités peuvent ne pas être accessibles dans une classe dérivée.
- Une classe dérivée peut ***Masquer*** les membres hérités (masquage par héritage) en déclarant de nouveaux membres portant le même nom ou la même signature. Notez cependant que le masquage d'un membre hérité ne supprime pas ce membre ; il rend simplement ce membre inaccessible directement via la classe dérivée.
- Une instance d'une classe contient un ensemble de tous les champs d'instance déclarés dans la classe et ses classes de base, et une conversion implicite (conversions de référence implicites) existe d'un type de classe dérivée vers l'un de ses types de classe de base. Ainsi, une référence à une instance d'une classe dérivée peut être traitée comme une référence à une instance de l'une de ses classes de base.
- Une classe peut déclarer des méthodes, des propriétés et des indexeurs virtuels, et les classes dérivées peuvent substituer l'implémentation de ces membres de fonction. Cela permet aux classes d'exposer un comportement polymorphe dans lequel les actions effectuées par un appel de membre de fonction varient en fonction du type d'exécution de l'instance par le biais duquel cette fonction membre est appelée.

Le membre hérité d'un type de classe construit sont les membres du type de classe de base immédiat (classes de base), qui est trouvé en substituant les arguments de type du type construit pour chaque occurrence des paramètres de type correspondants dans la spécification *class_base* . Ces membres, à leur tour, sont transformés en substituant, pour chaque *type_parameter* dans la déclaration de membre, les *type_argument* correspondants de la spécification de *class_base* .

C#

```
class B<U>
{
    public U F(long index) {...}
}

class D<T>: B<T[]>
{
    public T G(string s) {...}
}
```

Dans l'exemple ci-dessus, le type construit `D<int>` a un membre non hérité `public int G(string s)` obtenu en substituant l'argument `int` de type pour le paramètre de type `T`

. `D<int>` a également un membre hérité de la déclaration de classe `B` . Ce membre hérité est déterminé par la première détermination du type `B<int[]>` de classe de base de `D<int>` en remplaçant `int` `T` dans la spécification de la classe de base `B<T[]>` . Ensuite, en tant qu'argument de type pour `B` , `int[]` est substitué à `U` dans `public U F(long index)` , ce qui donne le membre hérité `public int[] F(long index)` .

Le nouveau modificateur

Un *class_member_declaration* est autorisé à déclarer un membre avec le même nom ou la même signature qu'un membre hérité. Dans ce cas, le membre de la classe dérivée est dit de ***Masquer*** le membre de la classe de base. Le masquage d'un membre hérité n'est pas considéré comme une erreur, mais il entraîne l'émission d'un avertissement par le compilateur. Pour supprimer l'avertissement, la déclaration du membre de classe dérivée peut inclure un `new` modificateur pour indiquer que le membre dérivé est destiné à masquer le membre de base. Cette rubrique est traitée plus en détail dans Masquer par héritage.

Si un `new` modificateur est inclus dans une déclaration qui ne masque pas un membre hérité, un avertissement est émis à cet effet. Cet avertissement est supprimé en supprimant le `new` modificateur.

Modificateurs d'accès

Un *class_member_declaration* peut avoir l'un des cinq types d'accessibilité déclarés possibles (accessibilité déclarée) : `public` , `protected internal` ,, `protected` `internal` ou `private` . À l'exception de la `protected internal` combinaison, il s'agit d'une erreur de compilation pour spécifier plusieurs modificateurs d'accès. Lorsqu'un *class_member_declaration* n'inclut pas de modificateur d'accès, `private` est utilisé par défaut.

Types constitutifs

Les types qui sont utilisés dans la déclaration d'un membre sont appelés types constitutifs de ce membre. Les types constitutifs possibles sont le type d'une constante, d'un champ, d'une propriété, d'un événement ou d'un indexeur, le type de retour d'une méthode ou d'un opérateur, ainsi que les types de paramètres d'une méthode, d'un indexeur, d'un opérateur ou d'un constructeur d'instance. Les types constitutifs d'un membre doivent être au moins aussi accessibles que ce membre lui-même (contraintes d'accessibilité).

Membres statiques et d'instance

Les membres d'une classe sont ***membres statiques*** _ *ou* _ * *membres de l'instance* *. En général, il est utile de considérer que les membres statiques appartiennent à des types de classe et des membres d'instance comme appartenant à des objets (instances de types de classe).

Lorsqu'une déclaration de champ, de méthode, de propriété, d'événement, d'opérateur ou de constructeur comprend un `static` modificateur, elle déclare un membre statique. En outre, une déclaration de constante ou de type déclare implicitement un membre statique. Les membres statiques présentent les caractéristiques suivantes :

- Lorsqu'un membre statique `M` est référencé dans un *member_access* (accès aux membres) du formulaire `E.M` , `E` doit désigner un type contenant `M` . Il s'agit d'une erreur au moment de la compilation pour `E` qu'une instance soit dénotée.
- Un champ statique identifie exactement un emplacement de stockage à partager par toutes les instances d'un type de classe fermé donné. Quel que soit le nombre d'instances d'un type de classe fermé donné, il n'existe qu'une seule copie d'un champ statique.
- Un membre de fonction statique (méthode, propriété, événement, opérateur ou constructeur) ne fonctionne pas sur une instance spécifique, et il s'agit d'une erreur de compilation pour faire référence à `this` dans ce type de membre de fonction.

Quand un champ, une méthode, une propriété, un événement, un indexeur, un constructeur ou une déclaration de destructeur n'inclut pas de `static` modificateur, il déclare un membre d'instance. (Un membre d'instance est parfois appelé membre non statique.) Les membres d'instance présentent les caractéristiques suivantes :

- Lorsqu'un membre d'instance `M` est référencé dans un *member_access* (accès aux membres) du formulaire `E.M` , `E` doit désigner une instance d'un type contenant `M` . Il s'agit d'une erreur de liaison au moment de la liaison pour `E` indiquer un type.
- Chaque instance d'une classe contient un ensemble distinct de tous les champs d'instance de la classe.
- Un membre de fonction d'instance (méthode, propriété, indexeur, constructeur d'instance ou destructeur) opère sur une instance donnée de la classe, et cette instance est accessible en tant que `this` (cet accès).

L'exemple suivant illustre les règles d'accès aux membres statiques et d'instance :

```
C#

class Test
{
```

```
    int x;
    static int y;

    void F() {
        x = 1;            // Ok, same as this.x = 1
        y = 1;            // Ok, same as Test.y = 1
    }

    static void G() {
        x = 1;            // Error, cannot access this.x
        y = 1;            // Ok, same as Test.y = 1
    }

    static void Main() {
        Test t = new Test();
        t.x = 1;          // Ok
        t.y = 1;          // Error, cannot access static member through
instance
        Test.x = 1;       // Error, cannot access instance member through
type
        Test.y = 1;       // Ok
    }
}
```

La `F` méthode montre que dans une fonction membre d'instance, un *simple_name* (noms simples) peut être utilisé pour accéder aux membres d'instance et aux membres statiques. La `G` méthode montre que dans une fonction membre statique, il s'agit d'une erreur au moment de la compilation pour accéder à un membre d'instance par le biais d'un *simple_name*. La `Main` méthode indique que dans une *member_access* (accès aux membres), les membres d'instance doivent être accessibles via des instances et les membres statiques doivent être accessibles via des types.

Types imbriqués

Un type déclaré dans une déclaration de classe ou de struct est appelé un ***type imbriqué** _. Un type déclaré dans une unité de compilation ou un espace de noms est appelé un _ *type non imbriqué** *.

Dans l'exemple

C#

```
using System;

class A
{
    class B
    {
```

```
        static void F() {
            Console.WriteLine("A.B.F");
        }
    }
}
```

`B`la classe est un type imbriqué, car elle est déclarée dans `A` la classe, et `A` la classe est un type non imbriqué, car elle est déclarée dans une unité de compilation.

Nom complet

Le nom qualifié complet (noms qualifiés complets) pour un type imbriqué est `S.N` où `S` est le nom qualifié complet du type dans lequel le type `N` est déclaré.

Accessibilité déclarée

Les types non imbriqués peuvent avoir `public` ou `internal` déclarés l'accessibilité et avoir `internal` déclaré l'accessibilité par défaut. Les types imbriqués peuvent également avoir ces formes d'accessibilité déclarée, plus une ou plusieurs formes supplémentaires d'accessibilité déclarée, selon que le type conteneur est une classe ou un struct :

- Un type imbriqué qui est déclaré dans une classe peut avoir l'une des cinq formes d'accessibilité déclarée (`public` , `protected internal` ,, `protected` `internal` ou `private`) et, comme les autres membres de classe, l' `private` accessibilité déclarée est par défaut.
- Un type imbriqué qui est déclaré dans un struct peut avoir l'une des trois formes d'accessibilité déclarée (`public` , `internal` ou `private`) et, comme les autres membres de struct, a comme valeur par défaut l' `private` accessibilité déclarée.

L'exemple

C#

```
public class List
{
    // Private data structure
    private class Node
    {
        public object Data;
        public Node Next;

        public Node(object data, Node next) {
            this.Data = data;
            this.Next = next;
        }
```

```
    }

    private Node first = null;
    private Node last = null;

    // Public interface
    public void AddToFront(object o) {...}
    public void AddToBack(object o) {...}
    public object RemoveFromFront() {...}
    public object RemoveFromBack() {...}
    public int Count { get {...} }
}
```

déclare une classe imbriquée privée `Node` .

Masquage

Un type imbriqué peut masquer (nom masquant) un membre de base. Le `new` modificateur est autorisé sur les déclarations de type imbriqué afin que le masquage puisse être exprimé explicitement. L'exemple

C#

```
using System;

class Base
{
    public static void M() {
        Console.WriteLine("Base.M");
    }
}

class Derived: Base
{
    new public class M
    {
        public static void F() {
            Console.WriteLine("Derived.M.F");
        }
    }
}

class Test
{
    static void Main() {
        Derived.M.F();
    }
}
```

affiche une classe imbriquée `M` qui masque la méthode `M` définie dans `Base` .

Cet accès

Un type imbriqué et son type conteneur n'ont pas de relation spéciale en ce qui concerne *This_Access* (cet accès). En particulier, `this` dans un type imbriqué ne peut pas être utilisé pour faire référence aux membres d'instance du type conteneur. Dans les cas où un type imbriqué doit accéder aux membres de l'instance de son type conteneur, l'accès peut être fourni en fournissant le `this` pour l'instance du type conteneur comme argument de constructeur pour le type imbriqué. Par exemple :

C#

```
using System;

class C
{
    int i = 123;

    public void F() {
        Nested n = new Nested(this);
        n.G();
    }

    public class Nested
    {
        C this_c;

        public Nested(C c) {
            this_c = c;
        }

        public void G() {
            Console.WriteLine(this_c.i);
        }
    }
}

class Test
{
    static void Main() {
        C c = new C();
        c.F();
    }
}
```

illustre cette technique. Une instance de `C` crée une instance de `Nested` et passe son propre `this` au `Nested` constructeur de pour fournir l'accès ultérieur aux `C` membres d'instance de.

Accès aux membres privés et protégés du type conteneur

Un type imbriqué a accès à tous les membres qui sont accessibles à son type conteneur, y compris les membres du type conteneur qui ont `private` et l' `protected` accessibilité déclarée. L'exemple

C#

```
using System;

class C
{
    private static void F() {
        Console.WriteLine("C.F");
    }

    public class Nested
    {
        public static void G() {
            F();
        }
    }
}

class Test
{
    static void Main() {
        C.Nested.G();
    }
}
```

affiche une classe `C` qui contient une classe imbriquée `Nested` . Dans `Nested` , la méthode `G` appelle la méthode statique `F` définie dans `C` et `F` a une accessibilité déclarée privée.

Un type imbriqué peut également accéder à des membres protégés définis dans un type de base de son type conteneur. Dans l'exemple

C#

```
using System;

class Base
{
    protected void F() {
        Console.WriteLine("Base.F");
    }
}

class Derived: Base
```

```
{
    public class Nested
    {
        public void G() {
            Derived d = new Derived();
            d.F();        // ok
        }
    }
}

class Test
{
    static void Main() {
        Derived.Nested n = new Derived.Nested();
        n.G();
    }
}
```

la classe imbriquée `Derived.Nested` accède à la méthode protégée `F` définie dans la `Derived` classe de base de, `Base` , en appelant par le biais d'une instance de `Derived` .

Types imbriqués dans les classes génériques

Une déclaration de classe générique peut contenir des déclarations de type imbriqué. Les paramètres de type de la classe englobante peuvent être utilisés dans les types imbriqués. Une déclaration de type imbriquée peut contenir des paramètres de type supplémentaires qui s'appliquent uniquement au type imbriqué.

Chaque déclaration de type contenue dans une déclaration de classe générique est implicitement une déclaration de type générique. Lors de l'écriture d'une référence à un type imbriqué dans un type générique, le type construit contenant, y compris ses arguments de type, doit être nommé. Toutefois, à partir de la classe externe, le type imbriqué peut être utilisé sans qualification ; le type d'instance de la classe externe peut être utilisé implicitement lors de la construction du type imbriqué. L'exemple suivant montre trois façons correctes de faire référence à un type construit créé à partir de `Inner` ; les deux premiers sont équivalents :

C#

```
class Outer<T>
{
    class Inner<U>
    {
        public static void F(T t, U u) {...}
    }

    static void F(T t) {
```

```
        Outer<T>.Inner<string>.F(t, "abc");      // These two statements
have
        Inner<string>.F(t, "abc");               // the same effect

        Outer<int>.Inner<string>.F(3, "abc");    // This type is different

        Outer.Inner<string>.F(t, "abc");         // Error, Outer needs type
arg
    }
}
```

Bien qu'il s'agisse d'un style de programmation incorrect, un paramètre de type dans un type imbriqué peut masquer un membre ou un paramètre de type déclaré dans le type externe :

C#

```
class Outer<T>
{
    class Inner<T>        // Valid, hides Outer's T
    {
        public T t;       // Refers to Inner's T
    }
}
```

Noms de membres réservés

Pour faciliter l'implémentation sous-jacente du runtime C#, pour chaque déclaration de membre source qui est une propriété, un événement ou un indexeur, l'implémentation doit réserver deux signatures de méthode selon le type de la déclaration de membre, son nom et son type. Il s'agit d'une erreur de compilation pour qu'un programme déclare un membre dont la signature correspond à l'une de ces signatures réservées, même si l'implémentation au moment de l'exécution sous-jacente n'utilise pas ces réservations.

Les noms réservés n'introduisent pas de déclarations, donc ils ne participent pas à la recherche de membres. Toutefois, les signatures de méthode réservée associées à une déclaration participent à l'héritage (héritage) et peuvent être masquées avec le `new` modificateur (le nouveau modificateur).

La réservation de ces noms remplit trois fonctions :

- Pour permettre à l'implémentation sous-jacente d'utiliser un identificateur ordinaire comme nom de méthode pour obtenir ou définir l'accès à la fonctionnalité du langage C#.

- Pour permettre à d'autres langages d'interagir à l'aide d'un identificateur ordinaire comme nom de méthode pour obtenir ou définir l'accès à la fonctionnalité du langage C#.
- Pour vous assurer que la source acceptée par un compilateur conforme est acceptée par un autre, en rendant les caractéristiques des noms de membres réservés cohérentes dans toutes les implémentations C#.

La déclaration d'un destructeur (destructeurs) entraîne également la réservation d'une signature (noms de membres réservés aux destructeurs).

Noms de membres réservés pour les propriétés

Pour une propriété `P` (Propriétés) de type `T` , les signatures suivantes sont réservées :

C#

```
T get_P();
void set_P(T value);
```

Les deux signatures sont réservées, même si la propriété est en lecture seule ou en écriture seule.

Dans l'exemple

C#

```
using System;

class A
{
    public int P {
        get { return 123; }
    }
}

class B: A
{
    new public int get_P() {
        return 456;
    }

    new public void set_P(int value) {
    }
}

class Test
{
    static void Main() {
```

```
        B b = new B();
        A a = b;
        Console.WriteLine(a.P);
        Console.WriteLine(b.P);
        Console.WriteLine(b.get_P());
    }
}
```

une classe `A` définit une propriété en lecture seule `P` , en réservant des signatures pour les `get_P` `set_P` méthodes et. Une classe `B` dérive de `A` et masque ces deux signatures réservées. L'exemple produit la sortie :

Console

```
123
123
456
```

Noms de membres réservés pour les événements

Pour un événement `E` (événements) de type délégué `T` , les signatures suivantes sont réservées :

C#

```
void add_E(T handler);
void remove_E(T handler);
```

Noms de membres réservés pour les indexeurs

Pour un indexeur (indexeurs) de type `T` avec Parameter-List `L` , les signatures suivantes sont réservées :

C#

```
T get_Item(L);
void set_Item(L, T value);
```

Les deux signatures sont réservées, même si l'indexeur est en lecture seule ou en écriture seule.

En outre, le nom du membre `Item` est réservé.

Noms de membres réservés pour les destructeurs

Pour une classe contenant un destructeur (destructeurs), la signature suivante est réservée :

```C#
void Finalize();
```

Constantes

Une **constante** _ _ est un membre de classe qui représente une valeur de constante : valeur qui peut être calculée au moment de la compilation. Une _constant_declaration * introduit une ou plusieurs constantes d'un type donné.

```antlr
constant_declaration
    : attributes? constant_modifier* 'const' type constant_declarators ';'
    ;

constant_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    ;

constant_declarators
    : constant_declarator (',' constant_declarator)*
    ;

constant_declarator
    : identifier '=' constant_expression
    ;
```

Une *constant_declaration* peut inclure un ensemble d' *attributs* (attributs), un `new` modificateur (le nouveau modificateur) et une combinaison valide des quatre modificateurs d'accès (modificateurs d'accès). Les attributs et les modificateurs s'appliquent à tous les membres déclarés par l' *constant_declaration*. Bien que les constantes soient considérées comme des membres statiques, une *constant_declaration* ne requiert pas et n'autorise pas de `static` modificateur. Il s'agit d'une erreur pour que le même modificateur apparaisse plusieurs fois dans une déclaration de constante.

Le *type* d'un *constant_declaration* spécifie le type des membres introduits par la déclaration. Le type est suivi d'une liste de *constant_declarator* s, chacun d'entre eux introduisant un nouveau membre. Un *constant_declarator* se compose d'un *identificateur* qui nomme le membre, suivi d'un `=` jeton «», suivi d'un *constant_expression* (expressions constantes) qui donne la valeur du membre.

Le *type* spécifié dans une déclaration de constante doit être `sbyte` , `byte` , `short` , `ushort` , `int` , `uint` , `long` , `ulong` , `char` , `float` , `double` , `decimal` , `bool` , `string` , un *enum_type* ou un *reference_type*. Chaque *constant_expression* doit produire une valeur du type cible ou d'un type qui peut être converti vers le type cible par une conversion implicite (conversions implicites).

Le *type* d'une constante doit être au moins aussi accessible que la constante elle-même (contraintes d'accessibilité).

La valeur d'une constante est obtenue dans une expression à l'aide d'un *simple_name* (noms simples) ou d'un *member_access* (accès aux membres).

Une constante peut elle-même participer à une *constant_expression*. Par conséquent, une constante peut être utilisée dans n'importe quelle construction nécessitant une *constant_expression*. Les exemples de ces constructions incluent les `case` étiquettes, les `goto case` instructions, les `enum` déclarations de membre, les attributs et d'autres déclarations de constante.

Comme décrit dans expressions constantes, une *constant_expression* est une expression qui peut être complètement évaluée au moment de la compilation. Étant donné que la seule façon de créer une valeur non null d'un *reference_type* autre que `string` consiste à appliquer l' `new` opérateur, et puisque l' `new` opérateur n'est pas autorisé dans une *constant_expression*, la seule valeur possible pour les constantes de *reference_type* s autres que `string` est `null` .

Lorsqu'un nom symbolique pour une valeur constante est souhaité, mais lorsque le type de cette valeur n'est pas autorisé dans une déclaration de constante, ou lorsque la valeur ne peut pas être calculée au moment de la compilation par un *constant_expression*, un `readonly` champ (champs ReadOnly) peut être utilisé à la place.

Une déclaration de constante qui déclare plusieurs constantes équivaut à plusieurs déclarations de constantes uniques avec les mêmes attributs, modificateurs et type. Par exemple

```
C#
```

```
class A
{
    public const double X = 1.0, Y = 2.0, Z = 3.0;
}
```

équivaut à :

C#

```
class A
{
    public const double X = 1.0;
    public const double Y = 2.0;
    public const double Z = 3.0;
}
```

Les constantes sont autorisées à dépendre d'autres constantes au sein du même programme tant que les dépendances ne sont pas de nature circulaire. Le compilateur s'arrange automatiquement pour évaluer les déclarations de constantes dans l'ordre approprié. Dans l'exemple

C#

```
class A
{
    public const int X = B.Z + 1;
    public const int Y = 10;
}

class B
{
    public const int Z = A.Y + 1;
}
```

le compilateur évalue `A.Y` , puis évalue `B.Z` et évalue Enfin, en `A.X` produisant les valeurs `10` , `11` et `12` . Les déclarations de constante peuvent dépendre de constantes d'autres programmes, mais ces dépendances ne sont possibles que dans une seule direction. En vous référant à l'exemple ci-dessus, si `A` et `B` ont été déclarés dans des programmes distincts, il serait possible que dépende de `A.X` `B.Z` , mais `B.Z` ne dépendait pas simultanément de `A.Y` .

Champs

Un **champ*** _ est un membre qui représente une variable associée à un objet ou une classe. Un _field_declaration * introduit un ou plusieurs champs d'un type donné.

antlr

```
field_declaration
    : attributes? field_modifier* type variable_declarators ';'
    ;

field_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'static'
    | 'readonly'
    | 'volatile'
    | field_modifier_unsafe
    ;

variable_declarators
    : variable_declarator (',' variable_declarator)*
    ;

variable_declarator
    : identifier ('=' variable_initializer)?
    ;

variable_initializer
    : expression
    | array_initializer
    ;
```

Une *field_declaration* peut inclure un ensemble d' *attributs* (attributs), un `new` modificateur (le nouveau modificateur), une combinaison valide des quatre modificateurs d'accès (modificateurs d'accès) et un `static` modificateur (champs statiques et d'instance). En outre, un *field_declaration* peut inclure un `readonly` modificateur (champs ReadOnly) ou un `volatile` modificateur (champs volatiles), mais pas les deux. Les attributs et les modificateurs s'appliquent à tous les membres déclarés par l' *field_declaration*. Il s'agit d'une erreur pour que le même modificateur apparaisse plusieurs fois dans une déclaration de champ.

Le *type* d'un *field_declaration* spécifie le type des membres introduits par la déclaration. Le type est suivi d'une liste de *variable_declarator* s, chacun d'entre eux introduisant un nouveau membre. Un *variable_declarator* se compose d'un *identificateur* qui nomme ce membre, éventuellement suivi d'un jeton « `=` » et d'un *variable_initializer* (initialiseurs de variable) qui donne la valeur initiale de ce membre.

Le *type* d'un champ doit être au moins aussi accessible que le champ lui-même (contraintes d'accessibilité).

La valeur d'un champ est obtenue dans une expression à l'aide d'un *simple_name* (noms simples) ou d'un *member_access* (accès aux membres). La valeur d'un champ qui n'est pas en lecture seule est modifiée à l'aide d'une *assignation* (opérateurs d'affectation). La valeur d'un champ non readonly peut être obtenue et modifiée à l'aide d'opérateurs suffixés d'incrémentation et de décrémentation (opérateurs suffixés d'incrémentation et de décrémentation) et d'opérateurs de préfixe et d'incrémentation (incréments de préfixe et opérateur de décrémentation).

Une déclaration de champ qui déclare plusieurs champs équivaut à plusieurs déclarations de champs uniques avec les mêmes attributs, modificateurs et type. Par exemple

C#

```
class A
{
    public static int X = 1, Y, Z = 100;
}
```

équivaut à :

C#

```
class A
{
    public static int X = 1;
    public static int Y;
    public static int Z = 100;
}
```

Champs d'instance et statiques

Lorsqu'une déclaration de champ comprend un `static` modificateur, les champs introduits par la déclaration sont *__champs statiques__ _. Si aucun `static` modificateur n'est présent, les champs introduits par la déclaration sont des *champs d'instance*. Les champs statiques et les champs d'instance sont deux des différents types de variables (variables) pris en charge par C#, et parfois appelés *variables statiques* et _ *variables d'instance** *, respectivement.

Un champ statique ne fait pas partie d'une instance spécifique ; au lieu de cela, il est partagé entre toutes les instances d'un type fermé (types ouverts et fermés). Quel que

soit le nombre d'instances d'un type de classe fermé, il n'existe qu'une seule copie d'un champ statique pour le domaine d'application associé.

Par exemple :

C#

```
class C<V>
{
    static int count = 0;

    public C() {
        count++;
    }

    public static int Count {
        get { return count; }
    }
}

class Application
{
    static void Main() {
        C<int> x1 = new C<int>();
        Console.WriteLine(C<int>.Count);        // Prints 1

        C<double> x2 = new C<double>();
        Console.WriteLine(C<int>.Count);        // Prints 1

        C<int> x3 = new C<int>();
        Console.WriteLine(C<int>.Count);        // Prints 2
    }
}
```

Un champ d'instance appartient à une instance. Plus précisément, chaque instance d'une classe contient un ensemble distinct de tous les champs d'instance de cette classe.

Quand un champ est référencé dans un *member_access* (accès aux membres) du formulaire `E.M`, si `M` est un champ statique, `E` doit désigner un type contenant `M`, et si `M` est un champ d'instance, E doit désigner une instance d'un type contenant `M`.

Les différences entre les membres statiques et les membres d'instance sont abordées plus en détail dans les membres statiques et d'instance.

Champs en lecture seule

Lorsqu'un *field_declaration* comprend un `readonly` modificateur, les champs introduits par la déclaration sont des ***champs en lecture seule***. Les assignations directes aux

champs ReadOnly ne peuvent se produire que dans le cadre de cette déclaration ou dans un constructeur d'instance ou un constructeur statique dans la même classe. (Un champ readonly peut être affecté à plusieurs fois dans ces contextes.) Plus précisément, les assignations directes à un `readonly` champ sont autorisées uniquement dans les contextes suivants :

- Dans le *variable_declarator* qui introduit le champ (en incluant un *variable_initializer* dans la déclaration).
- Pour un champ d'instance, dans les constructeurs d'instance de la classe qui contient la déclaration de champ ; pour un champ statique, dans le constructeur statique de la classe qui contient la déclaration de champ. Il s'agit également des seuls contextes dans lesquels il est possible de passer un `readonly` champ en tant que `out` `ref` paramètre ou.

Toute tentative d'assignation à un `readonly` champ ou de transmission en tant que `out` `ref` paramètre ou dans un autre contexte est une erreur au moment de la compilation.

Utilisation de champs ReadOnly statiques pour les constantes

Un `static readonly` champ est utile lorsqu'un nom symbolique pour une valeur constante est souhaité, mais lorsque le type de la valeur n'est pas autorisé dans une `const` déclaration, ou lorsque la valeur ne peut pas être calculée au moment de la compilation. Dans l'exemple

C#

```
public class Color
{
    public static readonly Color Black = new Color(0, 0, 0);
    public static readonly Color White = new Color(255, 255, 255);
    public static readonly Color Red = new Color(255, 0, 0);
    public static readonly Color Green = new Color(0, 255, 0);
    public static readonly Color Blue = new Color(0, 0, 255);

    private byte red, green, blue;

    public Color(byte r, byte g, byte b) {
        red = r;
        green = g;
        blue = b;
    }
}
```

les `Black` `White` membres,, `Red` , `Green` et `Blue` ne peuvent pas être déclarés comme `const` membres, car leurs valeurs ne peuvent pas être calculées au moment de la

compilation. Toutefois, leur déclaration à `static readonly` la place a un effet très similaire.

Contrôle de version des constantes et des champs ReadOnly statiques

Les constantes et les champs ReadOnly ont une sémantique de version binaire différente. Lorsqu'une expression fait référence à une constante, la valeur de la constante est obtenue au moment de la compilation, mais lorsqu'une expression fait référence à un champ ReadOnly, la valeur du champ n'est obtenue qu'au moment de l'exécution. Prenons l'exemple d'une application qui se compose de deux programmes distincts :

C#

```
using System;

namespace Program1
{
    public class Utils
    {
        public static readonly int X = 1;
    }
}

namespace Program2
{
    class Test
    {
        static void Main() {
            Console.WriteLine(Program1.Utils.X);
        }
    }
}
```

Les `Program1` `Program2` espaces de noms et désignent deux programmes qui sont compilés séparément. Étant donné que `Program1.Utils.X` est déclaré en tant que champ readonly statique, la valeur générée par l' `Console.WriteLine` instruction n'est pas connue au moment de la compilation, mais est plutôt obtenue au moment de l'exécution. Ainsi, si la valeur de `X` est modifiée et `Program1` est recompilée, l' `Console.WriteLine` instruction génère la nouvelle valeur même si `Program2` n'est pas recompilé. Toutefois, avait `X` été une constante, la valeur de `X` aurait été obtenue au moment de la `Program2` compilation et ne serait pas affectée par les modifications apportées à `Program1` jusqu'à la `Program2` recompilation.

Champs volatiles

Lorsqu'un *field_declaration* comprend un `volatile` modificateur, les champs introduits par cette déclaration sont des ***champs volatiles***.

Pour les champs non volatiles, les techniques d'optimisation qui réorganisent les instructions peuvent entraîner des résultats inattendus et imprévisibles dans les programmes multithread qui accèdent aux champs sans synchronisation, comme celui fourni par le *lock_statement* (instruction lock). Ces optimisations peuvent être effectuées par le compilateur, par le système d'exécution ou par le matériel. Pour les champs volatils, ces optimisations de réorganisation sont restreintes :

- La lecture d'un champ volatile est appelée une ***lecture volatile***. Une lecture volatile a une « sémantique d'acquisition »; autrement dit, il est garanti qu'il se produise avant toute référence à la mémoire qui se trouve après dans la séquence d'instructions.
- L'écriture d'un champ volatile est appelée une ***écriture volatile***. Une écriture volatile a une « sémantique de version »; autrement dit, il est garanti qu'il se produira après toute référence de mémoire avant l'instruction d'écriture dans la séquence d'instructions.

Ces restrictions garantissent que tous les threads observent les écritures volatiles effectuées par un autre thread dans l'ordre dans lequel elles ont été effectuées. Une implémentation conforme n'est pas requise pour fournir un classement total unique des écritures volatiles, comme vu par tous les threads d'exécution. Le type d'un champ volatile doit être l'un des suivants :

- *Reference_type*.
- Type,,,,,, `byte` `sbyte` `short` `ushort` `int` `uint` `char` , `float` ,, `bool` `System.IntPtr` ou `System.UIntPtr` .
- *Enum_type* ayant un type de base enum de,,,, `byte` `sbyte` `short` `ushort` `int` ou `uint` .

L'exemple

```C#
using System;
using System.Threading;

class Test
{
    public static int result;
    public static volatile bool finished;
```

```
    static void Thread2() {
        result = 143;
        finished = true;
    }

    static void Main() {
        finished = false;

        // Run Thread2() in a new thread
        new Thread(new ThreadStart(Thread2)).Start();

        // Wait for Thread2 to signal that it has a result by setting
        // finished to true.
        for (;;) {
            if (finished) {
                Console.WriteLine("result = {0}", result);
                return;
            }
        }
    }
}
```

génère cette sortie :

```
Console
```

```
result = 143
```

Dans cet exemple, la méthode `Main` démarre un nouveau thread qui exécute la méthode `Thread2` . Cette méthode stocke une valeur dans un champ non volatile appelé `result` , puis stocke `true` dans le champ volatile `finished` . Le thread principal attend que le champ `finished` soit défini sur `true` , puis lit le champ `result` . Étant donné que `finished` a été déclaré `volatile` , le thread principal doit lire la valeur `143` du champ `result` . Si le champ `finished` n'a pas été déclaré `volatile` , il est possible que le magasin `result` soit visible par le thread principal après le magasin `finished` , et par conséquent, pour que le thread principal Lise la valeur `0` à partir du champ `result` . La Déclaration `finished` en tant que `volatile` champ empêche toute incohérence.

Initialisation de champ

La valeur initiale d'un champ, qu'il s'agisse d'un champ statique ou d'un champ d'instance, est la valeur par défaut (valeurs par défaut) du type du champ. Il n'est pas possible d'observer la valeur d'un champ avant l'initialisation par défaut, et un champ n'est donc jamais « non initialisé ». L'exemple

```
```

C#

```
using System;

class Test
{
    static bool b;
    int i;

    static void Main() {
        Test t = new Test();
        Console.WriteLine("b = {0}, i = {1}", b, t.i);
    }
}
```

génère la sortie

Console

```
b = False, i = 0
```

étant donné que `b` et `i` sont initialisés automatiquement avec les valeurs par défaut.

Initialiseurs de variables

Les déclarations de champ peuvent inclure des *variable_initializer* s. Pour les champs statiques, les initialiseurs de variable correspondent aux instructions d'assignation qui sont exécutées pendant l'initialisation de la classe. Pour les champs d'instance, les initialiseurs de variable correspondent aux instructions d'assignation qui sont exécutées lors de la création d'une instance de la classe.

L'exemple

C#

```
using System;

class Test
{
    static double x = Math.Sqrt(2.0);
    int i = 100;
    string s = "Hello";

    static void Main() {
        Test a = new Test();
        Console.WriteLine("x = {0}, i = {1}, s = {2}", x, a.i, a.s);
    }
}
```

génère la sortie

```
Console
```

```
x = 1.4142135623731, i = 100, s = Hello
```

parce qu'une assignation à `x` se produit lorsque des initialiseurs de champs statiques exécutent des assignations à `i` et `s` se produisent lorsque les initialiseurs de champ d'instance s'exécutent.

L'initialisation de la valeur par défaut décrite dans initialisation de champ se produit pour tous les champs, y compris les champs qui ont des initialiseurs de variable. Ainsi, lorsqu'une classe est initialisée, tous les champs statiques de cette classe sont d'abord initialisés à leurs valeurs par défaut, puis les initialiseurs de champs statiques sont exécutés dans l'ordre textuel. De même, lorsqu'une instance d'une classe est créée, tous les champs d'instance dans cette instance sont d'abord initialisés à leurs valeurs par défaut, puis les initialiseurs de champ d'instance sont exécutés dans l'ordre textuel.

Il est possible que les champs statiques avec des initialiseurs de variable soient observés dans leur état de valeur par défaut. Toutefois, cela est fortement déconseillé en tant que question de style. L'exemple

```
C#
```

```
using System;

class Test
{
    static int a = b + 1;
    static int b = a + 1;

    static void Main() {
        Console.WriteLine("a = {0}, b = {1}", a, b);
    }
}
```

présente ce comportement. Malgré les définitions circulaires de a et b, le programme est valide. Elle génère la sortie

```
Console
```

```
a = 1, b = 2
```

étant donné que les champs statiques `a` et `b` sont initialisés à `0` (valeur par défaut pour `int`) avant l'exécution de leurs initialiseurs. Lorsque l'initialiseur pour `a` s'exécute, la valeur de `b` est égale à zéro, et par conséquent, `a` est initialisé à `1` . Lorsque l'initialiseur pour `b` s'exécute, la valeur de `a` est déjà `1` , et `b` par conséquent, est initialisé à `2` .

Initialisation de champ statique

Les initialiseurs de variable de champ statique d'une classe correspondent à une séquence d'assignations qui sont exécutées dans l'ordre textuel dans lequel elles apparaissent dans la déclaration de classe. Si un constructeur statique (constructeurs statiques) existe dans la classe, l'exécution des initialiseurs de champ statiques se produit immédiatement avant l'exécution de ce constructeur statique. Sinon, les initialiseurs de champs statiques sont exécutés à un moment dépendant de l'implémentation avant la première utilisation d'un champ statique de cette classe. L'exemple

C#

```
using System;

class Test
{
    static void Main() {
        Console.WriteLine("{0} {1}", B.Y, A.X);
    }

    public static int F(string s) {
        Console.WriteLine(s);
        return 1;
    }
}

class A
{
    public static int X = Test.F("Init A");
}

class B
{
    public static int Y = Test.F("Init B");
}
```

peut produire la sortie :

Console

```
Init A
Init B
1 1
```

ou la sortie :

Console

```
Init B
Init A
1 1
```

étant donné que l'exécution de l' `X` initialiseur et `Y` de l'initialiseur de est susceptible de se produire dans l'un ou l'autre ordre, il ne doit se produire que pour les références à ces champs. Toutefois, dans l'exemple :

C#

```
using System;

class Test
{
    static void Main() {
        Console.WriteLine("{0} {1}", B.Y, A.X);
    }

    public static int F(string s) {
        Console.WriteLine(s);
        return 1;
    }
}

class A
{
    static A() {}

    public static int X = Test.F("Init A");
}

class B
{
    static B() {}

    public static int Y = Test.F("Init B");
}
```

la sortie doit être :

```
```

Console

```
Init B
Init A
1 1
```

étant donné que les règles de moment où les constructeurs statiques s'exécutent (comme défini dans les constructeurs statiques) fournissent ce `B` constructeur statique (et par conséquent, les `B` initialiseurs de champ statiques) doivent exécuter avant `A` les initialiseurs de champ et de constructeur statique de.

Initialisation du champ d'instance

Les initialiseurs de variable de champ d'instance d'une classe correspondent à une séquence d'assignations qui sont exécutées immédiatement après l'entrée de l'un des constructeurs d'instance (initialiseurs de constructeur) de cette classe. Les initialiseurs de variable sont exécutés dans l'ordre textuel dans lequel ils apparaissent dans la déclaration de classe. Le processus de création et d'initialisation de l'instance de classe est décrit plus en détail dans les constructeurs d'instance.

Un initialiseur de variable pour un champ d'instance ne peut pas faire référence à l'instance en cours de création. Par conséquent, il s'agit d'une erreur de compilation pour référencer `this` dans un initialiseur de variable, car il s'agit d'une erreur de compilation pour qu'un initialiseur de variable référence un membre d'instance par le biais d'un *simple_name*. Dans l'exemple

C#

```
class A
{
    int x = 1;
    int y = x + 1;        // Error, reference to instance member of this
}
```

l'initialiseur de variable pour `y` génère une erreur de compilation, car il fait référence à un membre de l'instance en cours de création.

Méthodes

Une *__méthode__ _ est un membre qui implémente un calcul ou une action qui peut être effectuée par un objet ou une classe. Les méthodes sont déclarées à l'aide de _method_declaration * s :

antlr

```
method_declaration
    : method_header method_body
    ;

method_header
    : attributes? method_modifier* 'partial'? return_type member_name
type_parameter_list?
      '(' formal_parameter_list? ')' type_parameter_constraints_clause*
    ;

method_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'static'
    | 'virtual'
    | 'sealed'
    | 'override'
    | 'abstract'
    | 'extern'
    | 'async'
    | method_modifier_unsafe
    ;

return_type
    : type
    | 'void'
    ;

member_name
    : identifier
    | interface_type '.' identifier
    ;

method_body
    : block
    | '=>' expression ';'
    | ';'
    ;
```

Une *method_declaration* peut inclure un ensemble d' *attributs* (attributs) et une combinaison valide des quatre modificateurs d'accès (modificateurs d'accès), `new` (le nouveau modificateur), `static` (méthodes statiques et d'instance), `virtual` (méthodes virtuelles), (méthodes de `override` substitution), `sealed` (méthodes sealed), (`abstract` méthodes abstraites) et `extern` modificateurs (méthodes externes).

Une déclaration a une combinaison valide de modificateurs si toutes les conditions suivantes sont vraies :

- La déclaration comprend une combinaison valide de modificateurs d'accès (modificateurs d'accès).
- La déclaration n'inclut pas le même modificateur plusieurs fois.
- La déclaration comprend au plus l'un des modificateurs suivants : `static` , `virtual` et `override` .
- La déclaration comprend au plus l'un des modificateurs suivants : `new` et `override` .
- Si la déclaration inclut le `abstract` modificateur, la déclaration n'inclut pas l'un des modificateurs suivants : `static` , `virtual` `sealed` ou `extern` .
- Si la déclaration inclut le `private` modificateur, la déclaration n'inclut pas l'un des modificateurs suivants : `virtual` , `override` ou `abstract` .
- Si la déclaration comprend le `sealed` modificateur, la déclaration comprend également le `override` modificateur.
- Si la déclaration inclut le `partial` modificateur, elle n'inclut pas l'un des modificateurs suivants : `new` , `public` , `protected` , `internal` , `private` , `virtual` , `sealed` , `override` , `abstract` ou `extern` .

Une méthode qui a le `async` modificateur est une fonction Async et suit les règles décrites dans fonctions Async.

La *return_type* d'une déclaration de méthode spécifie le type de la valeur calculée et retournée par la méthode. Le *return_type* est `void` si la méthode ne retourne pas de valeur. Si la déclaration comprend le `partial` modificateur, le type de retour doit être `void` .

Le *Member_Name* spécifie le nom de la méthode. À moins que la méthode ne soit une implémentation de membre d'interface explicite (implémentations de membres d'interface explicites), le *Member_Name* est simplement un *identificateur*. Pour une implémentation de membre d'interface explicite, le *Member_Name* se compose d'un *INTERFACE_TYPE* suivi d'un « `.` » et d'un *identificateur*.

Le *type_parameter_list* facultatif spécifie les paramètres de type de la méthode (paramètres de type). Si une *type_parameter_list* est spécifiée, la méthode est une ***méthode générique** _. Si la méthode a un `extern` modificateur, un _type_parameter_list * ne peut pas être spécifié.

Le *formal_parameter_list* facultatif spécifie les paramètres de la méthode (paramètres de méthode).

Les *type_parameter_constraints_clause* facultatives spécifient des contraintes sur des paramètres de type individuels (contraintes de paramètre de type) et peuvent être spécifiés uniquement si une *type_parameter_list* est également fournie, et si la méthode n'a pas de `override` modificateur.

Les *return_type* et chacun des types référencés dans le *formal_parameter_list* d'une méthode doivent être au moins aussi accessibles que la méthode elle-même (contraintes d'accessibilité).

Le *method_body* est un point-virgule, un ***corps d'instruction*** *_ ou un corps d'* *_expression*.. Un corps d'instruction se compose d'un _block *, qui spécifie les instructions à exécuter lorsque la méthode est appelée. Un corps d'expression se compose de `=>` suivis d'une *expression* et d'un point-virgule, et désigne une expression unique à exécuter lorsque la méthode est appelée.

Pour `abstract` `extern` les méthodes et, le *method_body* se compose simplement d'un point-virgule. Pour `partial` les méthodes, le *method_body* peut être constitué soit d'un point-virgule, d'un corps de bloc ou d'un corps d'expression. Pour toutes les autres méthodes, le *method_body* est soit un corps de bloc, soit un corps d'expression.

Si le *method_body* se compose d'un point-virgule, la déclaration peut ne pas inclure le `async` modificateur.

Le nom, la liste des paramètres de type et la liste de paramètres formels d'une méthode définissent la signature (signatures et surcharge) de la méthode. Plus précisément, la signature d'une méthode se compose de son nom, du nombre de paramètres de type et du nombre, des modificateurs et des types de ses paramètres formels. Pour ces raisons, tout paramètre de type de la méthode qui se produit dans le type d'un paramètre formel est identifié par son nom, mais par sa position ordinale dans la liste d'arguments de type de la méthode. Le type de retour ne fait pas partie de la signature d'une méthode, ni les noms des paramètres de type ni les paramètres formels.

Le nom d'une méthode doit être différent des noms de toutes les autres méthodes non déclarées dans la même classe. En outre, la signature d'une méthode doit être différente des signatures de toutes les autres méthodes déclarées dans la même classe, et deux méthodes déclarées dans la même classe ne peuvent pas avoir de signatures qui diffèrent uniquement par `ref` et `out` .

Les *type_parameter* de la méthode se trouvent dans la portée de l' *method_declaration*, et peuvent être utilisés pour former des types dans cette portée dans *return_type*, *method_body* et *type_parameter_constraints_clause*, mais pas dans les *attributs*.

Tous les paramètres formels et les paramètres de type doivent avoir des noms différents.

Paramètres de méthode

Les paramètres d'une méthode, le cas échéant, sont déclarés par le *formal_parameter_list* de la méthode.

antlr

```
formal_parameter_list
    : fixed_parameters
    | fixed_parameters ',' parameter_array
    | parameter_array
    ;

fixed_parameters
    : fixed_parameter (',' fixed_parameter)*
    ;

fixed_parameter
    : attributes? parameter_modifier? type identifier default_argument?
    ;

default_argument
    : '=' expression
    ;

parameter_modifier
    : 'ref'
    | 'out'
    | 'this'
    ;

parameter_array
    : attributes? 'params' array_type identifier
    ;
```

La liste de paramètres formels se compose d'un ou de plusieurs paramètres séparés par des virgules dont seule la dernière peut être un *parameter_array*.

Un *fixed_parameter* se compose d'un ensemble facultatif d' *attributs* (attributs), d'un facultatif ou d'un `ref` `out` `this` modificateur, d'un *type*, d'un *identificateur* et d'un *default_argument* facultatif. Chaque *fixed_parameter* déclare un paramètre du type donné avec le nom donné. Le `this` modificateur désigne la méthode en tant que méthode d'extension et est uniquement autorisé sur le premier paramètre d'une méthode statique. Les méthodes d'extension sont décrites plus en détail dans méthodes d'extension.

Un *fixed_parameter* avec un *default_argument* est appelé ***paramètre facultatif** , *alors qu'un fixed_parameter * sans default_argument est un **paramètre obligatoire***.. Un

paramètre obligatoire ne peut pas apparaître après un paramètre facultatif dans un _formal_parameter_list *.

Un `ref` `out` paramètre ou ne peut pas avoir de *default_argument*. L' *expression* dans un *default_argument* doit être l'une des suivantes :

- *constant_expression*
- expression de la forme `new S()` où `S` est un type valeur
- expression de la forme `default(S)` où `S` est un type valeur

L' *expression* doit être implicitement convertible à l'aide d'une conversion d'identité ou de valeurs NULL dans le type du paramètre.

Si des paramètres facultatifs se produisent dans une déclaration de méthode partielle d'implémentation (méthodes partielles), une implémentation de membre d'interface explicite (implémentations de membres d'interface explicites) ou dans une déclaration d'indexeur à un seul paramètre (indexeurs), le compilateur doit fournir un avertissement, car ces membres ne peuvent jamais être appelés d'une manière qui autorise l'omission des arguments.

Un *parameter_array* se compose d'un ensemble facultatif d' *attributs* (attributs), d'un `params` modificateur, d'un *array_type* et d'un *identificateur*. Un tableau de paramètres déclare un paramètre unique du type de tableau donné avec le nom donné. Le *array_type* d'un tableau de paramètres doit être un type tableau unidimensionnel (types tableau). Dans un appel de méthode, un tableau de paramètres autorise la spécification d'un seul argument du type de tableau donné, ou elle permet de spécifier zéro ou plusieurs arguments du type d'élément de tableau. Les tableaux de paramètres sont décrits plus en détail dans les tableaux de paramètres.

Une *parameter_array* peut se produire après un paramètre facultatif, mais ne peut pas avoir de valeur par défaut ; l'omission des arguments d'un *parameter_array* entraînerait la création d'un tableau vide.

L'exemple suivant illustre différents genres de paramètres :

C#

```
public void M(
    ref int      i,
    decimal      d,
    bool         b = false,
    bool?        n = false,
    string       s = "Hello",
    object       o = null,
    T            t = default(T),
```

```
    params int[] a
) { }
```

Dans l' *formal_parameter_list* pour `M` , `i` est un paramètre ref obligatoire, `d` est un paramètre de valeur obligatoire `b` , `s` , `o` et `t` sont des paramètres de valeur facultatifs et `a` est un tableau de paramètres.

Une déclaration de méthode crée un espace de déclaration distinct pour les paramètres, les paramètres de type et les variables locales. Les noms sont introduits dans cet espace de déclaration par la liste de paramètres de type et la liste de paramètres formels de la méthode et par les déclarations de variable locale dans le *bloc* de la méthode. Il y a une erreur pour que deux membres d'un espace de déclaration de méthode aient le même nom. L'espace de déclaration de méthode et l'espace de déclaration de variable locale d'un espace de déclaration imbriquée doivent contenir des éléments portant le même nom.

Un appel de méthode (appel de méthode) crée une copie, spécifique à cet appel, des paramètres formels et des variables locales de la méthode, et la liste d'arguments de l'appel assigne des valeurs ou des références de variables aux paramètres formels nouvellement créés. Dans le *bloc* d'une méthode, les paramètres formels peuvent être référencés par leurs identificateurs dans *simple_name* expressions (noms simples).

Il existe quatre genres de paramètres formels :

- Les paramètres de valeur, qui sont déclarés sans modificateurs.
- Les paramètres de référence, qui sont déclarés avec le `ref` modificateur.
- Les paramètres de sortie, qui sont déclarés avec le `out` modificateur.
- Tableaux de paramètres, qui sont déclarés avec le `params` modificateur.

Comme décrit dans signatures et surcharge, les `ref` `out` modificateurs et font partie de la signature d'une méthode, mais le `params` modificateur n'est pas.

Paramètres de valeur

Un paramètre déclaré sans modificateurs est un paramètre de valeur. Un paramètre de valeur correspond à une variable locale qui obtient sa valeur initiale de l'argument correspondant fourni dans l'appel de méthode.

Quand un paramètre formel est un paramètre de valeur, l'argument correspondant dans un appel de méthode doit être une expression implicitement convertible (conversions implicites) en type de paramètre formel.

Une méthode est autorisée à assigner de nouvelles valeurs à un paramètre de valeur. Ces affectations affectent uniquement l'emplacement de stockage local représenté par le paramètre de valeur, elles n'ont aucun effet sur l'argument réel fourni dans l'appel de méthode.

Paramètres de référence

Un paramètre déclaré avec un `ref` modificateur est un paramètre de référence. Contrairement à un paramètre de valeur, un paramètre de référence ne crée pas un nouvel emplacement de stockage. Au lieu de cela, un paramètre de référence représente le même emplacement de stockage que la variable donnée comme argument dans l'appel de méthode.

Quand un paramètre formel est un paramètre de référence, l'argument correspondant dans un appel de méthode doit être constitué du mot clé `ref` suivi d'un *variable_reference* (règles précises pour déterminer l'assignation définie) du même type que le paramètre formel. Une variable doit être assignée définitivement pour pouvoir être passée en tant que paramètre de référence.

Dans une méthode, un paramètre de référence est toujours considéré comme étant définitivement assigné.

Une méthode déclarée en tant qu'itérateur (itérateurs) ne peut pas avoir de paramètres de référence.

L'exemple

C#

```
using System;

class Test
{
    static void Swap(ref int x, ref int y) {
        int temp = x;
        x = y;
        y = temp;
    }

    static void Main() {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        Console.WriteLine("i = {0}, j = {1}", i, j);
    }
}
```

génère la sortie

```
Console
```

```
i = 2, j = 1
```

Pour l'appel de `Swap` dans `Main` , `x` représente `i` et `y` représente `j` . Ainsi, l'appel a pour effet de permuter les valeurs de `i` et `j` .

Dans une méthode qui accepte des paramètres de référence, il est possible que plusieurs noms représentent le même emplacement de stockage. Dans l'exemple

```
C#
```

```
class A
{
    string s;

    void F(ref string a, ref string b) {
        s = "One";
        a = "Two";
        b = "Three";
    }

    void G() {
        F(ref s, ref s);
    }
}
```

l'appel de `F` dans `G` passe une référence à `s` pour `a` et `b` . Ainsi, pour cet appel, les noms `s` , `a` et `b` font tous référence au même emplacement de stockage, et les trois assignations modifient toutes le champ d'instance `s` .

Paramètres de sortie

Un paramètre déclaré avec un `out` modificateur est un paramètre de sortie. À l'instar d'un paramètre de référence, un paramètre de sortie ne crée pas un nouvel emplacement de stockage. Au lieu de cela, un paramètre de sortie représente le même emplacement de stockage que la variable donnée comme argument dans l'appel de la méthode.

Quand un paramètre formel est un paramètre de sortie, l'argument correspondant dans un appel de méthode doit être constitué du mot clé `out` suivi d'un *variable_reference* (règles précises pour déterminer l'assignation définie) du même type que le paramètre formel. Une variable n'a pas besoin d'être définitivement assignée avant de pouvoir être

transmise en tant que paramètre de sortie, mais après un appel où une variable a été passée comme paramètre de sortie, la variable est considérée comme assignée de manière définitive.

Dans une méthode, tout comme une variable locale, un paramètre de sortie est considéré initialement comme non assigné et doit être définitivement assigné avant que sa valeur ne soit utilisée.

Chaque paramètre de sortie d'une méthode doit être assigné définitivement avant le retour de la méthode.

Une méthode déclarée en tant que méthode partielle (méthodes partielles) ou itérateur (itérateurs) ne peut pas avoir de paramètres de sortie.

Les paramètres de sortie sont généralement utilisés dans les méthodes qui produisent plusieurs valeurs de retour. Par exemple :

C#

```
using System;

class Test
{
    static void SplitPath(string path, out string dir, out string name) {
        int i = path.Length;
        while (i > 0) {
            char ch = path[i - 1];
            if (ch == '\\' || ch == '/' || ch == ':') break;
            i--;
        }
        dir = path.Substring(0, i);
        name = path.Substring(i);
    }

    static void Main() {
        string dir, name;
        SplitPath("c:\\Windows\\System\\hello.txt", out dir, out name);
        Console.WriteLine(dir);
        Console.WriteLine(name);
    }
}
```

L'exemple produit la sortie :

Console

```
c:\Windows\System\
hello.txt
```

Notez que les `dir` `name` variables et peuvent être non assignées avant d'être passées à `SplitPath` , et qu'elles sont considérées comme définitivement assignées après l'appel.

Tableaux de paramètres

Un paramètre déclaré avec un `params` modificateur est un tableau de paramètres. Si une liste de paramètres formels comprend un tableau de paramètres, elle doit être le dernier paramètre de la liste et doit être de type tableau unidimensionnel. Par exemple, les types `string[]` et `string[][]` peuvent être utilisés comme type d'un tableau de paramètres, mais le type `string[,]` ne peut pas. Il n'est pas possible de combiner le `params` modificateur avec les modificateurs `ref` et `out` .

Un tableau de paramètres permet de spécifier des arguments de l'une des deux manières suivantes dans un appel de méthode :

- L'argument donné pour un tableau de paramètres peut être une expression unique implicitement convertible (conversions implicites) en type tableau de paramètres. Dans ce cas, le tableau de paramètres agit précisément comme un paramètre de valeur.
- L'appel peut également spécifier zéro ou plusieurs arguments pour le tableau de paramètres, où chaque argument est une expression implicitement convertible (conversions implicites) en type d'élément du tableau de paramètres. Dans ce cas, l'appel crée une instance du type de tableau de paramètres avec une longueur correspondant au nombre d'arguments, initialise les éléments de l'instance de tableau avec les valeurs d'argument données et utilise l'instance de tableau nouvellement créée comme argument réel.

À l'exception de l'autorisation d'un nombre variable d'arguments dans un appel, un tableau de paramètres équivaut précisément à un paramètre de valeur (paramètres de valeur) du même type.

L'exemple

C#

```
using System;

class Test
{
    static void F(params int[] args) {
        Console.Write("Array contains {0} elements:", args.Length);
        foreach (int i in args)
            Console.Write(" {0}", i);
        Console.WriteLine();
```

```
    }

    static void Main() {
        int[] arr = {1, 2, 3};
        F(arr);
        F(10, 20, 30, 40);
        F();
    }
}
```

génère la sortie

Console

```
Array contains 3 elements: 1 2 3
Array contains 4 elements: 10 20 30 40
Array contains 0 elements:
```

La première invocation de `F` passe simplement le tableau `a` en tant que paramètre de valeur. Le deuxième appel de `F` crée automatiquement un élément à quatre éléments `int[]` avec les valeurs d'élément données et passe cette instance de tableau en tant que paramètre de valeur. De même, le troisième appel de `F` crée un élément de zéro `int[]` et passe cette instance en tant que paramètre de valeur. Le deuxième et le troisième appel sont exactement équivalents à l'écriture :

C#

```
F(new int[] {10, 20, 30, 40});
F(new int[] {});
```

Lors de l'exécution de la résolution de surcharge, une méthode avec un tableau de paramètres peut s'appliquer soit dans sa forme normale, soit dans sa forme développée (fonction membre applicable). La forme développée d'une méthode est disponible uniquement si la forme normale de la méthode n'est pas applicable et uniquement si une méthode applicable avec la même signature que la forme développée n'est pas déjà déclarée dans le même type.

L'exemple

C#

```
using System;

class Test
{
    static void F(params object[] a) {
```

```
        Console.WriteLine("F(object[])");
    }

    static void F() {
        Console.WriteLine("F()");
    }

    static void F(object a0, object a1) {
        Console.WriteLine("F(object,object)");
    }

    static void Main() {
        F();
        F(1);
        F(1, 2);
        F(1, 2, 3);
        F(1, 2, 3, 4);
    }
}
```

génère la sortie

Console

```
F();
F(object[]);
F(object,object);
F(object[]);
F(object[]);
```

Dans l'exemple, deux des formes étendues possibles de la méthode avec un tableau de paramètres sont déjà incluses dans la classe en tant que méthodes normales. Ces formulaires développés ne sont donc pas pris en compte lors de l'exécution de la résolution de surcharge, et les premier et troisième appels de méthode sélectionnent donc les méthodes normales. Quand une classe déclare une méthode avec un tableau de paramètres, il n'est pas rare d'inclure également certains des formulaires développés comme méthodes normales. En procédant ainsi, il est possible d'éviter l'allocation d'une instance de tableau qui se produit lorsqu'une forme développée d'une méthode avec un tableau de paramètres est appelée.

Lorsque le type d'un tableau de paramètres est `object[]` , une ambiguïté potentielle se produit entre la forme normale de la méthode et la forme étendue pour un `object` paramètre unique. L'ambiguïté est due au fait qu'un `object[]` est lui-même convertible implicitement en type `object` . Toutefois, l'ambiguïté ne présente aucun problème, car elle peut être résolue en insérant un cast si nécessaire.

L'exemple

C#

```
using System;

class Test
{
    static void F(params object[] args) {
        foreach (object o in args) {
            Console.Write(o.GetType().FullName);
            Console.Write(" ");
        }
        Console.WriteLine();
    }

    static void Main() {
        object[] a = {1, "Hello", 123.456};
        object o = a;
        F(a);
        F((object)a);
        F(o);
        F((object[])o);
    }
}
```

génère la sortie

Console

```
System.Int32 System.String System.Double
System.Object[]
System.Object[]
System.Int32 System.String System.Double
```

Dans le premier et le dernier appel de `F` , la forme normale de `F` est applicable, car il existe une conversion implicite du type d'argument vers le type de paramètre (les deux sont de type `object[]`). Ainsi, la résolution de surcharge sélectionne la forme normale de `F` et l'argument est passé en tant que paramètre de valeur régulière. Dans le deuxième et le troisième appel, la forme normale de `F` n'est pas applicable, car il n'existe aucune conversion implicite du type d'argument vers le type de paramètre (le type `object` ne peut pas être converti implicitement en type `object[]`). Toutefois, la forme développée de `F` est applicable, donc elle est sélectionnée par la résolution de surcharge. Par conséquent, un élément `object[]` unique est créé par l'appel, et l'élément unique du tableau est initialisé avec la valeur d'argument donnée (qui est elle-même une référence à un `object[]`).

Méthodes statiques et d'instance

Lorsqu'une déclaration de méthode comprend un `static` modificateur, cette méthode est dite statique. Quand aucun `static` modificateur n'est présent, la méthode est considérée comme une méthode d'instance.

Une méthode statique ne fonctionne pas sur une instance spécifique, et il s'agit d'une erreur de compilation pour faire référence à `this` dans une méthode statique.

Une méthode d'instance opère sur une instance donnée d'une classe, et cette instance est accessible en tant que `this` (cet accès).

Quand une méthode est référencée dans un *member_access* (accès aux membres) du formulaire `E.M` , si `M` est une méthode statique, `E` doit désigner un type contenant `M` , et si `M` est une méthode d'instance, `E` doit désigner une instance d'un type contenant `M` .

Les différences entre les membres statiques et les membres d'instance sont abordées plus en détail dans les membres statiques et d'instance.

Méthodes virtuelles

Lorsqu'une déclaration de méthode d'instance comprend un `virtual` modificateur, cette méthode est dite méthode virtuelle. Quand aucun `virtual` modificateur n'est présent, la méthode est dite non virtuelle.

L'implémentation d'une méthode non virtuelle est indifférente : l'implémentation est la même que la méthode soit appelée sur une instance de la classe dans laquelle elle est déclarée ou une instance d'une classe dérivée. En revanche, l'implémentation d'une méthode virtuelle peut être remplacée par des classes dérivées. Le processus de remplacement de l'implémentation d'une méthode virtuelle héritée est connu sous le nom de ***substitution*** de cette méthode (méthodes override).

Dans un appel de méthode virtuelle, le *__type au moment__ de l'exécution _ de l'instance pour laquelle cet appel se produit détermine l'implémentation de méthode réelle à appeler. Dans un appel de méthode non virtuel, le _ *type au moment* de la compilation * de l'instance est le facteur déterminant. En termes précis, quand une méthode nommée `N` est appelée avec une liste d'arguments `A` sur une instance avec un type au moment de la compilation `C` et un type au moment de `R` l'exécution (où `R` est soit `C` ou une classe dérivée de `C`), l'appel est traité comme suit :

- Tout d'abord, la résolution de surcharge est appliquée à `C` , `N` et `A` , pour sélectionner une méthode spécifique `M` à partir de l'ensemble de méthodes déclaré dans et héritée par `C` . Cela est décrit dans appels de méthode.

- Ensuite, si `M` est une méthode non virtuelle, `M` est appelée.
- Sinon, `M` est une méthode virtuelle et l'implémentation la plus dérivée de `M` par rapport à `R` est appelée.

Pour chaque méthode virtuelle déclarée dans ou héritée par une classe, il existe une ***implémentation la plus dérivée*** de la méthode par rapport à cette classe. L'implémentation la plus dérivée d'une méthode virtuelle en `M` ce qui concerne une classe `R` est déterminée comme suit :

- Si `R` contient la `virtual` déclaration d'introduction de `M` , il s'agit de l'implémentation la plus dérivée de `M` .
- Sinon, si `R` contient un `override` de `M` , il s'agit de l'implémentation la plus dérivée de `M` .
- Sinon, l'implémentation la plus dérivée de `M` par rapport à `R` est identique à l'implémentation la plus dérivée de `M` par rapport à la classe de base directe de `R` .

L'exemple suivant illustre les différences entre les méthodes virtuelles et non virtuelles :

C#

```
using System;

class A
{
    public void F() { Console.WriteLine("A.F"); }

    public virtual void G() { Console.WriteLine("A.G"); }
}

class B: A
{
    new public void F() { Console.WriteLine("B.F"); }

    public override void G() { Console.WriteLine("B.G"); }
}

class Test
{
    static void Main() {
        B b = new B();
        A a = b;
        a.F();
        b.F();
        a.G();
        b.G();
    }
}
```

Dans l'exemple, `A` introduit une méthode non virtuelle `F` et une méthode virtuelle `G`. La classe `B` introduit une nouvelle méthode non virtuelle `F`, masquant ainsi la héritée `F` et substitue également la méthode héritée `G`. L'exemple produit la sortie :

Console

```
A.F
B.F
B.G
B.G
```

Notez que l'instruction `a.G()` appelle `B.G`, et non `A.G`. Cela est dû au fait que le type au moment de l'exécution de l'instance (qui est `B`), et non le type au moment de la compilation de l'instance (c'est-à-dire `A`), détermine l'implémentation réelle de la méthode à appeler.

Étant donné que les méthodes sont autorisées à masquer les méthodes héritées, il est possible qu'une classe contienne plusieurs méthodes virtuelles avec la même signature. Cela ne présente pas de problème d'ambiguïté, puisque toutes les méthodes sauf la plus dérivée sont masquées. Dans l'exemple

C#

```
using System;

class A
{
    public virtual void F() { Console.WriteLine("A.F"); }
}

class B: A
{
    public override void F() { Console.WriteLine("B.F"); }
}

class C: B
{
    new public virtual void F() { Console.WriteLine("C.F"); }
}

class D: C
{
    public override void F() { Console.WriteLine("D.F"); }
}

class Test
{
    static void Main() {
        D d = new D();
```

```
        A a = d;
        B b = d;
        C c = d;
        a.F();
        b.F();
        c.F();
        d.F();
    }
}
```

les `C` `D` classes et contiennent deux méthodes virtuelles avec la même signature : celle introduite `A` par et celle introduite par `C` . La méthode introduite par `C` masque la méthode héritée de `A` . Ainsi, la déclaration override de `D` substitue la méthode introduite par `C` , et il n'est pas possible pour `D` de substituer la méthode introduite par `A` . L'exemple produit la sortie :

Console

```
B.F
B.F
D.F
D.F
```

Notez qu'il est possible d'appeler la méthode virtuelle masquée en accédant à une instance de `D` via un type moins dérivé dans lequel la méthode n'est pas masquée.

Substituer les méthodes

Lorsqu'une déclaration de méthode d'instance comprend un `override` modificateur, la méthode est considérée comme une ***méthode de substitution***. Une méthode override remplace une méthode virtuelle héritée avec la même signature. Là où une déclaration de méthode virtuelle présente une nouvelle méthode, une déclaration de méthode de substitution spécialise une méthode virtuelle héritée existante en fournissant une nouvelle implémentation de cette méthode.

La méthode substituée par une `override` déclaration est connue sous le nom de ***méthode de base substituée***. Pour une méthode de substitution `M` déclarée dans une classe `C` , la méthode de base substituée est déterminée en examinant chaque type de classe de base de `C` , en commençant par le type de classe de base directe de `C` et en continuant avec chaque type de classe de base directe successif, jusqu'à ce que dans un type de classe de base donné, au moins une méthode accessible soit localisée avec la même signature `M` Dans le cadre de la recherche de la méthode de base substituée, une méthode est considérée comme accessible si c'est le cas, si c'est le cas, `public`

`protected` `protected internal` ou si elle est `internal` et est déclarée dans le même programme que `C` .

Une erreur de compilation se produit sauf si toutes les conditions suivantes sont vraies pour une déclaration override :

- Une méthode de base substituée peut être localisée comme décrit ci-dessus.
- Il existe exactement une méthode de base substituée. Cette restriction n'a d'effet que si le type de classe de base est un type construit où la substitution d'arguments de type rend la signature de deux méthodes identique.
- La méthode de base substituée est une méthode virtuelle, abstraite ou de substitution. En d'autres termes, la méthode de base substituée ne peut pas être statique ou non virtuelle.
- La méthode de base substituée n'est pas une méthode sealed.
- La méthode de substitution et la méthode de base substituée ont le même type de retour.
- La déclaration override et la méthode de base substituée ont la même accessibilité déclarée. En d'autres termes, une déclaration override ne peut pas modifier l'accessibilité de la méthode virtuelle. Toutefois, si la méthode de base substituée est protégée interne et qu'elle est déclarée dans un assembly différent de l'assembly contenant la méthode de substitution, l'accessibilité déclarée de la méthode de substitution doit être protégée.
- La déclaration override ne spécifie pas de clauses de type-Parameter-Constraints. Au lieu de cela, les contraintes sont héritées de la méthode de base substituée. Notez que les contraintes qui sont des paramètres de type dans la méthode substituée peuvent être remplacées par des arguments de type dans la contrainte héritée. Cela peut entraîner des contraintes qui ne sont pas autorisées quand elles sont explicitement spécifiées, telles que des types valeur ou des types sealed.

L'exemple suivant montre comment les règles de substitution fonctionnent pour les classes génériques :

C#

```
abstract class C<T>
{
    public virtual T F() {...}
    public virtual C<T> G() {...}
    public virtual void H(C<T> x) {...}
}

class D: C<string>
{
    public override string F() {...}           // Ok
    public override C<string> G() {...}        // Ok
```

```
    public override void H(C<T> x) {...}        // Error, should be
C<string>
}

class E<T,U>: C<U>
{
    public override U F() {...}                 // Ok
    public override C<U> G() {...}              // Ok
    public override void H(C<T> x) {...}        // Error, should be C<U>
}
```

Une déclaration override peut accéder à la méthode de base substituée à l'aide d'un *base_access* (accès de base). Dans l'exemple

C#

```
class A
{
    int x;

    public virtual void PrintFields() {
        Console.WriteLine("x = {0}", x);
    }
}

class B: A
{
    int y;

    public override void PrintFields() {
        base.PrintFields();
        Console.WriteLine("y = {0}", y);
    }
}
```

l' `base.PrintFields()` appel dans `B` appelle la `PrintFields` méthode déclarée dans `A` . Une *base_access* désactive le mécanisme d'appel virtuel et traite simplement la méthode de base comme une méthode non virtuelle. Si l'appel a `B` été écrit `((A)this).PrintFields()` , il appellerait de manière récursive la `PrintFields` méthode déclarée dans `B` , pas celle qui est déclarée dans `A` , car `PrintFields` est virtuel et le type d'exécution de `((A)this)` est `B` .

`override`Une méthode ne peut se substituer à une autre méthode qu'en incluant un modificateur. Dans tous les autres cas, une méthode avec la même signature qu'une méthode héritée masque simplement la méthode héritée. Dans l'exemple

C#

```
class A
{
    public virtual void F() {}
}

class B: A
{
    public virtual void F() {}        // Warning, hiding inherited F()
}
```

la `F` méthode dans `B` n'inclut pas de `override` modificateur et, par conséquent, ne substitue pas la `F` méthode dans `A` . Au lieu de cela, la `F` méthode dans `B` masque la méthode dans `A` et un avertissement est signalé, car la déclaration n'inclut pas de `new` modificateur.

Dans l'exemple

C#

```
class A
{
    public virtual void F() {}
}

class B: A
{
    new private void F() {}        // Hides A.F within body of B
}

class C: B
{
    public override void F() {}    // Ok, overrides A.F
}
```

la `F` méthode dans `B` masque la méthode virtuelle `F` héritée de `A` . Étant donné que le nouveau `F` dans `B` a un accès privé, son étendue inclut uniquement le corps de classe de `B` et n'étend pas à `C` . Par conséquent, la déclaration de `F` dans `C` est autorisée à substituer l' `F` héritage de `A` .

Méthodes sealed

Lorsqu'une déclaration de méthode d'instance comprend un `sealed` modificateur, cette méthode est dite ***sealed***. Si une déclaration de méthode d'instance inclut le `sealed` modificateur, elle doit également inclure le `override` modificateur. L'utilisation du `sealed` modificateur empêche une classe dérivée de substituer davantage la méthode.

Dans l'exemple

C#

```
using System;

class A
{
    public virtual void F() {
        Console.WriteLine("A.F");
    }

    public virtual void G() {
        Console.WriteLine("A.G");
    }
}

class B: A
{
    sealed override public void F() {
        Console.WriteLine("B.F");
    }

    override public void G() {
        Console.WriteLine("B.G");
    }
}

class C: B
{
    override public void G() {
        Console.WriteLine("C.G");
    }
}
```

la classe `B` fournit deux méthodes override : une `F` méthode qui a le `sealed` modificateur et une `G` méthode qui ne le fait pas. `B`l'utilisation du sealed `modifier` empêche toute `C` substitution supplémentaire `F` .

Méthodes abstraites

Lorsqu'une déclaration de méthode d'instance comprend un `abstract` modificateur, cette méthode est dite ***abstraite***. Bien qu'une méthode abstraite soit implicitement également une méthode virtuelle, elle ne peut pas avoir de modificateur `virtual` .

Une déclaration de méthode abstraite introduit une nouvelle méthode virtuelle, mais ne fournit pas d'implémentation de cette méthode. Au lieu de cela, les classes dérivées non abstraites sont requises pour fournir leur propre implémentation en substituant cette

méthode. Étant donné qu'une méthode abstraite ne fournit aucune implémentation réelle, le *method_body* d'une méthode abstraite se compose simplement d'un point-virgule.

Les déclarations de méthode abstraite sont autorisées uniquement dans les classes abstraites (classes abstraites).

Dans l'exemple

```
C#

public abstract class Shape
{
    public abstract void Paint(Graphics g, Rectangle r);
}

public class Ellipse: Shape
{
    public override void Paint(Graphics g, Rectangle r) {
        g.DrawEllipse(r);
    }
}

public class Box: Shape
{
    public override void Paint(Graphics g, Rectangle r) {
        g.DrawRect(r);
    }
}
```

la `Shape` classe définit la notion abstraite d'un objet de forme géométrique qui peut se peindre lui-même. La `Paint` méthode est abstraite, car il n'y a pas d'implémentation significative par défaut. Les `Ellipse` `Box` classes et sont des `Shape` implémentations concrètes. Étant donné que ces classes ne sont pas abstraites, elles doivent substituer la `Paint` méthode et fournir une implémentation réelle.

Il s'agit d'une erreur au moment de la compilation pour qu'un *base_access* (accès de base) référence une méthode abstraite. Dans l'exemple

```
C#

abstract class A
{
    public abstract void F();
}

class B: A
{
    public override void F() {
```

```
        base.F();                       // Error, base.F is abstract
    }
}
```

une erreur de compilation est signalée pour l' `base.F()` appel, car elle fait référence à une méthode abstraite.

Une déclaration de méthode abstraite est autorisée à substituer une méthode virtuelle. Cela permet à une classe abstraite de forcer la réimplémentation de la méthode dans les classes dérivées et rend l'implémentation d'origine de la méthode indisponible. Dans l'exemple

C#

```
using System;

class A
{
    public virtual void F() {
        Console.WriteLine("A.F");
    }
}

abstract class B: A
{
    public abstract override void F();
}

class C: B
{
    public override void F() {
        Console.WriteLine("C.F");
    }
}
```

`A`la classe déclare une méthode virtuelle, `B` la classe remplace cette méthode par une méthode abstraite, et `C` la classe substitue la méthode abstraite pour fournir sa propre implémentation.

Méthodes externes

Lorsqu'une déclaration de méthode comprend un `extern` modificateur, cette méthode est dite « ***External Method** _ ». Les méthodes externes sont implémentées en externe, généralement à l'aide d'un langage autre que C#. Étant donné qu'une déclaration de méthode externe ne fournit aucune implémentation réelle, le _method_body * d'une

méthode externe se compose simplement d'un point-virgule. Une méthode externe ne peut pas être générique.

Le `extern` modificateur est généralement utilisé conjointement avec un `DllImport` attribut (interopérabilité avec les composants COM et Win32), ce qui permet d'implémenter des méthodes externes par des dll (bibliothèques de liens dynamiques). L'environnement d'exécution peut prendre en charge d'autres mécanismes par lesquels des implémentations de méthodes externes peuvent être fournies.

Quand une méthode externe inclut un `DllImport` attribut, la déclaration de méthode doit également inclure un `static` modificateur. Cet exemple illustre l'utilisation du `extern` modificateur et de l' `DllImport` attribut :

C#

```
using System.Text;
using System.Security.Permissions;
using System.Runtime.InteropServices;

class Path
{
    [DllImport("kernel32", SetLastError=true)]
    static extern bool CreateDirectory(string name, SecurityAttribute sa);

    [DllImport("kernel32", SetLastError=true)]
    static extern bool RemoveDirectory(string name);

    [DllImport("kernel32", SetLastError=true)]
    static extern int GetCurrentDirectory(int bufSize, StringBuilder buf);

    [DllImport("kernel32", SetLastError=true)]
    static extern bool SetCurrentDirectory(string name);
}
```

Méthodes partielles (récapitulatif)

Lorsqu'une déclaration de méthode comprend un `partial` modificateur, cette méthode est dite ***partielle***. Les méthodes partielles ne peuvent être déclarées que comme membres de types partiels (types partiels) et sont soumises à un certain nombre de restrictions. Les méthodes partielles sont décrites plus en détail dans les méthodes partielles.

Méthodes d'extension

Quand le premier paramètre d'une méthode comprend le `this` modificateur, cette méthode est considérée comme une ***méthode d'extension***. Les méthodes d'extension ne peuvent être déclarées que dans des classes statiques non imbriquées et non génériques. Le premier paramètre d'une méthode d'extension ne peut pas avoir de modificateurs autres que `this` , et le type de paramètre ne peut pas être un type pointeur.

Voici un exemple de classe statique qui déclare deux méthodes d'extension :

C#

```
public static class Extensions
{
    public static int ToInt32(this string s) {
        return Int32.Parse(s);
    }

    public static T[] Slice<T>(this T[] source, int index, int count) {
        if (index < 0 || count < 0 || source.Length - index < count)
            throw new ArgumentException();
        T[] result = new T[count];
        Array.Copy(source, index, result, 0, count);
        return result;
    }
}
```

Une méthode d'extension est une méthode statique normale. En outre, lorsque sa classe statique englobante est dans la portée, une méthode d'extension peut être appelée à l'aide de la syntaxe d'appel de méthode d'instance (appels de méthode d'extension), à l'aide de l'expression de récepteur comme premier argument.

Le programme suivant utilise les méthodes d'extension déclarées ci-dessus :

C#

```
static class Program
{
    static void Main() {
        string[] strings = { "1", "22", "333", "4444" };
        foreach (string s in strings.Slice(1, 2)) {
            Console.WriteLine(s.ToInt32());
        }
    }
}
```

La `Slice` méthode est disponible sur le `string[]` , et la `ToInt32` méthode est disponible sur `string` , car elles ont été déclarées comme méthodes d'extension. La signification du

programme est la même que celle qui suit, à l'aide d'appels de méthode statique ordinaires :

```C#
static class Program
{
    static void Main() {
        string[] strings = { "1", "22", "333", "4444" };
        foreach (string s in Extensions.Slice(strings, 1, 2)) {
            Console.WriteLine(Extensions.ToInt32(s));
        }
    }
}
```

Corps de la méthode

La *method_body* d'une déclaration de méthode se compose soit d'un corps de bloc, d'un corps d'expression ou d'un point-virgule.

Le ***type de résultat*** d'une méthode est `void` si le type de retour est `void` , ou si la méthode est asynchrone et que le type de retour est `System.Threading.Tasks.Task` . Sinon, le type de résultat d'une méthode non asynchrone est son type de retour et le type de résultat d'une méthode Async avec le type de retour `System.Threading.Tasks.Task<T>` est `T` .

Quand une méthode a un `void` type de résultat et un corps de bloc, les `return` instructions (l'instruction return) dans le bloc ne sont pas autorisées à spécifier une expression. Si l'exécution du bloc d'une méthode void se termine normalement (autrement dit, le contrôle circule à la fin du corps de la méthode), cette méthode retourne simplement à son appelant actuel.

Quand une méthode a un `void` résultat et un corps d'expression, l'expression `E` doit être un *statement_expression*, et le corps est exactement équivalent à un corps de bloc de la forme `{ E; }` .

Quand une méthode a un type de résultat non void et un corps de bloc, chaque `return` instruction du bloc doit spécifier une expression qui est implicitement convertible en type de résultat. Le point de terminaison d'un corps de bloc d'une méthode retournant une valeur ne doit pas être accessible. En d'autres termes, dans une méthode retournant une valeur avec un corps de bloc, le contrôle n'est pas autorisé à circuler à la fin du corps de la méthode.

Quand une méthode a un type de résultat non void et un corps d'expression, l'expression doit être implicitement convertible en type de résultat, et le corps est exactement équivalent à un corps de bloc de la forme `{ return E; }` .

Dans l'exemple

```C#
class A
{
    public int F() {}            // Error, return value required

    public int G() {
        return 1;
    }

    public int H(bool b) {
        if (b) {
            return 1;
        }
        else {
            return 0;
        }
    }

    public int I(bool b) => b ? 1 : 0;
}
```

la méthode qui retourne `F` une valeur génère une erreur au moment de la compilation, car le contrôle peut circuler à la fin du corps de la méthode. Les `G` `H` méthodes et sont correctes, car tous les chemins d'exécution possibles se terminent par une instruction return qui spécifie une valeur de retour. La `I` méthode est correcte, car son corps est équivalent à un bloc d'instructions avec une seule instruction return.

Surcharge de méthode

Les règles de résolution de surcharge de méthode sont décrites dans inférence de type.

Propriétés

Un ***Property** _ est un membre qui fournit l'accès à une caractéristique d'un objet ou d'une classe. Les exemples de propriétés incluent la longueur d'une chaîne, la taille d'une police, la légende d'une fenêtre, le nom d'un client, etc. Les propriétés sont une extension naturelle des champs, les deux étant des membres nommés avec des types associés, et la syntaxe d'accès aux champs et aux propriétés est la même. Toutefois,

contrairement aux champs, les propriétés ne désignent pas des emplacements de stockage. Au lieu de cela, les propriétés ont des _ *accesseurs** qui spécifient les instructions à exécuter lorsque leurs valeurs sont lues ou écrites. Les propriétés fournissent donc un mécanisme permettant d'associer des actions à la lecture et à l'écriture des attributs d'un objet. en outre, ils permettent de calculer ces attributs.

Les propriétés sont déclarées à l'aide de *property_declaration* s :

antlr

```
property_declaration
    : attributes? property_modifier* type member_name property_body
    ;

property_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'static'
    | 'virtual'
    | 'sealed'
    | 'override'
    | 'abstract'
    | 'extern'
    | property_modifier_unsafe
    ;

property_body
    : '{' accessor_declarations '}' property_initializer?
    | '=>' expression ';'
    ;

property_initializer
    : '=' variable_initializer ';'
    ;
```

Une *property_declaration* peut inclure un ensemble d' *attributs* (attributs) et une combinaison valide des quatre modificateurs d'accès (modificateurs d'accès), `new` (le nouveau modificateur), `static` (méthodes statiques et d'instance), `virtual` (méthodes virtuelles), (méthodes de `override` substitution), `sealed` (méthodes sealed), (`abstract` méthodes abstraites) et `extern` modificateurs (méthodes externes).

Les déclarations de propriété sont soumises aux mêmes règles que les déclarations de méthode (méthodes) en ce qui concerne les combinaisons valides de modificateurs.

Le *type* d'une déclaration de propriété spécifie le type de la propriété introduite par la déclaration, et le *Member_Name* spécifie le nom de la propriété. À moins que la propriété ne soit une implémentation de membre d'interface explicite, le *Member_Name* est simplement un *identificateur*. Pour une implémentation de membre d'interface explicite (implémentations de membres d'interface explicites), le *Member_Name* se compose d'un *INTERFACE_TYPE* suivi d'un « `.` » et d'un *identificateur*.

Le *type* d'une propriété doit être au moins aussi accessible que la propriété elle-même (contraintes d'accessibilité).

Un *property_body* peut se composer d'un ***corps d'accesseur*** _ *ou d'un _corps d'expression**.. Dans un corps d'accesseur, _accessor_declarations *, qui doit être placé entre des `{` jetons « » et « » `}` , déclarez les accesseurs (accesseurs) de la propriété. Les accesseurs spécifient les instructions exécutables associées à la lecture et à l'écriture de la propriété.

Un corps d'expression composé de `=>` suivis d'une *expression* `E` et d'un point-virgule est exactement équivalent au corps de l'instruction `{ get { return E; } }` et ne peut donc être utilisé que pour spécifier des propriétés Getter uniquement lorsque le résultat de l'accesseur Get est donné par une expression unique.

Une *property_initializer* peut uniquement être fournie pour une propriété implémentée automatiquement (Propriétés implémentées automatiquement) et provoque l'initialisation du champ sous-jacent de telles propriétés avec la valeur donnée par l' *expression*.

Même si la syntaxe d'accès à une propriété est identique à celle d'un champ, une propriété n'est pas classée comme une variable. Par conséquent, il n'est pas possible de passer une propriété en tant qu' `ref` `out` argument ou.

Lorsqu'une déclaration de propriété comprend un `extern` modificateur, la propriété est dite « ***propriété externe** _ ». Étant donné qu'une déclaration de propriété externe ne fournit aucune implémentation réelle, chacun de ses _accessor_declarations * se compose d'un point-virgule.

Propriétés statiques et d'instance

Lorsqu'une déclaration de propriété comprend un `static` modificateur, la propriété est dite « ***static Property** _ ». Si aucun `static` modificateur n'est présent, la propriété est considérée comme une propriété d'*instance* _ * * *.

Une propriété statique n'est pas associée à une instance spécifique, et il s'agit d'une erreur de compilation pour faire référence aux `this` accesseurs d'une propriété statique.

Une propriété d'instance est associée à une instance donnée d'une classe, et cette instance est accessible en tant que `this` (cet accès) dans les accesseurs de cette propriété.

Lorsqu'une propriété est référencée dans un *member_access* (accès aux membres) du formulaire `E.M` , si `M` est une propriété statique, `E` doit désigner un type contenant `M` , et si `M` est une propriété d'instance, E doit désigner une instance d'un type contenant `M` .

Les différences entre les membres statiques et les membres d'instance sont abordées plus en détail dans les membres statiques et d'instance.

Accesseurs

La *accessor_declarations* d'une propriété spécifie les instructions exécutables associées à la lecture et à l'écriture de cette propriété.

antlr

```
accessor_declarations
    : get_accessor_declaration set_accessor_declaration?
    | set_accessor_declaration get_accessor_declaration?
    ;

get_accessor_declaration
    : attributes? accessor_modifier? 'get' accessor_body
    ;

set_accessor_declaration
    : attributes? accessor_modifier? 'set' accessor_body
    ;

accessor_modifier
    : 'protected'
    | 'internal'
    | 'private'
    | 'protected' 'internal'
    | 'internal' 'protected'
    ;

accessor_body
    : block
    | ';'
    ;
```

Les déclarations d'accesseur se composent d'un *get_accessor_declaration*, d'un *set_accessor_declaration* ou des deux. Chaque déclaration d'accesseur se compose du jeton `get` ou `set` suivi d'un *accessor_modifier* facultatif et d'un *accessor_body*.

L'utilisation de *accessor_modifier* s est régie par les restrictions suivantes :

- Un *accessor_modifier* ne peut pas être utilisé dans une interface ou dans une implémentation de membre d'interface explicite.
- Pour une propriété ou un indexeur qui n'a pas de `override` modificateur, une *accessor_modifier* est autorisée uniquement si la propriété ou l'indexeur a à la fois un `get` `set` accesseur et, puis n'est autorisé que sur l'un de ces accesseurs.
- Pour une propriété ou un indexeur qui comprend un `override` modificateur, un accesseur doit correspondre à l' *accessor_modifier*, le cas échéant, de l'accesseur en cours de substitution.
- Le *accessor_modifier* doit déclarer une accessibilité qui est strictement plus restrictive que l'accessibilité déclarée de la propriété ou de l'indexeur lui-même. Pour être précis :
  - Si la propriété ou l'indexeur a une accessibilité déclarée de `public` , le *accessor_modifier* peut être `protected internal` , `internal` , `protected` ou `private` .
  - Si la propriété ou l'indexeur a une accessibilité déclarée de `protected internal` , le *accessor_modifier* peut être `internal` , `protected` ou `private` .
  - Si la propriété ou l'indexeur a une accessibilité déclarée de `internal` ou `protected` , le *accessor_modifier* doit être `private` .
  - Si la propriété ou l'indexeur a une accessibilité déclarée de `private` , aucune *accessor_modifier* ne peut être utilisée.

Pour `abstract` `extern` les propriétés et, le *accessor_body* pour chaque accesseur spécifié est simplement un point-virgule. Une propriété non abstraite, non-extern peut avoir chaque *accessor_body* être un point-virgule, auquel cas il s'agit d'une **propriété implémentée automatiquement** _ (Propriétés implémentées automatiquement). Une propriété implémentée automatiquement doit avoir au moins un accesseur Get. Pour les accesseurs de toute autre propriété non-abstract non-extern, le _accessor_body * est un *bloc* qui spécifie les instructions à exécuter lorsque l'accesseur correspondant est appelé.

Un `get` accesseur correspond à une méthode sans paramètre avec une valeur de retour du type de propriété. À l'exception de la cible d'une assignation, lorsqu'une propriété est référencée dans une expression, l' `get` accesseur de la propriété est appelé pour calculer la valeur de la propriété (valeurs des expressions). Le corps d'un `get` accesseur doit se conformer aux règles pour les méthodes retournant des valeurs décrites dans le corps de la méthode. En particulier, toutes les `return` instructions dans le corps d'un

`get` accesseur doivent spécifier une expression qui est implicitement convertible en type de propriété. En outre, le point de terminaison d'un `get` accesseur ne doit pas être accessible.

Un `set` accesseur correspond à une méthode avec un paramètre de valeur unique du type de propriété et un `void` type de retour. Le paramètre implicite d'un `set` accesseur est toujours nommé `value` . Lorsqu'une propriété est référencée en tant que cible d'une assignation (opérateurs d'affectation), ou en tant qu'opérande de `++` ou `--` (opérateurs suffixés d'incrémentation et de décrémentation, opérateurs de préfixe d'incrémentation et de décrémentation), l' `set` accesseur est appelé avec un argument (dont la valeur est celle du côté droit de l'assignation ou de l' `++` `--` opérateuror) qui fournit la nouvelle valeur Le corps d'un `set` accesseur doit se conformer aux règles pour les `void` méthodes décrites dans le corps de la méthode. En particulier, `return` les instructions dans le corps de l' `set` accesseur ne sont pas autorisées à spécifier une expression. Comme un `set` accesseur a implicitement un paramètre nommé `value` , il s'agit d'une erreur de compilation pour une variable locale ou une déclaration de constante dans un `set` accesseur portant ce nom.

En fonction de la présence ou de l'absence des `get` `set` accesseurs et, une propriété est classée comme suit :

- Une propriété qui comprend à la fois un `get` accesseur et un `set` accesseur est considérée comme une propriété ***en lecture-écriture*** .
- Une propriété qui a uniquement un `get` accesseur est considérée comme étant une propriété ***en lecture seule*** . Il s'agit d'une erreur de compilation pour qu'une propriété en lecture seule soit la cible d'une assignation.
- Une propriété qui a uniquement un `set` accesseur est considérée comme étant une propriété en ***écriture seule*** . À l'exception de la cible d'une assignation, il s'agit d'une erreur au moment de la compilation pour référencer une propriété en écriture seule dans une expression.

Dans l'exemple

C#

```
public class Button: Control
{
    private string caption;

    public string Caption {
        get {
            return caption;
        }
```

```
        set {
            if (caption != value) {
                caption = value;
                Repaint();
            }
        }
    }

    public override void Paint(Graphics g, Rectangle r) {
        // Painting code goes here
    }
}
```

le `Button` contrôle déclare une propriété publique `Caption` . L' `get` accesseur de la `Caption` propriété retourne la chaîne stockée dans le `caption` champ privé. L' `set` accesseur vérifie si la nouvelle valeur est différente de la valeur actuelle, et si c'est le cas, elle stocke la nouvelle valeur et repeint le contrôle. Les propriétés suivent souvent le modèle ci-dessus : l' `get` accesseur retourne simplement une valeur stockée dans un champ privé, et l' `set` accesseur modifie ce champ privé, puis exécute toutes les actions supplémentaires requises pour mettre à jour complètement l'état de l'objet.

À partir de la `Button` classe ci-dessus, voici un exemple d'utilisation de la `Caption` propriété :

C#

```
Button okButton = new Button();
okButton.Caption = "OK";            // Invokes set accessor
string s = okButton.Caption;        // Invokes get accessor
```

Ici, l' `set` accesseur est appelé en affectant une valeur à la propriété, et l' `get` accesseur est appelé en référençant la propriété dans une expression.

Les `get` `set` accesseurs et d'une propriété ne sont pas des membres distincts et il n'est pas possible de déclarer les accesseurs d'une propriété séparément. Par conséquent, il n'est pas possible que les deux accesseurs d'une propriété en lecture-écriture aient une accessibilité différente. L'exemple

C#

```
class A
{
    private string name;

    public string Name {                // Error, duplicate member name
        get { return name; }
    }
```

```
    public string Name {                // Error, duplicate member name
        set { name = value; }
    }
}
```

ne déclare pas une propriété en lecture-écriture unique. Au lieu de cela, elle déclare deux propriétés portant le même nom, une seule en lecture seule et une seule en écriture seule. Dans la mesure où deux membres déclarés dans la même classe ne peuvent pas avoir le même nom, l'exemple provoque une erreur au moment de la compilation.

Lorsqu'une classe dérivée déclare une propriété à l'aide du même nom qu'une propriété héritée, la propriété dérivée masque la propriété héritée en ce qui concerne la lecture et l'écriture. Dans l'exemple

C#

```
class A
{
    public int P {
        set {...}
    }
}

class B: A
{
    new public int P {
        get {...}
    }
}
```

la `P` propriété dans `B` masque la `P` propriété dans en `A` ce qui concerne la lecture et l'écriture. Par conséquent, dans les instructions

C#

```
B b = new B();
b.P = 1;          // Error, B.P is read-only
((A)b).P = 1;     // Ok, reference to A.P
```

l'assignation à `b.P` provoque le signalement d'une erreur au moment de la compilation, puisque la propriété en lecture seule `P` dans `B` masque la propriété en écriture seule `P` dans `A` . Notez, toutefois, qu'un cast peut être utilisé pour accéder à la `P` propriété Hidden.

Contrairement aux champs publics, les propriétés fournissent une séparation entre l'état interne d'un objet et son interface publique. Prenons cet exemple :

C#

```
class Label
{
    private int x, y;
    private string caption;

    public Label(int x, int y, string caption) {
        this.x = x;
        this.y = y;
        this.caption = caption;
    }

    public int X {
        get { return x; }
    }

    public int Y {
        get { return y; }
    }

    public Point Location {
        get { return new Point(x, y); }
    }

    public string Caption {
        get { return caption; }
    }
}
```

Ici, la `Label` classe utilise deux `int` champs, `x` et `y` , pour stocker son emplacement. L'emplacement est exposé publiquement en tant que et en tant que propriété `X` `Y` et en tant que `Location` propriété de type `Point` . Si, dans une future version de `Label` , il devient plus pratique de stocker l'emplacement en `Point` interne, la modification peut être effectuée sans affecter l'interface publique de la classe :

C#

```
class Label
{
    private Point location;
    private string caption;

    public Label(int x, int y, string caption) {
        this.location = new Point(x, y);
        this.caption = caption;
    }
```

```
    public int X {
        get { return location.x; }
    }

    public int Y {
        get { return location.y; }
    }

    public Point Location {
        get { return location; }
    }

    public string Caption {
        get { return caption; }
    }
}
```

Avait `x` et `y` `public readonly` à la place des champs, il aurait été impossible d'apporter une telle modification à la `Label` classe.

L'exposition de l'état via des propriétés n'est pas nécessairement moins efficace que l'exposition directe de champs. En particulier, lorsqu'une propriété est non virtuelle et ne contient qu'une petite quantité de code, l'environnement d'exécution peut remplacer les appels aux accesseurs par le code réel des accesseurs. Ce processus est appelé ***incorporation***, et il rend l'accès aux propriétés aussi efficace que l'accès au champ, tout en préservant la flexibilité accrue des propriétés.

Étant donné que `get` l'appel d'un accesseur est conceptuellement équivalent à la lecture de la valeur d'un champ, il est considéré comme un style de programmation incorrect pour les `get` accesseurs d'avoir des effets secondaires observables. Dans l'exemple

C#

```
class Counter
{
    private int next;

    public int Next {
        get { return next++; }
    }
}
```

la valeur de la `Next` propriété dépend du nombre de fois où l'accès à la propriété a été effectué précédemment. Ainsi, l'accès à la propriété produit un effet secondaire observable, et la propriété doit être implémentée comme une méthode à la place.

La Convention « sans effets secondaires » pour les `get` accesseurs ne signifie pas que les `get` accesseurs doivent toujours être écrits pour renvoyer simplement des valeurs stockées dans des champs. En effet, `get` les accesseurs calculent souvent la valeur d'une propriété en accédant à plusieurs champs ou en appelant des méthodes. Toutefois, un `get` accesseur correctement conçu n'effectue aucune action qui entraîne des modifications observables de l'état de l'objet.

Les propriétés peuvent être utilisées pour différer l'initialisation d'une ressource jusqu'au moment où elle est référencée pour la première fois. Par exemple :

C#

```
using System.IO;

public class Console
{
    private static TextReader reader;
    private static TextWriter writer;
    private static TextWriter error;

    public static TextReader In {
        get {
            if (reader == null) {
                reader = new StreamReader(Console.OpenStandardInput());
            }
            return reader;
        }
    }

    public static TextWriter Out {
        get {
            if (writer == null) {
                writer = new StreamWriter(Console.OpenStandardOutput());
            }
            return writer;
        }
    }

    public static TextWriter Error {
        get {
            if (error == null) {
                error = new StreamWriter(Console.OpenStandardError());
            }
            return error;
        }
    }
}
```

La `Console` classe contient trois propriétés, `In` , `Out` et `Error` , qui représentent respectivement les périphériques d'entrée, de sortie et d'erreur standard. En exposant

ces membres en tant que propriétés, la `Console` classe peut différer leur initialisation jusqu'à ce qu'elles soient réellement utilisées. Par exemple, lors de la première référence à la `Out` propriété, comme dans

C#

```
Console.Out.WriteLine("hello, world");
```

le sous-jacent `TextWriter` pour le périphérique de sortie est créé. Toutefois, si l'application ne fait pas référence `In` aux `Error` Propriétés et, aucun objet n'est créé pour ces appareils.

Propriétés implémentées automatiquement

Une propriété implémentée automatiquement (ou ***propriété automatique*** pour Short) est une propriété non-extern non abstraite avec des corps d'accesseur de point-virgule uniquement. Les propriétés automatiques doivent avoir un accesseur get et peuvent éventuellement avoir un accesseur Set.

Quand une propriété est spécifiée en tant que propriété implémentée automatiquement, un champ de stockage masqué est automatiquement disponible pour la propriété, et les accesseurs sont implémentés pour lire et écrire dans ce champ de stockage. Si la propriété automatique n'a pas d'accesseur Set, le champ de stockage est pris en compte `readonly` (champs en lecture seule). Tout comme un `readonly` champ, une propriété automatique Getter uniquement peut également être assignée à dans le corps d'un constructeur de la classe englobante. Une assignation de ce type est directement assignée au champ de stockage en lecture seule de la propriété.

Une propriété automatique peut éventuellement avoir une *property_initializer*, qui est appliquée directement au champ de stockage en tant que *variable_initializer* (initialiseurs de variable).

L'exemple suivant :

C#

```
public class Point {
    public int X { get; set; } = 0;
    public int Y { get; set; } = 0;
}
```

équivaut à la déclaration suivante :

C#

```
public class Point {
    private int __x = 0;
    private int __y = 0;
    public int X { get { return __x; } set { __x = value; } }
    public int Y { get { return __y; } set { __y = value; } }
}
```

L'exemple suivant :

C#

```
public class ReadOnlyPoint
{
    public int X { get; }
    public int Y { get; }
    public ReadOnlyPoint(int x, int y) { X = x; Y = y; }
}
```

équivaut à la déclaration suivante :

C#

```
public class ReadOnlyPoint
{
    private readonly int __x;
    private readonly int __y;
    public int X { get { return __x; } }
    public int Y { get { return __y; } }
    public ReadOnlyPoint(int x, int y) { __x = x; __y = y; }
}
```

Notez que les assignations au champ ReadOnly sont autorisées, car elles se produisent dans le constructeur.

Accessibilité

Si un accesseur a un *accessor_modifier*, le domaine d'accessibilité (domaines d'accessibilité) de l'accesseur est déterminé à l'aide de l'accessibilité déclarée du *accessor_modifier*. Si un accesseur n'a pas de *accessor_modifier*, le domaine d'accessibilité de l'accesseur est déterminé à partir de l'accessibilité déclarée de la propriété ou de l'indexeur.

La présence d'un *accessor_modifier* affecte jamais la recherche de membre (opérateurs) ou la résolution de surcharge (résolution de surcharge). Les modificateurs de la

propriété ou de l'indexeur déterminent toujours la propriété ou l'indexeur auquel il est lié, quel que soit le contexte de l'accès.

Une fois qu'une propriété ou un indexeur particulier a été sélectionné, les domaines d'accessibilité des accesseurs spécifiques impliqués sont utilisés pour déterminer si cette utilisation est valide :

- Si l'utilisation est en tant que valeur (valeurs d'expressions), l' `get` accesseur doit exister et être accessible.
- Si l'utilisation est la cible d'une assignation simple (assignation simple), l' `set` accesseur doit exister et être accessible.
- Si l'utilisation est la cible de l'assignation composée (assignation composée), ou en tant que cible des `++` `--` opérateurs ou (fonctions membres0,9, expressions d'appel), les `get` accesseurs et l' `set` accesseur doivent exister et être accessibles.

Dans l'exemple suivant, la propriété `A.Text` est masquée par la propriété `B.Text` , même dans les contextes où seul l' `set` accesseur est appelé. En revanche, la propriété `B.Count` n'est pas accessible à la classe `M` , donc la propriété accessible `A.Count` est utilisée à la place.

C#

```
class A
{
    public string Text {
        get { return "hello"; }
        set { }
    }

    public int Count {
        get { return 5; }
        set { }
    }
}

class B: A
{
    private string text = "goodbye";
    private int count = 0;

    new public string Text {
        get { return text; }
        protected set { text = value; }
    }

    new protected int Count {
        get { return count; }
        set { count = value; }
```

```
    }
}

class M
{
    static void Main() {
        B b = new B();
        b.Count = 12;              // Calls A.Count set accessor
        int i = b.Count;           // Calls A.Count get accessor
        b.Text = "howdy";          // Error, B.Text set accessor not
accessible
        string s = b.Text;         // Calls B.Text get accessor
    }
}
```

Un accesseur utilisé pour implémenter une interface peut ne pas avoir de *accessor_modifier*. Si un seul accesseur est utilisé pour implémenter une interface, l'autre accesseur peut être déclaré avec un *accessor_modifier*:

C#

```
public interface I
{
    string Prop { get; }
}

public class C: I
{
    public string Prop {
        get { return "April"; }       // Must not have a modifier here
        internal set {...}            // Ok, because I.Prop has no set
accessor
    }
}
```

Accesseurs de propriété Virtual, sealed, override et abstract

Une `virtual` déclaration de propriété spécifie que les accesseurs de la propriété sont virtuels. Le `virtual` modificateur s'applique aux deux accesseurs d'une propriété en lecture-écriture ; il n'est pas possible pour un seul accesseur d'une propriété en lecture-écriture d'être virtuel.

Une `abstract` déclaration de propriété spécifie que les accesseurs de la propriété sont virtuels, mais ne fournissent pas d'implémentation réelle des accesseurs. Au lieu de cela, les classes dérivées non abstraites sont requises pour fournir leur propre implémentation des accesseurs en substituant la propriété. Comme un accesseur pour

une déclaration de propriété abstraite ne fournit pas d'implémentation réelle, son *accessor_body* se compose simplement d'un point-virgule.

Une déclaration de propriété qui comprend à la fois les `abstract` `override` modificateurs et spécifie que la propriété est abstraite et substitue une propriété de base. Les accesseurs d'une telle propriété sont également abstraits.

Les déclarations de propriété abstraite sont autorisées uniquement dans les classes abstraites (classes abstraites). Les accesseurs d'une propriété virtuelle héritée peuvent être substitués dans une classe dérivée en incluant une déclaration de propriété qui spécifie une `override` directive. C'est ce qu'on appelle une ***déclaration de propriété de substitution***. Une déclaration de propriété de substitution ne déclare pas de nouvelle propriété. Au lieu de cela, il spécialise simplement les implémentations des accesseurs d'une propriété virtuelle existante.

Une déclaration de propriété de substitution doit spécifier exactement les mêmes modificateurs d'accessibilité, le même type et le même nom que la propriété héritée. Si la propriété héritée n'a qu'un seul accesseur (autrement dit, si la propriété héritée est en lecture seule ou en écriture seule), la propriété de substitution doit inclure uniquement cet accesseur. Si la propriété héritée inclut des accesseurs (c'est-à-dire que si la propriété héritée est en lecture-écriture), la propriété de substitution peut inclure un seul accesseur ou les deux accesseurs.

Une déclaration de propriété de substitution peut inclure le `sealed` modificateur. L'utilisation de ce modificateur empêche une classe dérivée de substituer davantage la propriété. Les accesseurs d'une propriété sealed sont également sealed.

À l'exception des différences dans la syntaxe de déclaration et d'appel, les accesseurs Virtual, sealed, override et abstract se comportent exactement comme des méthodes virtuelles, sealed, override et abstract. Plus précisément, les règles décrites dans les méthodes virtuelles, les méthodes override, les méthodes sealedet les méthodes abstraites s'appliquent comme si les accesseurs étaient des méthodes d'un formulaire correspondant :

- Un `get` accesseur correspond à une méthode sans paramètre avec une valeur de retour du type de propriété et les mêmes modificateurs que la propriété conteneur.
- Un `set` accesseur correspond à une méthode avec un paramètre de valeur unique du type de propriété, un `void` type de retour et les mêmes modificateurs que la propriété conteneur.

Dans l'exemple

C#

```
abstract class A
{
    int y;

    public virtual int X {
        get { return 0; }
    }

    public virtual int Y {
        get { return y; }
        set { y = value; }
    }

    public abstract int Z { get; set; }
}
```

`X` est une propriété en lecture seule virtuelle, `Y` est une propriété en lecture-écriture virtuelle et `Z` est une propriété en lecture-écriture abstraite. Étant donné que `Z` est abstrait, la classe conteneur `A` doit également être déclarée abstract.

Une classe qui dérive de `A` est affichée ci-dessous :

C#

```
class B: A
{
    int z;

    public override int X {
        get { return base.X + 1; }
    }

    public override int Y {
        set { base.Y = value < 0? 0: value; }
    }

    public override int Z {
        get { return z; }
        set { z = value; }
    }
}
```

Ici, les déclarations de `X` , `Y` et `Z` remplacent les déclarations de propriété. Chaque déclaration de propriété correspond exactement aux modificateurs d'accessibilité, au type et au nom de la propriété héritée correspondante. L' `get` accesseur de `X` et l' `set` accesseur de `Y` utilisent le `base` mot clé pour accéder aux accesseurs hérités. La déclaration de `Z` substitue les deux accesseurs abstraits. par conséquent, il n'existe

aucun membre de fonction abstraite en attente dans `B` , et il `B` est autorisé à être une classe non abstraite.

Quand une propriété est déclarée en tant que `override` , tout accesseur substitué doit être accessible au code de substitution. En outre, l'accessibilité déclarée de la propriété ou de l'indexeur lui-même, et des accesseurs, doit correspondre à celle du membre et des accesseurs substitués. Par exemple :

C#

```
public class B
{
    public virtual int P {
        protected set {...}
        get {...}
    }
}

public class D: B
{
    public override int P {
        protected set {...}           // Must specify protected here
        get {...}                     // Must not have a modifier here
    }
}
```

Événements

Un *__Event__ _ est un membre qui permet à un objet ou à une classe de fournir des notifications. Les clients peuvent joindre du code exécutable pour les événements en fournissant des *gestionnaires d'événements* _ * * *.

Les événements sont déclarés à l'aide de *event_declaration* s :

antlr

```
event_declaration
    : attributes? event_modifier* 'event' type variable_declarators ';'
    | attributes? event_modifier* 'event' type member_name '{'
event_accessor_declarations '}'
    ;

event_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
```

```
    | 'static'
    | 'virtual'
    | 'sealed'
    | 'override'
    | 'abstract'
    | 'extern'
    | event_modifier_unsafe
    ;

event_accessor_declarations
    : add_accessor_declaration remove_accessor_declaration
    | remove_accessor_declaration add_accessor_declaration
    ;

add_accessor_declaration
    : attributes? 'add' block
    ;

remove_accessor_declaration
    : attributes? 'remove' block
    ;
```

Une *event_declaration* peut inclure un ensemble d' *attributs* (attributs) et une combinaison valide des quatre modificateurs d'accès (modificateurs d'accès), `new` (le nouveau modificateur), `static` (méthodes statiques et d'instance), `virtual` (méthodes virtuelles), (méthodes de `override` substitution), `sealed` (méthodes sealed), (`abstract` méthodes abstraites) et `extern` modificateurs (méthodes externes).

Les déclarations d'événements sont soumises aux mêmes règles que les déclarations de méthode (méthodes) en ce qui concerne les combinaisons valides de modificateurs.

Le *type* d'une déclaration d'événement doit être un *delegate_type* (types référence), et ce *delegate_type* doit être au moins aussi accessible que l'événement lui-même (contraintes d'accessibilité).

Une déclaration d'événement peut inclure des *event_accessor_declarations*. Toutefois, si ce n'est pas le cas pour les événements non-extern et non abstraits, le compilateur les fournit automatiquement (événements de type champ); pour les événements extern, les accesseurs sont fournis en externe.

Déclaration d'événement qui omet *event_accessor_declarations* définit un ou plusieurs événements, un pour chacun des *variable_declarator* s. Les attributs et les modificateurs s'appliquent à tous les membres déclarés par ce *event_declaration*.

Il s'agit d'une erreur de compilation pour un *event_declaration* d'inclure à la fois le `abstract` modificateur et les *event_accessor_declarations* délimités par des accolades.

Lorsqu'une déclaration Event comprend un `extern` modificateur, l'événement est dit « *__External Event__ _ ». Étant donné qu'une déclaration d'événement externe ne fournit aucune implémentation réelle, c'est une erreur qu'elle inclut à la fois le `extern` modificateur et _event_accessor_declarations *.

Il s'agit d'une erreur de compilation pour une *variable_declarator* d'une déclaration d'événement avec un `abstract` `external` modificateur ou pour inclure une *variable_initializer*.

Un événement peut être utilisé comme opérande de gauche des `+=` `-=` opérateurs et (assignation d'événement). Ces opérateurs sont utilisés, respectivement, pour attacher des gestionnaires d'événements à ou pour supprimer des gestionnaires d'événements d'un événement, et les modificateurs d'accès de l'événement contrôlent les contextes dans lesquels ces opérations sont autorisées.

Étant donné que `+=` et `-=` sont les seules opérations autorisées sur un événement à l'extérieur du type qui déclare l'événement, le code externe peut ajouter et supprimer des gestionnaires pour un événement, mais il ne peut pas non plus obtenir ni modifier la liste sous-jacente des gestionnaires d'événements.

Dans une opération de la forme `x += y` ou `x -= y` , quand `x` est un événement et que la référence a lieu en dehors du type qui contient la déclaration de `x` , le résultat de l'opération a le type `void` (par opposition au type de `x` , avec la valeur de `x` après l'assignation). Cette règle interdit au code externe d'examiner indirectement le délégué sous-jacent d'un événement.

L'exemple suivant montre comment les gestionnaires d'événements sont attachés aux instances de la `Button` classe :

C#

```
public delegate void EventHandler(object sender, EventArgs e);

public class Button: Control
{
    public event EventHandler Click;
}

public class LoginDialog: Form
{
    Button OkButton;
    Button CancelButton;

    public LoginDialog() {
        OkButton = new Button(...);
        OkButton.Click += new EventHandler(OkButtonClick);
```

```
        CancelButton = new Button(...);
        CancelButton.Click += new EventHandler(CancelButtonClick);
    }

    void OkButtonClick(object sender, EventArgs e) {
        // Handle OkButton.Click event
    }

    void CancelButtonClick(object sender, EventArgs e) {
        // Handle CancelButton.Click event
    }
}
```

Ici, le `LoginDialog` constructeur d'instance crée deux `Button` instances et attache des gestionnaires d'événements aux `Click` événements.

Événements de type champ

Dans le texte de programme de la classe ou du struct qui contient la déclaration d'un événement, certains événements peuvent être utilisés comme des champs. Pour être utilisé de cette manière, un événement ne doit pas être `abstract` ou `extern` , et ne doit pas inclure explicitement *event_accessor_declarations*. Un tel événement peut être utilisé dans n'importe quel contexte qui autorise un champ. Le champ contient un délégué (délégués) qui fait référence à la liste des gestionnaires d'événements qui ont été ajoutés à l'événement. Si aucun gestionnaire d'événements n'a été ajouté, le champ contient `null` .

Dans l'exemple

C#

```
public delegate void EventHandler(object sender, EventArgs e);

public class Button: Control
{
    public event EventHandler Click;

    protected void OnClick(EventArgs e) {
        if (Click != null) Click(this, e);
    }

    public void Reset() {
        Click = null;
    }
}
```

`Click` est utilisé en tant que champ dans la `Button` classe. Comme le montre l'exemple, le champ peut être examiné, modifié et utilisé dans les expressions d'appel de délégué. La `OnClick` méthode de la `Button` classe « déclenche » l' `Click` événement. La notion de déclenchement d'un événement est équivalente à l'appel de délégué représenté par l'événement. Par conséquent, il n'existe aucune construction de langage particulière pour déclencher des événements. Notez que l'appel de délégué est précédé d'une vérification qui garantit que le délégué est non null.

En dehors de la déclaration de la `Button` classe, le `Click` membre peut uniquement être utilisé sur le côté gauche des `+=` opérateurs et `-=` , comme dans

C#

```
b.Click += new EventHandler(...);
```

qui ajoute un délégué à la liste d'appel de l' `Click` événement, et

C#

```
b.Click -= new EventHandler(...);
```

qui supprime un délégué de la liste d'appel de l' `Click` événement.

Lors de la compilation d'un événement de type champ, le compilateur crée automatiquement le stockage pour contenir le délégué et crée des accesseurs pour l'événement qui ajoute ou supprime des gestionnaires d'événements dans le champ délégué. Les opérations d'ajout et de suppression sont thread-safe et peuvent (mais n'ont pas besoin d'être) être effectuées tout en détenant le verrou (l'instruction lock) sur l'objet conteneur d'un événement d'instance, ou l'objet type (expressions de création d'objet anonyme) pour un événement statique.

Ainsi, une déclaration d'événement d'instance de la forme :

C#

```
class X
{
    public event D Ev;
}
```

sera compilé en une valeur équivalente à :

C#

```
class X
{
    private D __Ev;  // field to hold the delegate

    public event D Ev {
        add {
            /* add the delegate in a thread safe way */
        }

        remove {
            /* remove the delegate in a thread safe way */
        }
    }
}
```

Dans la classe `X` , les références à `Ev` sur le côté gauche des `+=` `-=` opérateurs et entraînent l'appel des accesseurs Add et Remove. Toutes les autres références à `Ev` sont compilées pour référencer le champ masqué à la `__Ev` place (accès aux membres). Le nom « `__Ev` » est arbitraire ; le champ masqué peut avoir n'importe quel nom ou aucun nom.

Accesseurs d'événement

Les déclarations d'événements omettent généralement *event_accessor_declarations*, comme dans l' `Button` exemple ci-dessus. Une situation pour cela implique le cas où le coût de stockage d'un champ par événement n'est pas acceptable. Dans ce cas, une classe peut inclure des *event_accessor_declarations* et utiliser un mécanisme privé pour stocker la liste des gestionnaires d'événements.

La *event_accessor_declarations* d'un événement spécifie les instructions exécutables associées à l'ajout et à la suppression de gestionnaires d'événements.

Les déclarations d'accesseur se composent d'un *add_accessor_declaration* et d'un *remove_accessor_declaration*. Chaque déclaration d'accesseur se compose du jeton `add` ou `remove` suivi d'un *bloc*. Le *bloc* associé à un *add_accessor_declaration* spécifie les instructions à exécuter lorsqu'un gestionnaire d'événements est ajouté, et le *bloc* associé à un *remove_accessor_declaration* spécifie les instructions à exécuter lorsqu'un gestionnaire d'événements est supprimé.

Chaque *add_accessor_declaration* et *remove_accessor_declaration* correspond à une méthode avec un paramètre de valeur unique du type d'événement et un `void` type de retour. Le paramètre implicite d'un accesseur d'événement est nommé `value` . Lorsqu'un événement est utilisé dans une assignation d'événement, l'accesseur

d'événement approprié est utilisé. Plus précisément, si l'opérateur d'assignation est, `+=` l'accesseur Add est utilisé, et si l'opérateur d'assignation est, `-=` l'accesseur remove est utilisé. Dans les deux cas, l'opérande droit de l'opérateur d'assignation est utilisé comme argument de l'accesseur d'événement. Le bloc d'un *add_accessor_declaration* ou d'un *remove_accessor_declaration* doit respecter les règles applicables aux `void` méthodes décrites dans corps de la méthode. En particulier, `return` les instructions d'un tel bloc ne sont pas autorisées à spécifier une expression.

Comme un accesseur d'événement a implicitement un paramètre nommé `value` , il s'agit d'une erreur de compilation pour une variable locale ou une constante déclarée dans un accesseur d'événement pour avoir ce nom.

Dans l'exemple

C#

```
class Control: Component
{
    // Unique keys for events
    static readonly object mouseDownEventKey = new object();
    static readonly object mouseUpEventKey = new object();

    // Return event handler associated with key
    protected Delegate GetEventHandler(object key) {...}

    // Add event handler associated with key
    protected void AddEventHandler(object key, Delegate handler) {...}

    // Remove event handler associated with key
    protected void RemoveEventHandler(object key, Delegate handler) {...}

    // MouseDown event
    public event MouseEventHandler MouseDown {
        add { AddEventHandler(mouseDownEventKey, value); }
        remove { RemoveEventHandler(mouseDownEventKey, value); }
    }

    // MouseUp event
    public event MouseEventHandler MouseUp {
        add { AddEventHandler(mouseUpEventKey, value); }
        remove { RemoveEventHandler(mouseUpEventKey, value); }
    }

    // Invoke the MouseUp event
    protected void OnMouseUp(MouseEventArgs args) {
        MouseEventHandler handler;
        handler = (MouseEventHandler)GetEventHandler(mouseUpEventKey);
        if (handler != null)
            handler(this, args);
```

```
    }
}
```

la `Control` classe implémente un mécanisme de stockage interne pour les événements. La `AddEventHandler` méthode associe une valeur de délégué à une clé, la `GetEventHandler` méthode retourne le délégué actuellement associé à une clé, et la `RemoveEventHandler` méthode supprime un délégué en tant que gestionnaire d'événements pour l'événement spécifié. Vraisemblablement, le mécanisme de stockage sous-jacent est conçu de telle sorte qu'il n'y a aucun coût d'association d'une `null` valeur de délégué à une clé, et par conséquent, les événements non gérés ne consomment aucun stockage.

Événements statiques et d'instance

Lorsqu'une déclaration Event comprend un `static` modificateur, l'événement est considéré comme un ***static Event** _. Si aucun `static` modificateur n'est présent, l'événement est considéré comme un _ *événement d'instance* *.

Un événement statique n'est pas associé à une instance spécifique, et il s'agit d'une erreur de compilation pour faire référence aux `this` accesseurs d'un événement statique.

Un événement d'instance est associé à une instance donnée d'une classe, et cette instance est accessible en tant que `this` (cet accès) dans les accesseurs de cet événement.

Lorsqu'un événement est référencé dans un *member_access* (accès aux membres) du formulaire `E.M` , si `M` est un événement statique, `E` doit désigner un type contenant `M` , et si est un `M` événement d'instance, E doit désigner une instance d'un type contenant `M` .

Les différences entre les membres statiques et les membres d'instance sont abordées plus en détail dans les membres statiques et d'instance.

Accesseurs d'événement virtuels, sealed, override et abstract

Une `virtual` déclaration d'événement spécifie que les accesseurs de cet événement sont virtuels. Le `virtual` modificateur s'applique aux deux accesseurs d'un événement.

Une `abstract` déclaration d'événement spécifie que les accesseurs de l'événement sont virtuels, mais ne fournissent pas d'implémentation réelle des accesseurs. Au lieu de cela, les classes dérivées non abstraites sont requises pour fournir leur propre implémentation des accesseurs en substituant l'événement. Étant donné qu'une déclaration d'événement abstraite ne fournit pas d'implémentation réelle, elle ne peut pas fournir d' *event_accessor_declarations* délimités par des accolades.

Une déclaration d'événement qui comprend à la fois les `abstract` `override` modificateurs et spécifie que l'événement est abstrait et substitue un événement de base. Les accesseurs d'un tel événement sont également abstraits.

Les déclarations d'événements abstract sont autorisées uniquement dans les classes abstraites (classes abstraites).

Les accesseurs d'un événement virtuel hérité peuvent être substitués dans une classe dérivée en incluant une déclaration d'événement qui spécifie un `override` modificateur. C'est ce qu'on appelle une ***déclaration d'événement de substitution***. Une déclaration d'événement de substitution ne déclare pas un nouvel événement. Au lieu de cela, il spécialise simplement les implémentations des accesseurs d'un événement virtuel existant.

Une déclaration d'événement de substitution doit spécifier exactement les mêmes modificateurs d'accessibilité, le même type et le même nom que l'événement substitué.

Une déclaration d'événement de substitution peut inclure le `sealed` modificateur. L'utilisation de ce modificateur empêche une classe dérivée de substituer davantage l'événement. Les accesseurs d'un événement sealed sont également sealed.

Il s'agit d'une erreur au moment de la compilation pour qu'une déclaration d'événement de substitution inclue un `new` modificateur.

À l'exception des différences dans la syntaxe de déclaration et d'appel, les accesseurs Virtual, sealed, override et abstract se comportent exactement comme des méthodes virtuelles, sealed, override et abstract. Plus précisément, les règles décrites dans les méthodes virtuelles, les méthodes override, les méthodes sealedet les méthodes abstraites s'appliquent comme si les accesseurs étaient des méthodes d'un formulaire correspondant. Chaque accesseur correspond à une méthode avec un paramètre de valeur unique du type d'événement, un `void` type de retour et les mêmes modificateurs que l'événement conteneur.

Indexeurs

Un **indexeur** _ est un membre qui permet l'indexation d'un objet de la même façon qu'un tableau. Les indexeurs sont déclarés à l'aide de _indexer_declaration * s :

antlr

```
indexer_declaration
    : attributes? indexer_modifier* indexer_declarator indexer_body
    ;

indexer_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'virtual'
    | 'sealed'
    | 'override'
    | 'abstract'
    | 'extern'
    | indexer_modifier_unsafe
    ;

indexer_declarator
    : type 'this' '[' formal_parameter_list ']'
    | type interface_type '.' 'this' '[' formal_parameter_list ']'
    ;

indexer_body
    : '{' accessor_declarations '}'
    | '=>' expression ';'
    ;
```

Une *indexer_declaration* peut inclure un ensemble d' *attributs* (attributs) et une combinaison valide des quatre modificateurs d'accès (modificateurs d'accès), `new` (le nouveau modificateur), `virtual` (méthodes virtuelles), (méthodes de `override` substitution), `sealed` (méthodes sealed), `abstract` (méthodes abstraites) et `extern` modificateurs (méthodes externes).

Les déclarations de l'indexeur sont soumises aux mêmes règles que les déclarations de méthode (méthodes) en ce qui concerne les combinaisons valides de modificateurs, à la seule exception que le modificateur static n'est pas autorisé sur une déclaration d'indexeur.

Les modificateurs `virtual` , `override` et s'excluent `abstract` mutuellement, sauf dans un cas. Les `abstract` `override` modificateurs et peuvent être utilisés ensemble afin qu'un indexeur abstrait puisse remplacer un indexeur virtuel.

Le *type* d'une déclaration d'indexeur spécifie le type d'élément de l'indexeur introduit par la déclaration. À moins que l'indexeur ne soit une implémentation de membre d'interface explicite, le *type* est suivi du mot clé `this` . Pour une implémentation de membre d'interface explicite, le *type* est suivi d'un *INTERFACE_TYPE*, d'un « `.` » et du mot clé `this` . Contrairement aux autres membres, les indexeurs n'ont pas de noms définis par l'utilisateur.

L' *formal_parameter_list* spécifie les paramètres de l'indexeur. La liste de paramètres formels d'un indexeur correspond à celle d'une méthode (paramètres de méthode), à ceci près qu'au moins un paramètre doit être spécifié et que les `ref` modificateurs de paramètres et ne `out` sont pas autorisés.

Le *type* d'un indexeur et chacun des types référencés dans le *formal_parameter_list* doivent être au moins aussi accessibles que l'indexeur lui-même (contraintes d'accessibilité).

Une *indexer_body* peut se composer d'un ***corps d'accesseur*** _ *ou d'un _corps d'expression*.. Dans un corps d'accesseur, _accessor_declarations *, qui doit être placé entre des `{` jetons « » et « » `}` , déclarez les accesseurs (accesseurs) de la propriété. Les accesseurs spécifient les instructions exécutables associées à la lecture et à l'écriture de la propriété.

Un corps d'expression composé de « `=>` » suivi d'une expression `E` et d'un point-virgule est exactement équivalent au corps de l'instruction `{ get { return E; } }` , et ne peut donc être utilisé que pour spécifier des indexeurs Getter uniquement où le résultat de l'accesseur Get est donné par une expression unique.

Même si la syntaxe d'accès à un élément d'indexeur est identique à celle d'un élément de tableau, un élément d'indexeur n'est pas classifié comme une variable. Par conséquent, il n'est pas possible de passer un élément d'indexeur en tant qu' `ref` `out` argument ou.

La liste de paramètres formels d'un indexeur définit la signature (signatures et surcharge) de l'indexeur. Plus précisément, la signature d'un indexeur se compose du nombre et des types de ses paramètres formels. Le type d'élément et les noms des paramètres formels ne font pas partie de la signature d'un indexeur.

La signature d'un indexeur doit être différente des signatures de tous les autres indexeurs déclarés dans la même classe.

Les indexeurs et les propriétés sont très similaires, mais diffèrent des façons suivantes :

- Une propriété est identifiée par son nom, tandis qu'un indexeur est identifié par sa signature.
- Une propriété est accessible par le biais d'une *simple_name* (noms simples) ou d'un *member_access* (accès aux membres), tandis qu'un élément d'indexeur est accessible par le biais d'un *element_access* (accès à l'indexeur).
- Une propriété peut être un `static` membre, alors qu'un indexeur est toujours un membre d'instance.
- Un `get` accesseur d'une propriété correspond à une méthode sans paramètres, alors qu'un `get` accesseur d'un indexeur correspond à une méthode avec la même liste de paramètres formels que l'indexeur.
- Un `set` accesseur d'une propriété correspond à une méthode avec un paramètre unique nommé `value` , alors qu'un `set` accesseur d'un indexeur correspond à une méthode avec la même liste de paramètres formels que l'indexeur, plus un paramètre supplémentaire nommé `value` .
- Il s'agit d'une erreur de compilation permettant à un accesseur d'indexeur de déclarer une variable locale portant le même nom qu'un paramètre d'indexeur.
- Dans une déclaration de propriété de substitution, la propriété héritée est accessible à l'aide de la syntaxe `base.P` , où `P` est le nom de la propriété. Dans une déclaration d'indexeur de substitution, l'indexeur hérité est accessible à l'aide de la syntaxe `base[E]` , où `E` est une liste d'expressions séparées par des virgules.
- Il n'existe aucun concept d'indexeur implémenté automatiquement. Il s'agit d'une erreur d'avoir un indexeur non abstrait non externe avec des accesseurs de point-virgule.

Hormis ces différences, toutes les règles définies dans les accesseurs et les Propriétés implémentées automatiquement s'appliquent aux accesseurs d'indexeur ainsi qu'aux accesseurs de propriété.

Lorsqu'une déclaration d'indexeur comprend un `extern` modificateur, l'indexeur est considéré comme un ***indexeur externe** _. Étant donné qu'une déclaration d'indexeur externe ne fournit aucune implémentation réelle, chacun de ses _accessor_declarations * se compose d'un point-virgule.

L'exemple ci-dessous déclare une `BitArray` classe qui implémente un indexeur pour accéder aux bits individuels dans le tableau de bits.

C#

```
using System;

class BitArray
{
    int[] bits;
```

```
    int length;

    public BitArray(int length) {
        if (length < 0) throw new ArgumentException();
        bits = new int[((length - 1) >> 5) + 1];
        this.length = length;
    }

    public int Length {
        get { return length; }
    }

    public bool this[int index] {
        get {
            if (index < 0 || index >= length) {
                throw new IndexOutOfRangeException();
            }
            return (bits[index >> 5] & 1 << index) != 0;
        }
        set {
            if (index < 0 || index >= length) {
                throw new IndexOutOfRangeException();
            }
            if (value) {
                bits[index >> 5] |= 1 << index;
            }
            else {
                bits[index >> 5] &= ~(1 << index);
            }
        }
    }
}
```

Une instance de la `BitArray` classe consomme sensiblement moins de mémoire qu'une correspondante `bool[]` (puisque chaque valeur du premier n'occupe qu'un seul bit au lieu du dernier octet), mais elle autorise les mêmes opérations qu'un `bool[]` .

La `CountPrimes` classe suivante utilise un `BitArray` et l'algorithme « crible » classique pour calculer le nombre de premiers compris entre 1 et un maximum donné :

C#

```
class CountPrimes
{
    static int Count(int max) {
        BitArray flags = new BitArray(max + 1);
        int count = 1;
        for (int i = 2; i <= max; i++) {
            if (!flags[i]) {
                for (int j = i * 2; j <= max; j += i) flags[j] = true;
                count++;
            }
```

```
        }
        return count;
    }

    static void Main(string[] args) {
        int max = int.Parse(args[0]);
        int count = Count(max);
        Console.WriteLine("Found {0} primes between 1 and {1}", count, max);
    }
}
```

Notez que la syntaxe pour accéder aux éléments de `BitArray` est exactement la même que pour un `bool[]` .

L'exemple suivant montre une classe de grille 26 * 10 qui a un indexeur avec deux paramètres. Le premier paramètre doit être une lettre majuscule ou minuscule dans la plage A-Z, tandis que le second doit être un entier compris dans la plage 0-9.

C#

```
using System;

class Grid
{
    const int NumRows = 26;
    const int NumCols = 10;

    int[,] cells = new int[NumRows, NumCols];

    public int this[char c, int col] {
        get {
            c = Char.ToUpper(c);
            if (c < 'A' || c > 'Z') {
                throw new ArgumentException();
            }
            if (col < 0 || col >= NumCols) {
                throw new IndexOutOfRangeException();
            }
            return cells[c - 'A', col];
        }

        set {
            c = Char.ToUpper(c);
            if (c < 'A' || c > 'Z') {
                throw new ArgumentException();
            }
            if (col < 0 || col >= NumCols) {
                throw new IndexOutOfRangeException();
            }
            cells[c - 'A', col] = value;
        }
```

```
    }
}
```

Surcharge de l'indexeur

Les règles de résolution de surcharge de l'indexeur sont décrites dans inférence de type.

Opérateurs

Un **opérateur*** _ est un membre qui définit la signification d'un opérateur d'expression qui peut être appliqué aux instances de la classe. Les opérateurs sont déclarés à l'aide de _operator_declaration * s :

antlr

```
operator_declaration
    : attributes? operator_modifier+ operator_declarator operator_body
    ;

operator_modifier
    : 'public'
    | 'static'
    | 'extern'
    | operator_modifier_unsafe
    ;

operator_declarator
    : unary_operator_declarator
    | binary_operator_declarator
    | conversion_operator_declarator
    ;

unary_operator_declarator
    : type 'operator' overloadable_unary_operator '(' type identifier ')'
    ;

overloadable_unary_operator
    : '+' | '-' | '!' | '~' | '++' | '--' | 'true' | 'false'
    ;

binary_operator_declarator
    : type 'operator' overloadable_binary_operator '(' type identifier ','
type identifier ')'
    ;

overloadable_binary_operator
    : '+'   | '-'   | '*'   | '/'   | '%'   | '&'   | '|'   | '^'   | '<<'
    | right_shift | '=='  | '!='  | '>'   | '<'   | '>='  | '<='
    ;
```

```
conversion_operator_declarator
    : 'implicit' 'operator' type '(' type identifier ')'
    | 'explicit' 'operator' type '(' type identifier ')'
    ;

operator_body
    : block
    | '=>' expression ';'
    | ';'
    ;
```

Il existe trois catégories d'opérateurs surchargeables : les opérateurs unaires (opérateurs unaires), les opérateurs binaires (opérateurs binaires) et les opérateurs de conversion (opérateurs de conversion).

Le *operator_body* est un point-virgule, un ***corps d'instruction*** _ *ou un corps d'* _*expression*_*.. Un corps d'instruction se compose d'un _block *, qui spécifie les instructions à exécuter lorsque l'opérateur est appelé. Le *bloc* doit se conformer aux règles pour les méthodes qui retournent des valeurs décrites dans le corps de la méthode. Un corps d'expression se compose de `=>` suivis d'une expression et d'un point-virgule, et désigne une expression unique à exécuter lorsque l'opérateur est appelé.

Pour `extern` les opérateurs, le *operator_body* se compose simplement d'un point-virgule. Pour tous les autres opérateurs, le *operator_body* est soit un corps de bloc, soit un corps d'expression.

Les règles suivantes s'appliquent à toutes les déclarations d'opérateur :

- Une déclaration d'opérateur doit inclure à la fois un `public` et un `static` modificateur.
- Le (s) paramètre (s) d'un opérateur doivent être des paramètres de valeur (paramètres de valeur). Il s'agit d'une erreur de compilation pour qu'une déclaration d'opérateur spécifie des `ref` `out` paramètres ou.
- La signature d'un opérateur (opérateurs unaires, opérateurs binaires, opérateurs de conversion) doit être différente des signatures de tous les autres opérateurs déclarés dans la même classe.
- Tous les types référencés dans une déclaration d'opérateur doivent être au moins aussi accessibles que l'opérateur lui-même (contraintes d'accessibilité).
- Il s'agit d'une erreur pour que le même modificateur apparaisse plusieurs fois dans une déclaration d'opérateur.

Chaque catégorie d'opérateur impose des restrictions supplémentaires, comme décrit dans les sections suivantes.

Comme les autres membres, les opérateurs déclarés dans une classe de base sont hérités par les classes dérivées. Étant donné que les déclarations d'opérateur requièrent toujours la classe ou le struct dans lequel l'opérateur est déclaré pour participer à la signature de l'opérateur, il n'est pas possible pour un opérateur déclaré dans une classe dérivée de masquer un opérateur déclaré dans une classe de base. Ainsi, le `new` modificateur n'est jamais requis et, par conséquent, n'est jamais autorisé dans une déclaration d'opérateur.

Vous trouverez des informations supplémentaires sur les opérateurs unaires et binaires dans opérateurs.

Vous trouverez des informations supplémentaires sur les opérateurs de conversion dans les conversions définies par l'utilisateur.

Opérateurs unaires

Les règles suivantes s'appliquent aux déclarations d'opérateur unaires, où `T` désigne le type d'instance de la classe ou du struct qui contient la déclaration d'opérateur :

- Un opérateur unaire `+` ,, `-` `!` ou `~` doit accepter un seul paramètre de type `T` ou `T?` et peut retourner n'importe quel type.
- Un `++` opérateur or unaire `--` doit accepter un seul paramètre de type `T` ou `T?` et doit retourner ce même type ou un type dérivé de celui-ci.
- Un `true` opérateur or unaire `false` doit accepter un seul paramètre de type `T` ou `T?` et doit retourner le type `bool` .

La signature d'un opérateur unaire se compose du jeton d'opérateur (`+` , `-` , `!` ,, `~` `++` , `--` , `true` ou `false`) et du type du paramètre formel unique. Le type de retour ne fait pas partie de la signature d'un opérateur unaire, ni du nom du paramètre formel.

Les `true` `false` opérateurs unaires et requièrent une déclaration de couple. Une erreur de compilation se produit si une classe déclare l'un de ces opérateurs sans également déclarer l'autre. Les `true` `false` opérateurs et sont décrits plus en détail dans les opérateurs logiques conditionnels définis par l'utilisateur et les expressions booléennes.

L'exemple suivant illustre une implémentation et une utilisation ultérieure de `operator ++` pour une classe Vector entière :

C#

```
public class IntVector
{
    public IntVector(int length) {...}
```

```
    public int Length {...}                 // read-only property

    public int this[int index] {...}        // read-write indexer

    public static IntVector operator ++(IntVector iv) {
        IntVector temp = new IntVector(iv.Length);
        for (int i = 0; i < iv.Length; i++)
            temp[i] = iv[i] + 1;
        return temp;
    }
}

class Test
{
    static void Main() {
        IntVector iv1 = new IntVector(4);    // vector of 4 x 0
        IntVector iv2;

        iv2 = iv1++;    // iv2 contains 4 x 0, iv1 contains 4 x 1
        iv2 = ++iv1;    // iv2 contains 4 x 2, iv1 contains 4 x 2
    }
}
```

Notez que la méthode Operator retourne la valeur produite par l'ajout de 1 à l'opérande, tout comme les opérateurs d'incrémentation et de décrémentation suffixés (opérateurs d'incrémentation et de décrémentation suffixés) et les opérateurs d'incrémentation et de décrémentation de préfixe (opérateurs de préfixe d'incrémentation et de décrémentation). Contrairement à C++, cette méthode n'a pas besoin de modifier directement la valeur de son opérande. En fait, la modification de la valeur de l'opérande violerait la sémantique standard de l'opérateur d'incrémentation suffixée.

Opérateurs binaires

Les règles suivantes s'appliquent aux déclarations d'opérateur binaire, où `T` dénote le type d'instance de la classe ou du struct qui contient la déclaration d'opérateur :

- Un opérateur binaire sans décalage doit accepter deux paramètres, au moins un qui doit avoir le type `T` ou `T?` , et peut retourner tout type.
- Un binaire `<<` ou un `>>` opérateur doit accepter deux paramètres, le premier qui doit avoir le type `T` ou `T?` et le second, le type `int` ou `int?` , et peut retourner n'importe quel type.

La signature d'un opérateur binaire se compose du jeton d'opérateur (`+` , `-` ,,, `*` `/` `%` ,,,,, `&` `|` `^` `<<` `>>` , `==` , `!=` , `>` , `<` , `>=` ou `<=`) et des types des deux paramètres

formels. Le type de retour et les noms des paramètres formels ne font pas partie de la signature d'un opérateur binaire.

Certains opérateurs binaires requièrent une déclaration de couple. Pour chaque déclaration de l'un des opérateurs d'une paire, il doit y avoir une déclaration correspondante de l'autre opérateur de la paire. Deux déclarations d'opérateur correspondent lorsqu'elles ont le même type de retour et le même type pour chaque paramètre. Les opérateurs suivants requièrent une déclaration de paires :

- `operator ==` et `operator !=`
- `operator >` et `operator <`
- `operator >=` et `operator <=`

Opérateurs de conversion

Une déclaration d'opérateur de conversion introduit une ***conversion définie par l'utilisateur*** (conversions définies par l'utilisateur) qui augmente les conversions implicites et explicites prédéfinies.

Une déclaration d'opérateur de conversion qui inclut le `implicit` mot clé introduit une conversion implicite définie par l'utilisateur. Les conversions implicites peuvent se produire dans diverses situations, y compris les appels de membres de fonction, les expressions de Cast et les assignations. Cela est décrit plus en détail dans conversions implicites.

Une déclaration d'opérateur de conversion qui inclut le `explicit` mot clé introduit une conversion explicite définie par l'utilisateur. Les conversions explicites peuvent se produire dans les expressions de Cast et sont décrites plus en détail dans les conversions explicites.

Un opérateur de conversion convertit un type source, indiqué par le type de paramètre de l'opérateur de conversion, en un type cible, indiqué par le type de retour de l'opérateur de conversion.

Pour un type de source `S` et un type de cible donnés `T` , si `S` ou `T` sont des types Nullable, Let `S0` et `T0` font référence à leurs types sous-jacents, sinon `S0` et `T0` sont égaux à `S` et `T` respectivement. Une classe ou un struct est autorisé à déclarer une conversion d'un type source `S` en type cible `T` uniquement si toutes les conditions suivantes sont vraies :

- `S0` et `T0` sont des types différents.

- `S0`Ou `T0` est le type de classe ou de struct dans lequel la déclaration d'opérateur a lieu.
- Ni `S0` ni `T0` n'est un *INTERFACE_TYPE*.
- En excluant les conversions définies par l'utilisateur, il n'existe pas de conversion de `S` vers `T` ou de à `T` `S` .

Dans le cadre de ces règles, tous les paramètres de type associés à `S` ou `T` sont considérés comme des types uniques qui n'ont aucune relation d'héritage avec d'autres types, et toutes les contraintes sur ces paramètres de type sont ignorées.

Dans l'exemple

```
C#
```

```
class C<T> {...}

class D<T>: C<T>
{
    public static implicit operator C<int>(D<T> value) {...}      // Ok
    public static implicit operator C<string>(D<T> value) {...}   // Ok
    public static implicit operator C<T>(D<T> value) {...}        // Error
}
```

les deux premières déclarations d'opérateur sont autorisées, car, à des fins d' indexeurs. 3 `T` et `int` et, respectivement, `string` sont considérés comme des types uniques sans relation. Toutefois, le troisième opérateur est une erreur, car `C<T>` est la classe de base de `D<T>` .

À partir de la deuxième règle, il suit qu'un opérateur de conversion doit effectuer une conversion vers ou à partir du type de classe ou de struct dans lequel l'opérateur est déclaré. Par exemple, il est possible pour un type de classe ou de struct de `C` définir une conversion de `C` vers `int` et de à `int` `C` , mais pas de à `int` `bool` .

Il n'est pas possible de redéfinir directement une conversion prédéfinie. Ainsi, les opérateurs de conversion ne sont pas autorisés à effectuer une conversion depuis ou vers `object` , car les conversions implicites et explicites existent déjà entre `object` et tous les autres types. De même, ni la source ni les types cibles d'une conversion ne peuvent être un type de base de l'autre, car une conversion existera déjà.

Toutefois, il est possible de déclarer des opérateurs sur des types génériques qui, pour des arguments de type particuliers, spécifient des conversions qui existent déjà en tant que conversions prédéfinies. Dans l'exemple

```
C#
```

```
struct Convertible<T>
{
    public static implicit operator Convertible<T>(T value) {...}
    public static explicit operator T(Convertible<T> value) {...}
}
```

quand `object` le type est spécifié en tant qu'argument de type pour `T` , le deuxième opérateur déclare une conversion qui existe déjà (implicite et, par conséquent, une conversion explicite existe de tout type en type `object`).

Dans les cas où une conversion prédéfinie existe entre deux types, toutes les conversions définies par l'utilisateur entre ces types sont ignorées. Plus précisément :

- Si une conversion implicite prédéfinie (conversions implicites) existe du type `S` en type `T` , toutes les conversions définies par l'utilisateur (implicites ou explicites) de `S` à `T` sont ignorées.
- Si une conversion explicite prédéfinie (conversions explicites) existe du type `S` en type `T` , toutes les conversions explicites définies par l'utilisateur de `S` à `T` sont ignorées. En outre :

Si `T` est un type interface, les conversions implicites définies par l'utilisateur de `S` à `T` sont ignorées.

Sinon, les conversions implicites définies par l'utilisateur de `S` à `T` sont toujours prises en compte.

Pour tous les types `object` , mais, les opérateurs déclarés par le `Convertible<T>` type ci-dessus ne sont pas en conflit avec les conversions prédéfinies. Par exemple :

C#

```
void F(int i, Convertible<int> n) {
    i = n;                          // Error
    i = (int)n;                     // User-defined explicit conversion
    n = i;                          // User-defined implicit conversion
    n = (Convertible<int>)i;        // User-defined implicit conversion
}
```

Toutefois, pour `object` les conversions de type, les conversions prédéfinies masquent les conversions définies par l'utilisateur dans tous les cas, à l'exception de :

C#

```
void F(object o, Convertible<object> n) {
    o = n;                          // Pre-defined boxing conversion
```

```
    o = (object)n;                  // Pre-defined boxing conversion
    n = o;                          // User-defined implicit conversion
    n = (Convertible<object>)o;     // Pre-defined unboxing conversion
}
```

Les conversions définies par l'utilisateur ne sont pas autorisées à effectuer une conversion à partir de ou vers *INTERFACE_TYPE* s. En particulier, cette restriction garantit qu'aucune transformation définie par l'utilisateur ne se produit lors de la conversion vers un *INTERFACE_TYPE*, et qu'une conversion vers un *INTERFACE_TYPE* réussit uniquement si l'objet en cours de conversion implémente réellement le *INTERFACE_TYPE* spécifié.

La signature d'un opérateur de conversion se compose du type de source et du type de cible. (Notez qu'il s'agit de la seule forme de membre pour laquelle le type de retour participe à la signature.) La `implicit` `explicit` classification ou d'un opérateur de conversion ne fait pas partie de la signature de l'opérateur. Ainsi, une classe ou un struct ne peut pas déclarer `implicit` à la fois un opérateur de conversion et un `explicit` opérateur de conversion avec les mêmes types source et cible.

En général, les conversions implicites définies par l'utilisateur doivent être conçues pour ne jamais lever d'exceptions et ne pas perdre d'informations. Si une conversion définie par l'utilisateur peut donner lieu à des exceptions (par exemple, parce que l'argument source est hors limites) ou une perte d'informations (par exemple, en ignorant les bits de poids fort), cette conversion doit être définie comme une conversion explicite.

Dans l'exemple

C#

```
using System;

public struct Digit
{
    byte value;

    public Digit(byte value) {
        if (value < 0 || value > 9) throw new ArgumentException();
        this.value = value;
    }

    public static implicit operator byte(Digit d) {
        return d.value;
    }

    public static explicit operator Digit(byte b) {
        return new Digit(b);
    }
}
```

la conversion de `Digit` en `byte` est implicite, car elle ne lève jamais d'exception ou ne perd pas d'informations, mais la conversion de `byte` en `Digit` est explicite, car `Digit` peut uniquement représenter un sous-ensemble des valeurs possibles d'un `byte` .

Constructeurs d'instance

Un *__constructeur d'instance__ _ est un membre qui implémente les actions requises pour initialiser une instance d'une classe. Les constructeurs d'instance sont déclarés à l'aide de _constructor_declaration * s :

antlr

```
constructor_declaration
    : attributes? constructor_modifier* constructor_declarator
constructor_body
    ;

constructor_modifier
    : 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'extern'
    | constructor_modifier_unsafe
    ;

constructor_declarator
    : identifier '(' formal_parameter_list? ')' constructor_initializer?
    ;

constructor_initializer
    : ':' 'base' '(' argument_list? ')'
    | ':' 'this' '(' argument_list? ')'
    ;

constructor_body
    : block
    | ';'
    ;
```

Une *constructor_declaration* peut inclure un ensemble d' *attributs* (attributs), une combinaison valide des quatre modificateurs d'accès (modificateurs d'accès) et un `extern` modificateur (méthodes externes). Une déclaration de constructeur n'est pas autorisée à inclure le même modificateur plusieurs fois.

L' *identificateur* d'un *constructor_declarator* doit nommer la classe dans laquelle le constructeur d'instance est déclaré. Si un autre nom est spécifié, une erreur de

compilation se produit.

La *formal_parameter_list* facultative d'un constructeur d'instance est soumise aux mêmes règles que les *formal_parameter_list* d'une méthode (méthodes). La liste de paramètres formels définit la signature (signatures et surcharge) d'un constructeur d'instance et régit le processus par lequel la résolution de surcharge (inférence de type) sélectionne un constructeur d'instance particulier dans un appel.

Chacun des types référencés dans le *formal_parameter_list* d'un constructeur d'instance doit être au moins aussi accessible que le constructeur lui-même (contraintes d'accessibilité).

Le *constructor_initializer* facultatif spécifie un autre constructeur d'instance à appeler avant d'exécuter les instructions fournies dans le *constructor_body* de ce constructeur d'instance. Cela est décrit plus en détail dans les initialiseurs de constructeur.

Lorsqu'une déclaration de constructeur comprend un `extern` modificateur, le constructeur est considéré comme un ***constructeur externe** _. Étant donné qu'une déclaration de constructeur externe ne fournit aucune implémentation réelle, son _constructor_body * se compose d'un point-virgule. Pour tous les autres constructeurs, le *constructor_body* se compose d'un *bloc* qui spécifie les instructions pour initialiser une nouvelle instance de la classe. Cela correspond exactement au *bloc* d'une méthode d'instance avec un `void` type de retour (corps de méthode).

Les constructeurs d'instance ne sont pas hérités. Par conséquent, une classe n'a pas de constructeurs d'instance autres que ceux réellement déclarés dans la classe. Si une classe ne contient aucune déclaration de constructeur d'instance, un constructeur d'instance par défaut est fourni automatiquement (constructeurs par défaut).

Les constructeurs d'instance sont appelés par *object_creation_expression* s (expressions de création d'objet) et par le biais de *constructor_initializer* s.

Initialiseurs de constructeur

Tous les constructeurs d'instance (sauf ceux de `object` la classe) incluent implicitement un appel d'un autre constructeur d'instance immédiatement avant l' *constructor_body*. Le constructeur pour appeler implicitement est déterminé par la *constructor_initializer*:

- Un initialiseur de constructeur d'instance du formulaire `base(argument_list)` ou `base()` provoque l'appel d'un constructeur d'instance de la classe de base directe. Ce constructeur est sélectionné à l'aide de *argument_list* s'il existe et des règles de résolution de surcharge de la résolution de surcharge. L'ensemble de constructeurs d'instance candidat se compose de tous les constructeurs d'instance accessibles

contenus dans la classe de base directe, ou du constructeur par défaut (constructeurs par défaut), si aucun constructeur d'instance n'est déclaré dans la classe de base directe. Si cet ensemble est vide ou si un seul constructeur d'instance unique ne peut pas être identifié, une erreur de compilation se produit.

- Un initialiseur de constructeur d'instance du formulaire `this(argument-list)` ou `this()` provoque l'appel d'un constructeur d'instance de la classe elle-même. Le constructeur est sélectionné à l'aide de *argument_list* s'il existe et des règles de résolution de surcharge de la résolution de surcharge. L'ensemble de constructeurs d'instance candidat se compose de tous les constructeurs d'instance accessibles déclarés dans la classe elle-même. Si cet ensemble est vide ou si un seul constructeur d'instance unique ne peut pas être identifié, une erreur de compilation se produit. Si une déclaration de constructeur d'instance comprend un initialiseur de constructeur qui appelle le constructeur lui-même, une erreur de compilation se produit.

Si un constructeur d'instance n'a pas d'initialiseur de constructeur, un initialiseur de constructeur du formulaire `base()` est fourni implicitement. Ainsi, une déclaration de constructeur d'instance de la forme

```
C#

C(...) {...}
```

est exactement équivalent à

```
C#

C(...): base() {...}
```

La portée des paramètres fournis par l' *formal_parameter_list* d'une déclaration de constructeur d'instance comprend l'initialiseur de constructeur de cette déclaration. Ainsi, un initialiseur de constructeur est autorisé à accéder aux paramètres du constructeur. Par exemple :

```
C#

class A
{
    public A(int x, int y) {}
}

class B: A
{
```

```
    public B(int x, int y): base(x + y, x - y) {}
}
```

Un initialiseur de constructeur d'instance ne peut pas accéder à l'instance en cours de création. Par conséquent, il s'agit d'une erreur de compilation pour référencer `this` dans une expression d'argument de l'initialiseur de constructeur, comme c'est le cas une erreur de compilation pour qu'une expression d'argument référence n'importe quel membre d'instance par le biais d'un *simple_name*.

Initialiseurs de variable d'instance

Lorsqu'un constructeur d'instance n'a pas d'initialiseur de constructeur, ou qu'il a un initialiseur de constructeur de la forme `base(...)` , ce constructeur effectue implicitement les initialisations spécifiées par le *variable_initializer* s des champs d'instance déclarés dans sa classe. Cela correspond à une séquence d'assignations qui sont exécutées immédiatement après l'entrée dans le constructeur et avant l'appel implicite du constructeur de classe de base directe. Les initialiseurs de variable sont exécutés dans l'ordre textuel dans lequel ils apparaissent dans la déclaration de classe.

Exécution du constructeur

Les initialiseurs de variable sont transformés en instructions d'assignation, et ces instructions d'assignation sont exécutées avant l'appel du constructeur d'instance de la classe de base. Ce classement garantit que tous les champs d'instance sont initialisés par leurs initialiseurs de variable avant l'exécution des instructions qui ont accès à cette instance.

À partir de l'exemple

C#

```
using System;

class A
{
    public A() {
        PrintFields();
    }

    public virtual void PrintFields() {}
}

class B: A
{
    int x = 1;
```

```
    int y;

    public B() {
        y = -1;
    }

    public override void PrintFields() {
        Console.WriteLine("x = {0}, y = {1}", x, y);
    }
}
```

Lorsque `new B()` est utilisé pour créer une instance de `B` , la sortie suivante est produite :

Console

```
x = 1, y = 0
```

La valeur de `x` est 1, car l'initialiseur de variable est exécuté avant l'appel du constructeur d'instance de la classe de base. Toutefois, la valeur de `y` est 0 (la valeur par défaut d'un `int`), car l'assignation à `y` n'est pas exécutée tant que le constructeur de classe de base n'a pas été retourné.

Il est utile de considérer les initialiseurs de variable d'instance et les initialiseurs de constructeur comme des instructions qui sont insérées automatiquement avant le *constructor_body*. L'exemple

C#

```
using System;
using System.Collections;

class A
{
    int x = 1, y = -1, count;

    public A() {
        count = 0;
    }

    public A(int n) {
        count = n;
    }
}

class B: A
{
    double sqrt2 = Math.Sqrt(2.0);
    ArrayList items = new ArrayList(100);
```

```
    int max;

    public B(): this(100) {
        items.Add("default");
    }

    public B(int n): base(n - 1) {
        max = n;
    }
}
```

contient plusieurs initialiseurs de variable ; Il contient également des initialiseurs de constructeur des deux formes (`base` et `this`). L'exemple correspond au code indiqué ci-dessous, où chaque commentaire indique une instruction insérée automatiquement (la syntaxe utilisée pour les appels de constructeur insérés automatiquement n'est pas valide, mais sert simplement à illustrer le mécanisme).

C#

```
using System.Collections;

class A
{
    int x, y, count;

    public A() {
        x = 1;                      // Variable initializer
        y = -1;                     // Variable initializer
        object();                   // Invoke object() constructor
        count = 0;
    }

    public A(int n) {
        x = 1;                      // Variable initializer
        y = -1;                     // Variable initializer
        object();                   // Invoke object() constructor
        count = n;
    }
}

class B: A
{
    double sqrt2;
    ArrayList items;
    int max;

    public B(): this(100) {
        B(100);                     // Invoke B(int) constructor
        items.Add("default");
    }

    public B(int n): base(n - 1) {
```

```
        sqrt2 = Math.Sqrt(2.0);      // Variable initializer
        items = new ArrayList(100);  // Variable initializer
        A(n - 1);                    // Invoke A(int) constructor
        max = n;
    }
}
```

Constructeurs par défaut

Si une classe ne contient aucune déclaration de constructeur d'instance, un constructeur d'instance par défaut est fourni automatiquement. Ce constructeur par défaut appelle simplement le constructeur sans paramètre de la classe de base directe. Si la classe est abstraite, l'accessibilité déclarée pour le constructeur par défaut est protégée. Dans le cas contraire, l'accessibilité déclarée pour le constructeur par défaut est public. Par conséquent, le constructeur par défaut est toujours de la forme

C#

```
protected C(): base() {}
```

or

C#

```
public C(): base() {}
```

où `C` est le nom de la classe. Si la résolution de surcharge ne peut pas déterminer un meilleur candidat unique pour l'initialiseur de constructeur de classe de base, une erreur de compilation se produit.

Dans l'exemple

C#

```
class Message
{
    object sender;
    string text;
}
```

un constructeur par défaut est fourni, car la classe ne contient aucune déclaration de constructeur d'instance. Ainsi, l'exemple est exactement équivalent à

C#

```
class Message
{
    object sender;
    string text;

    public Message(): base() {}
}
```

Constructeurs privés

Quand une classe `T` déclare uniquement des constructeurs d'instance privés, il n'est pas possible que les classes situées en dehors du texte de programme de soient `T` dérivées de `T` ou de pour créer directement des instances de `T` . Ainsi, si une classe contient uniquement des membres statiques et n'est pas destinée à être instanciée, l'ajout d'un constructeur d'instance privée vide empêchera l'instanciation. Par exemple :

C#

```
public class Trig
{
    private Trig() {}        // Prevent instantiation

    public const double PI = 3.14159265358979323846;

    public static double Sin(double x) {...}
    public static double Cos(double x) {...}
    public static double Tan(double x) {...}
}
```

La `Trig` classe regroupe des constantes et des méthodes connexes, mais n'est pas destinée à être instanciée. Par conséquent, il déclare un seul constructeur d'instance privée vide. Au moins un constructeur d'instance doit être déclaré pour supprimer la génération automatique d'un constructeur par défaut.

Paramètres facultatifs du constructeur d'instance

La `this(...)` forme de l'initialiseur de constructeur est couramment utilisée conjointement avec la surcharge pour implémenter des paramètres de constructeur d'instance facultatifs. Dans l'exemple

C#

```
class Text
{
    public Text(): this(0, 0, null) {}
```

```
    public Text(int x, int y): this(x, y, null) {}

    public Text(int x, int y, string s) {
        // Actual constructor implementation
    }
}
```

les deux premiers constructeurs d'instance fournissent simplement les valeurs par défaut pour les arguments manquants. Les deux utilisent un `this(...)` initialiseur de constructeur pour appeler le troisième constructeur d'instance, qui effectue en fait le travail d'initialisation de la nouvelle instance. L'effet est celui des paramètres de constructeur facultatifs :

C#

```
Text t1 = new Text();                    // Same as Text(0, 0, null)
Text t2 = new Text(5, 10);               // Same as Text(5, 10, null)
Text t3 = new Text(5, 20, "Hello");
```

Constructeurs statiques

Un *__constructeur statique__ _ est un membre qui implémente les actions requises pour initialiser un type de classe fermé. Les constructeurs statiques sont déclarés à l'aide de _static_constructor_declaration * s :

antlr

```
static_constructor_declaration
    : attributes? static_constructor_modifiers identifier '(' ')'
static_constructor_body
    ;

static_constructor_modifiers
    : 'extern'? 'static'
    | 'static' 'extern'?
    | static_constructor_modifiers_unsafe
    ;

static_constructor_body
    : block
    | ';'
    ;
```

Une *static_constructor_declaration* peut inclure un ensemble d' *attributs* (attributs) et un `extern` modificateur (méthodes externes).

L' *identificateur* d'un *static_constructor_declaration* doit nommer la classe dans laquelle le constructeur statique est déclaré. Si un autre nom est spécifié, une erreur de compilation se produit.

Lorsqu'une déclaration de constructeur statique comprend un `extern` modificateur, le constructeur statique est considéré comme un ***constructeur statique externe** _. Étant donné qu'une déclaration de constructeur statique externe ne fournit aucune implémentation réelle, son _static_constructor_body * se compose d'un point-virgule. Pour toutes les autres déclarations de constructeur statiques, le *static_constructor_body* se compose d'un *bloc* qui spécifie les instructions à exécuter pour initialiser la classe. Cela correspond exactement au *method_body* d'une méthode statique avec un `void` type de retour (corps de méthode).

Les constructeurs statiques ne sont pas hérités et ne peuvent pas être appelés directement.

Le constructeur statique d'un type de classe fermé s'exécute au plus une fois dans un domaine d'application donné. L'exécution d'un constructeur statique est déclenchée par le premier des événements suivants pour se produire au sein d'un domaine d'application :

- Une instance du type de classe est créée.
- Tous les membres statiques du type de classe sont référencés.

Si une classe contient la `Main` méthode (démarragede l'application) dans laquelle l'exécution commence, le constructeur statique de cette classe s'exécute avant l'appel de la `Main` méthode.

Pour initialiser un nouveau type de classe fermé, tout d'abord un nouvel ensemble de champs statiques (champs d'instance et statiques) pour ce type fermé particulier est créé. Chacun des champs statiques est initialisé à sa valeur par défaut (valeurs par défaut). Ensuite, les initialiseurs de champ statique (initialisation de champ statique) sont exécutés pour ces champs statiques. Enfin, le constructeur statique est exécuté.

L'exemple

C#

```
using System;

class Test
{
    static void Main() {
        A.F();
        B.F();
```

```
    }
}

class A
{
    static A() {
        Console.WriteLine("Init A");
    }
    public static void F() {
        Console.WriteLine("A.F");
    }
}

class B
{
    static B() {
        Console.WriteLine("Init B");
    }
    public static void F() {
        Console.WriteLine("B.F");
    }
}
```

doit produire la sortie :

Console

```
Init A
A.F
Init B
B.F
```

étant donné que l'exécution du `A` constructeur statique de est déclenchée par l'appel à `A.F` , et que l'exécution du `B` constructeur statique de est déclenchée par l'appel à `B.F` .

Il est possible de construire des dépendances circulaires qui autorisent le respect des champs statiques avec des initialiseurs de variable dans leur état de valeur par défaut.

L'exemple

C#

```
using System;

class A
{
    public static int X;

    static A() {
        X = B.Y + 1;
```

```
    }
}

class B
{
    public static int Y = A.X + 1;

    static B() {}

    static void Main() {
        Console.WriteLine("X = {0}, Y = {1}", A.X, B.Y);
    }
}
```

génère la sortie

Console

```
X = 1, Y = 2
```

Pour exécuter la `Main` méthode, le système exécute d'abord l'initialiseur pour `B.Y` , avant le `B` constructeur statique de la classe. `Y`l'initialiseur de entraîne l' `A` exécution du constructeur statique de, car la valeur de `A.X` est référencée. Le constructeur statique de `A` à son tour poursuit le calcul de la valeur de `X` , ce qui permet d'extraire la valeur par défaut de `Y` , qui est égale à zéro. `A.X` est donc initialisé à 1. Le processus d'exécution des `A` initialiseurs de champ statiques et du constructeur statique se termine alors, en retournant au calcul de la valeur initiale de `Y` , dont le résultat devient 2.

Étant donné que le constructeur statique est exécuté une seule fois pour chaque type de classe construite fermée, il s'agit d'un emplacement pratique pour appliquer des contrôles au moment de l'exécution sur le paramètre de type qui ne peuvent pas être vérifiés au moment de la compilation via des contraintes (contraintes de paramètre de type). Par exemple, le type suivant utilise un constructeur statique pour appliquer que l'argument de type est une énumération :

C#

```
class Gen<T> where T: struct
{
    static Gen() {
        if (!typeof(T).IsEnum) {
            throw new ArgumentException("T must be an enum");
        }
    }
}
```

Destructeurs

Un *__destructeur__ _ est un membre qui implémente les actions requises pour détruire une instance d'une classe. Un destructeur est déclaré à l'aide d'un _destructor_declaration * :

antlr

```
destructor_declaration
    : attributes? 'extern'? '~' identifier '(' ')' destructor_body
    | destructor_declaration_unsafe
    ;

destructor_body
    : block
    | ';'
    ;
```

Une *destructor_declaration* peut inclure un ensemble d' *attributs* (attributs).

L' *identificateur* d'un *destructor_declaration* doit nommer la classe dans laquelle le destructeur est déclaré. Si un autre nom est spécifié, une erreur de compilation se produit.

Lorsqu'une déclaration de destructeur comprend un `extern` modificateur, le destructeur est considéré comme un *__destructeur externe__ _. Étant donné qu'une déclaration de destructeur externe ne fournit aucune implémentation réelle, son _destructor_body * se compose d'un point-virgule. Pour tous les autres destructeurs, le *destructor_body* se compose d'un *bloc* qui spécifie les instructions à exécuter afin de détruire une instance de la classe. Une *destructor_body* correspond exactement à la *method_body* d'une méthode d'instance avec un `void` type de retour (corps de méthode).

Les destructeurs ne sont pas hérités. Par conséquent, une classe n'a pas de destructeurs autres que celui qui peut être déclaré dans cette classe.

Étant donné qu'un destructeur ne doit pas avoir de paramètres, il ne peut pas être surchargé, donc une classe peut avoir, au plus, un destructeur.

Les destructeurs sont appelés automatiquement et ne peuvent pas être appelés explicitement. Une instance devient éligible pour la destruction lorsqu'il n'est plus possible pour un code d'utiliser cette instance. L'exécution du destructeur pour l'instance peut se produire à tout moment une fois que l'instance est éligible pour la destruction. Lorsqu'une instance est détruite, les destructeurs dans la chaîne d'héritage de cette instance sont appelés, dans l'ordre, de la plus dérivée à la moins dérivée. Un destructeur peut être exécuté sur n'importe quel thread. Pour plus d'informations sur les

règles qui régissent le moment et le mode d'exécution d'un destructeur, consultez gestion automatiquede la mémoire.

La sortie de l'exemple

C#

```
using System;

class A
{
    ~A() {
        Console.WriteLine("A's destructor");
    }
}

class B: A
{
    ~B() {
        Console.WriteLine("B's destructor");
    }
}

class Test
{
   static void Main() {
        B b = new B();
        b = null;
        GC.Collect();
        GC.WaitForPendingFinalizers();
   }
}
```

is

```
B's destructor
A's destructor
```

étant donné que les destructeurs dans une chaîne d'héritage sont appelés dans l'ordre, de la plus dérivée à la moins dérivée.

Les destructeurs sont implémentés en substituant la méthode virtuelle `Finalize` sur `System.Object` . Les programmes C# ne sont pas autorisés à substituer cette méthode ou à l'appeler (ou les remplacer) directement. Par exemple, le programme

C#

```
class A
{
    override protected void Finalize() {}    // error

    public void F() {
        this.Finalize();                      // error
    }
}
```

contient deux erreurs.

Le compilateur se comporte comme si cette méthode, et ses substitutions, n'existent pas du tout. Ainsi, ce programme :

C#

```
class A
{
    void Finalize() {}                            // permitted
}
```

est valide, et la méthode affichée masque la `System.Object` `Finalize` méthode de.

Pour une description du comportement lorsqu'une exception est levée à partir d'un destructeur, consultez la rubrique Comment les exceptions sont gérées.

Iterators

Une fonction membre (fonction members) implémentée à l'aide d'un bloc itérateur (blocs) est appelée un ***itérateur***.

Un bloc itérateur peut être utilisé comme corps d'un membre de fonction tant que le type de retour de la fonction membre correspondante est l'une des interfaces d'énumérateur (interfaces d'énumérateur) ou l'une des interfaces énumérables (interfaces énumérables). Elle peut se produire comme un *method_body*, *operator_body* ou *accessor_body*, alors que des événements, des constructeurs d'instance, des constructeurs statiques et des destructeurs ne peuvent pas être implémentés en tant qu'itérateurs.

Lorsqu'un membre de fonction est implémenté à l'aide d'un bloc itérateur, il s'agit d'une erreur de compilation pour la liste de paramètres formels de la fonction membre qui spécifie les `ref` `out` paramètres ou.

Interfaces d'énumérateur

Les ***interfaces d'énumérateur*** sont l'interface non générique `System.Collections.IEnumerator` et toutes les instanciations de l'interface générique `System.Collections.Generic.IEnumerator<T>` . Par souci de concision, dans ce chapitre, ces interfaces sont référencées en tant que `IEnumerator` et `IEnumerator<T>` , respectivement.

Interfaces énumérables

Les ***interfaces énumérables*** sont l'interface non générique `System.Collections.IEnumerable` et toutes les instanciations de l'interface générique `System.Collections.Generic.IEnumerable<T>` . Par souci de concision, dans ce chapitre, ces interfaces sont référencées en tant que `IEnumerable` et `IEnumerable<T>` , respectivement.

Type yield

Un itérateur produit une séquence de valeurs, toutes du même type. Ce type est appelé le ***type yield*** de l'itérateur.

- Type yield d'un itérateur qui retourne `IEnumerator` ou `IEnumerable` est `object` .
- Type yield d'un itérateur qui retourne `IEnumerator<T>` ou `IEnumerable<T>` est `T` .

Objets Enumerator

Lorsqu'un membre de fonction qui retourne un type d'interface énumérateur est implémenté à l'aide d'un bloc itérateur, l'appel du membre de fonction n'exécute pas immédiatement le code dans le bloc itérateur. Au lieu de cela, un ***objet énumérateur*** est créé et retourné. Cet objet encapsule le code spécifié dans le bloc Iterator, et l'exécution du code dans le bloc Iterator se produit lorsque la méthode de l'objet énumérateur `MoveNext` est appelée. Un objet énumérateur a les caractéristiques suivantes :

- Il implémente `IEnumerator` et `IEnumerator<T>` , où `T` est le type de rendement de l'itérateur.
- Elle implémente `System.IDisposable`.
- Elle est initialisée à l'aide d'une copie des valeurs d'argument (le cas échéant) et de la valeur d'instance passée à la fonction membre.
- Il a quatre États potentiels, ***Before*** *_, _Running_, _Suspended_ et _after*, et est initialement dans l'État _ *Before***., *suspended*, and *after*, and is initially in the _

*before** state.

Un objet énumérateur est généralement une instance d'une classe d'énumérateur générée par le compilateur qui encapsule le code dans le bloc itérateur et implémente les interfaces d'énumérateur, mais d'autres méthodes d'implémentation sont possibles. Si une classe d'énumération est générée par le compilateur, cette classe sera imbriquée, directement ou indirectement, dans la classe contenant la fonction membre, elle aura une accessibilité privée et un nom est réservé à l'utilisation du compilateur (identificateurs).

Un objet énumérateur peut implémenter plus d'interfaces que celles spécifiées ci-dessus.

Les sections suivantes décrivent le comportement exact des `MoveNext` `Current` membres, et `Dispose` des `IEnumerable` `IEnumerable<T>` implémentations d'interface et fournies par un objet énumérateur.

Notez que les objets Enumerator ne prennent pas en charge la `IEnumerator.Reset` méthode. L'appel de cette méthode entraîne la `System.NotSupportedException` levée d'une exception.

Méthode MoveNext

La `MoveNext` méthode d'un objet énumérateur encapsule le code d'un bloc itérateur. L'appel de la `MoveNext` méthode exécute le code dans le bloc Iterator et définit la `Current` propriété de l'objet énumérateur comme il convient. L'action exacte effectuée par `MoveNext` dépend de l'état de l'objet énumérateur lorsque `MoveNext` est appelé :

- Si l'état de l'objet énumérateur est ***antérieur***, appelez `MoveNext` :
  - Modifie l'État en ***en cours d'exécution***.
  - Initialise les paramètres (y compris `this`) du bloc itérateur aux valeurs d'argument et à la valeur d'instance enregistrées lorsque l'objet énumérateur a été initialisé.
  - Exécute le bloc itérateur à partir du début jusqu'à ce que l'exécution soit interrompue (comme décrit ci-dessous).
- Si l'état de l'objet énumérateur est ***en cours d'exécution***, le résultat de l'appel de `MoveNext` n'est pas spécifié.
- Si l'état de l'objet énumérateur est ***suspendu***, appelez `MoveNext` :
  - Modifie l'État en ***en cours d'exécution***.
  - Restaure les valeurs de tous les paramètres et variables locales (y compris celui-ci) aux valeurs enregistrées lors de la dernière suspension de l'exécution du bloc

itérateur. Notez que le contenu de tous les objets référencés par ces variables peut avoir changé depuis l'appel précédent à MoveNext.
    - Reprend l'exécution du bloc itérateur immédiatement après l' `yield return` instruction qui a provoqué la suspension de l'exécution et continue jusqu'à ce que l'exécution soit interrompue (comme décrit ci-dessous).
- Si l'état de l'objet énumérateur est ***après***, l'appel de `MoveNext` retourne `false` .

Lorsque `MoveNext` exécute le bloc Iterator, l'exécution peut être interrompue de quatre façons : par une instruction, par `yield return` une `yield break` instruction, en rencontrant la fin du bloc itérateur et par une exception levée et propagée hors du bloc itérateur.

- Quand une `yield return` instruction est rencontrée (l'instruction yield) :
    - L'expression donnée dans l'instruction est évaluée, implicitement convertie en type yield et assignée à la `Current` propriété de l'objet énumérateur.
    - L'exécution du corps de l'itérateur est suspendue. Les valeurs de tous les paramètres et variables locales (y compris `this`) sont enregistrées, comme c'est l'emplacement de cette `yield return` instruction. Si l' `yield return` instruction se trouve dans un ou plusieurs `try` blocs, les `finally` blocs associés ne sont pas exécutés pour l'instant.
    - L'état de l'objet énumérateur est modifié en ***suspendu***.
    - La `MoveNext` méthode retourne `true` à son appelant, ce qui indique que l'itération a réussi à avancer jusqu'à la valeur suivante.
- Quand une `yield break` instruction est rencontrée (l'instruction yield) :
    - Si l' `yield break` instruction se trouve dans un ou plusieurs `try` blocs, les `finally` blocs associés sont exécutés.
    - L'état de l'objet énumérateur est remplacé par ***après***.
    - La `MoveNext` méthode retourne `false` à son appelant, ce qui indique que l'itération est terminée.
- Quand la fin du corps de l'itérateur est rencontrée :
    - L'état de l'objet énumérateur est remplacé par ***après***.
    - La `MoveNext` méthode retourne `false` à son appelant, ce qui indique que l'itération est terminée.
- Lorsqu'une exception est levée et propagée en dehors du bloc Iterator :
    - `finally`Les blocs appropriés dans le corps de l'itérateur ont été exécutés par la propagation de l'exception.
    - L'état de l'objet énumérateur est remplacé par ***après***.
    - La propagation de l'exception continue à l'appelant de la `MoveNext` méthode.

La propriété actuelle

La propriété d'un objet Enumerator `Current` est affectée par `yield return` les instructions du bloc Iterator.

Lorsqu'un objet énumérateur est dans l'état *__Suspended__ _, la valeur de `Current` est la valeur définie par l'appel précédent à `MoveNext` . Lorsqu'un objet énumérateur se trouve dans les États *Before*, *Running* ou _ **after**, le résultat de l'accès `Current` n'est pas spécifié.

Pour un itérateur avec un type yield autre que `object` , le résultat de l'accès `Current` via l'implémentation de l'objet énumérateur `IEnumerable` correspond à l'accès `Current` via l'implémentation de l'objet énumérateur `IEnumerator<T>` et au cast du résultat vers `object` .

Méthode dispose

La `Dispose` méthode est utilisée pour nettoyer l'itération en plaçant l'objet énumérateur à l'état ***after*** .

- Si l'état de l'objet énumérateur est *__Before__ _, l'appel de `Dispose` change l'État en _ *après* *.
- Si l'état de l'objet énumérateur est ***en cours d'exécution***, le résultat de l'appel de `Dispose` n'est pas spécifié.
- Si l'état de l'objet énumérateur est ***suspendu***, appelez `Dispose` :
  - Modifie l'État en ***en cours d'exécution***.
  - Exécute tous les blocs finally comme si la dernière `yield return` instruction exécutée était une `yield break` instruction. Si cela entraîne la levée d'une exception et sa propagation en dehors du corps de l'itérateur, l'état de l'objet énumérateur est défini sur ***après*** et l'exception est propagée à l'appelant de la `Dispose` méthode.
  - Passe l'État à ***après***.
- Si l'état de l'objet énumérateur est ***après***, l'appel de n' `Dispose` a aucun effet.

Objets énumérables

Lorsqu'un membre de fonction qui retourne un type d'interface énumérable est implémenté à l'aide d'un bloc itérateur, l'appel de la fonction member n'exécute pas immédiatement le code dans le bloc itérateur. Au lieu de cela, un ***objet énumérable*** est créé et retourné. La méthode de l'objet énumérable `GetEnumerator` retourne un objet

énumérateur qui encapsule le code spécifié dans le bloc Iterator, et l'exécution du code dans le bloc Iterator se produit lorsque la méthode de l'objet énumérateur `MoveNext` est appelée. Un objet énumérable présente les caractéristiques suivantes :

- Il implémente `IEnumerable` et `IEnumerable<T>` , où `T` est le type de rendement de l'itérateur.
- Elle est initialisée à l'aide d'une copie des valeurs d'argument (le cas échéant) et de la valeur d'instance passée à la fonction membre.

Un objet énumérable est généralement une instance d'une classe énumérable générée par le compilateur qui encapsule le code dans le bloc itérateur et implémente les interfaces énumérables, mais d'autres méthodes d'implémentation sont possibles. Si une classe énumérable est générée par le compilateur, cette classe sera imbriquée, directement ou indirectement, dans la classe contenant la fonction membre, elle aura une accessibilité privée et un nom est réservé à l'utilisation du compilateur (identificateurs).

Un objet énumérable peut implémenter plus d'interfaces que celles spécifiées ci-dessus. En particulier, un objet énumérable peut également implémenter `IEnumerator` et `IEnumerator<T>` , ce qui lui permet de servir à la fois un énumérable et un énumérateur. Dans ce type d'implémentation, la première fois qu'une méthode d'un objet énumérable `GetEnumerator` est appelée, l'objet énumérable lui-même est retourné. Les appels suivants de l'objet énumérable `GetEnumerator` , le cas échéant, retournent une copie de l'objet énumérable. Ainsi, chaque énumérateur retourné a son propre État et les modifications apportées dans un énumérateur n'en affectent pas une autre.

Méthode GetEnumerator

Un objet énumérable fournit une implémentation des `GetEnumerator` méthodes des `IEnumerable` interfaces et `IEnumerable<T>` . Les deux `GetEnumerator` méthodes partagent une implémentation commune qui acquiert et retourne un objet énumérateur disponible. L'objet énumérateur est initialisé avec les valeurs d'argument et la valeur d'instance enregistrées lorsque l'objet énumérable a été initialisé, mais dans le cas contraire, l'objet énumérateur fonctionne comme décrit dans objets Enumerator.

Exemple d'implémentation

Cette section décrit une implémentation possible d'itérateurs en termes de constructions C# standard. L'implémentation décrite ici est basée sur les mêmes principes que ceux utilisés par le compilateur Microsoft C#, mais cela n'est en aucun cas une implémentation obligatoire ou le seul possible.

La `Stack<T>` classe suivante implémente sa `GetEnumerator` méthode à l'aide d'un itérateur. L'itérateur énumère les éléments de la pile dans l'ordre de haut en bas.

C#

```
using System;
using System.Collections;
using System.Collections.Generic;

class Stack<T>: IEnumerable<T>
{
    T[] items;
    int count;

    public void Push(T item) {
        if (items == null) {
            items = new T[4];
        }
        else if (items.Length == count) {
            T[] newItems = new T[count * 2];
            Array.Copy(items, 0, newItems, 0, count);
            items = newItems;
        }
        items[count++] = item;
    }

    public T Pop() {
        T result = items[--count];
        items[count] = default(T);
        return result;
    }

    public IEnumerator<T> GetEnumerator() {
        for (int i = count - 1; i >= 0; --i) yield return items[i];
    }
}
```

La `GetEnumerator` méthode peut être traduite en une instanciation d'une classe d'énumérateur générée par le compilateur qui encapsule le code dans le bloc itérateur, comme indiqué dans l'exemple suivant.

C#

```
class Stack<T>: IEnumerable<T>
{
    ...

    public IEnumerator<T> GetEnumerator() {
        return new __Enumerator1(this);
    }
```

```
    class __Enumerator1: IEnumerator<T>, IEnumerator
    {
        int __state;
        T __current;
        Stack<T> __this;
        int i;

        public __Enumerator1(Stack<T> __this) {
            this.__this = __this;
        }

        public T Current {
            get { return __current; }
        }

        object IEnumerator.Current {
            get { return __current; }
        }

        public bool MoveNext() {
            switch (__state) {
                case 1: goto __state1;
                case 2: goto __state2;
            }
            i = __this.count - 1;
        __loop:
            if (i < 0) goto __state2;
            __current = __this.items[i];
            __state = 1;
            return true;
        __state1:
            --i;
            goto __loop;
        __state2:
            __state = 2;
            return false;
        }

        public void Dispose() {
            __state = 2;
        }

        void IEnumerator.Reset() {
            throw new NotSupportedException();
        }
    }
}
```

Dans la traduction précédente, le code du bloc itérateur est converti en ordinateur d'État et placé dans la `MoveNext` méthode de la classe Enumerator. En outre, la variable locale `i` est transformée en champ dans l'objet énumérateur afin de pouvoir continuer à exister entre les appels de `MoveNext` .

L'exemple suivant imprime une table de multiplication simple des entiers 1 à 10. La `FromTo` méthode de l'exemple retourne un objet énumérable et est implémentée à l'aide d'un itérateur.

C#

```
using System;
using System.Collections.Generic;

class Test
{
    static IEnumerable<int> FromTo(int from, int to) {
        while (from <= to) yield return from++;
    }

    static void Main() {
        IEnumerable<int> e = FromTo(1, 10);
        foreach (int x in e) {
            foreach (int y in e) {
                Console.Write("{0,3} ", x * y);
            }
            Console.WriteLine();
        }
    }
}
```

La `FromTo` méthode peut être traduite en une instanciation d'une classe énumérable générée par le compilateur qui encapsule le code dans le bloc itérateur, comme indiqué dans l'exemple suivant.

C#

```
using System;
using System.Threading;
using System.Collections;
using System.Collections.Generic;

class Test
{
    ...

    static IEnumerable<int> FromTo(int from, int to) {
        return new __Enumerable1(from, to);
    }

    class __Enumerable1:
        IEnumerable<int>, IEnumerable,
        IEnumerator<int>, IEnumerator
    {
        int __state;
        int __current;
```

```
        int __from;
        int from;
        int to;
        int i;

        public __Enumerable1(int __from, int to) {
            this.__from = __from;
            this.to = to;
        }

        public IEnumerator<int> GetEnumerator() {
            __Enumerable1 result = this;
            if (Interlocked.CompareExchange(ref __state, 1, 0) != 0) {
                result = new __Enumerable1(__from, to);
                result.__state = 1;
            }
            result.from = result.__from;
            return result;
        }

        IEnumerator IEnumerable.GetEnumerator() {
            return (IEnumerator)GetEnumerator();
        }

        public int Current {
            get { return __current; }
        }

        object IEnumerator.Current {
            get { return __current; }
        }

        public bool MoveNext() {
            switch (__state) {
            case 1:
                if (from > to) goto case 2;
                __current = from++;
                __state = 1;
                return true;
            case 2:
                __state = 2;
                return false;
            default:
                throw new InvalidOperationException();
            }
        }

        public void Dispose() {
            __state = 2;
        }

        void IEnumerator.Reset() {
            throw new NotSupportedException();
        }
```

```
    }
}
```

La classe Enumerable implémente à la fois les interfaces Enumerable et les interfaces d'énumérateur, ce qui lui permet de servir à la fois un énumérable et un énumérateur. La première fois que la `GetEnumerator` méthode est appelée, l'objet énumérable lui-même est retourné. Les appels suivants de l'objet énumérable `GetEnumerator` , le cas échéant, retournent une copie de l'objet énumérable. Ainsi, chaque énumérateur retourné a son propre État et les modifications apportées dans un énumérateur n'en affectent pas une autre. La `Interlocked.CompareExchange` méthode est utilisée pour garantir l'opération thread-safe.

Les `from` `to` paramètres et sont convertis en champs dans la classe Enumerable. Étant donné que `from` est modifié dans le bloc Iterator, un `__from` champ supplémentaire est introduit pour contenir la valeur initiale donnée à `from` dans chaque énumérateur.

La `MoveNext` méthode lève une `InvalidOperationException` si elle est appelée lorsque `__state` est `0` . Cela permet de protéger l'utilisation de l'objet énumérable en tant qu'objet énumérateur sans appeler au préalable `GetEnumerator` .

L'exemple suivant montre une classe d'arborescence simple. La `Tree<T>` classe implémente sa `GetEnumerator` méthode à l'aide d'un itérateur. L'itérateur énumère les éléments de l'arborescence dans l'ordre des infixes.

C#

```
using System;
using System.Collections.Generic;

class Tree<T>: IEnumerable<T>
{
    T value;
    Tree<T> left;
    Tree<T> right;

    public Tree(T value, Tree<T> left, Tree<T> right) {
        this.value = value;
        this.left = left;
        this.right = right;
    }

    public IEnumerator<T> GetEnumerator() {
        if (left != null) foreach (T x in left) yield x;
        yield value;
        if (right != null) foreach (T x in right) yield x;
    }
}
```

```
class Program
{
    static Tree<T> MakeTree<T>(T[] items, int left, int right) {
        if (left > right) return null;
        int i = (left + right) / 2;
        return new Tree<T>(items[i],
            MakeTree(items, left, i - 1),
            MakeTree(items, i + 1, right));
    }

    static Tree<T> MakeTree<T>(params T[] items) {
        return MakeTree(items, 0, items.Length - 1);
    }

    // The output of the program is:
    // 1 2 3 4 5 6 7 8 9
    // Mon Tue Wed Thu Fri Sat Sun

    static void Main() {
        Tree<int> ints = MakeTree(1, 2, 3, 4, 5, 6, 7, 8, 9);
        foreach (int i in ints) Console.Write("{0} ", i);
        Console.WriteLine();

        Tree<string> strings = MakeTree(
            "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun");
        foreach (string s in strings) Console.Write("{0} ", s);
        Console.WriteLine();
    }
}
```

La `GetEnumerator` méthode peut être traduite en une instanciation d'une classe d'énumérateur générée par le compilateur qui encapsule le code dans le bloc itérateur, comme indiqué dans l'exemple suivant.

C#

```
class Tree<T>: IEnumerable<T>
{
    ...

    public IEnumerator<T> GetEnumerator() {
        return new __Enumerator1(this);
    }

    class __Enumerator1 : IEnumerator<T>, IEnumerator
    {
        Node<T> __this;
        IEnumerator<T> __left, __right;
        int __state;
        T __current;
```

```
        public __Enumerator1(Node<T> __this) {
            this.__this = __this;
        }

        public T Current {
            get { return __current; }
        }

        object IEnumerator.Current {
            get { return __current; }
        }

        public bool MoveNext() {
            try {
                switch (__state) {

                case 0:
                    __state = -1;
                    if (__this.left == null) goto __yield_value;
                    __left = __this.left.GetEnumerator();
                    goto case 1;

                case 1:
                    __state = -2;
                    if (!__left.MoveNext()) goto __left_dispose;
                    __current = __left.Current;
                    __state = 1;
                    return true;

                __left_dispose:
                    __state = -1;
                    __left.Dispose();

                __yield_value:
                    __current = __this.value;
                    __state = 2;
                    return true;

                case 2:
                    __state = -1;
                    if (__this.right == null) goto __end;
                    __right = __this.right.GetEnumerator();
                    goto case 3;

                case 3:
                    __state = -3;
                    if (!__right.MoveNext()) goto __right_dispose;
                    __current = __right.Current;
                    __state = 3;
                    return true;

                __right_dispose:
                    __state = -1;
                    __right.Dispose();
```

```
                __end:
                    __state = 4;
                    break;

                }
            }
            finally {
                if (__state < 0) Dispose();
            }
            return false;
        }

        public void Dispose() {
            try {
                switch (__state) {

                case 1:
                case -2:
                    __left.Dispose();
                    break;

                case 3:
                case -3:
                    __right.Dispose();
                    break;

                }
            }
            finally {
                __state = 4;
            }
        }

        void IEnumerator.Reset() {
            throw new NotSupportedException();
        }
    }
}
```

Les compilateurs temporaires générés par le compilateur utilisés dans les `foreach` instructions sont levés dans les `__left` `__right` champs et de l'objet énumérateur. Le `__state` champ de l'objet énumérateur est mis à jour avec soin afin que la `Dispose()` méthode correcte soit appelée correctement si une exception est levée. Notez qu'il n'est pas possible d'écrire le code traduit avec des `foreach` instructions simples.

Fonctions asynchrones

Une méthode (méthodes) ou une fonction anonyme (expressions de fonction anonymes) avec le `async` modificateur est appelée « ***Async Function** _ ». En général, le

terme _ *Async** est utilisé pour décrire tout genre de fonction qui a le `async` modificateur.

Il s'agit d'une erreur de compilation pour la liste de paramètres formels d'une fonction Async pour spécifier des `ref` `out` paramètres ou.

La *return_type* d'une méthode Async doit être `void` ou un ***type de tâche***. Les types de tâches sont `System.Threading.Tasks.Task` et les types construits à partir de `System.Threading.Tasks.Task<T>` . Par souci de concision, dans ce chapitre, ces types sont référencés en tant que `Task` et `Task<T>` , respectivement. Une méthode Async qui retourne un type de tâche est dite retournant des tâches.

La définition exacte des types de tâches est définie par l'implémentation, mais du point de vue du langage, un type de tâche se trouve dans l'un des États incomplet, SUCCEEDED ou Faulted. Une tâche ayant généré une erreur enregistre une exception pertinente. Une opération réussie `Task<T>` enregistre un résultat de type `T` . Les types de tâches sont await, et peuvent par conséquent être les opérandes des expressions await (expression await).

Un appel de fonction Async a la possibilité de suspendre l'évaluation à l'aide d'expressions await (expression await) dans son corps. L'évaluation peut être reprise ultérieurement au moment de l'interruption de l'expression await au moyen d'un **délégué de reprise** _. Le délégué de reprise est de type `System.Action` , et lorsqu'il est appelé, l'évaluation de l'appel de la fonction Async reprend à partir de l'expression await là où elle s'est arrêtée. L'*appelant actuel** d'un appel de fonction Async est l'appelant d'origine si l'appel de fonction n'a jamais été interrompu ou l'appelant le plus récent du délégué de reprise dans le cas contraire.

Évaluation d'une fonction asynchrone retournant des tâches

L'appel d'une fonction Async retournant des tâches provoque la génération d'une instance du type de tâche retourné. C'est ce que l'on appelle la ***tâche de retour*** de la fonction Async. L'état initial de la tâche est incomplet.

Le corps de la fonction Async est ensuite évalué jusqu'à ce qu'il soit suspendu (en atteignant une expression await) ou se termine, auquel point le contrôle est retourné à l'appelant, ainsi que la tâche de retour.

Lorsque le corps de la fonction Async se termine, la tâche de retour est déplacée de l'état incomplet :

- Si le corps de la fonction se termine suite à l'atteinte d'une instruction return ou à la fin du corps, toute valeur de résultat est enregistrée dans la tâche de retour, qui est placée dans un état de réussite.
- Si le corps de la fonction se termine à la suite d'une exception non interceptée (instruction throw), l'exception est enregistrée dans la tâche de retour qui est placée dans un état d'erreur.

Évaluation d'une fonction Async qui retourne void

Si le type de retour de la fonction Async est `void` , l'évaluation diffère de la méthode ci-dessus de la manière suivante : étant donné qu'aucune tâche n'est retournée, la fonction communique à la place l'achèvement et les exceptions au ***contexte de synchronisation*** du thread actuel. La définition exacte du contexte de synchronisation dépend de l'implémentation, mais est une représentation de « où » le thread actuel est en cours d'exécution. Le contexte de synchronisation est notifié lorsque l'évaluation d'une fonction Async qui retourne une valeur void commence, se termine correctement ou provoque la levée d'une exception non interceptée.

Cela permet au contexte d'effectuer le suivi du nombre de fonctions Async à retour void qui s'exécutent sous celui-ci et de décider comment propager les exceptions en provenance de ces dernières.

Structures

Article • 13/05/2021

Les structs sont similaires aux classes en ce qu'ils représentent des structures de données qui peuvent contenir des membres de données et des fonctions membres. Toutefois, contrairement aux classes, les structs sont des types valeur et ne nécessitent pas d'allocation de tas. Une variable d'un type struct contient directement les données de la structure, alors qu'une variable d'un type de classe contient une référence aux données, la dernière appelée objet.

Les structures sont particulièrement utiles pour les petites structures de données qui ont une sémantique par rapport à leurs valeurs. Les nombres complexes, les points dans un système de coordonnées ou les paires clé-valeur dans un dictionnaire sont de bons exemples de structures. La clé de ces structures de données est qu'elles ont peu de membres de données, qu'elles ne nécessitent pas l'utilisation de l'héritage ou de l'identité référentielle, et qu'elles peuvent être implémentées facilement à l'aide de la sémantique de valeur où l'assignation copie la valeur au lieu de la référence.

Comme décrit dans types simples, les types simples fournis par C#, tels que `int` , `double` et `bool` , sont en fait tous des types struct. Tout comme ces types prédéfinis sont des structs, il est également possible d'utiliser des structs et la surcharge d'opérateur pour implémenter de nouveaux types « primitifs » en langage C#. Deux exemples de ces types sont fournis à la fin de ce chapitre (exemples de struct).

Déclarations de struct

Un *struct_declaration* est un *type_declaration* (déclarations de type) qui déclare un nouveau struct :

antlr

```
struct_declaration
    : attributes? struct_modifier* 'partial'? 'struct' identifier
type_parameter_list?
      struct_interfaces? type_parameter_constraints_clause* struct_body ';'?
    ;
```

Un *struct_declaration* se compose d'un ensemble facultatif d' *attributs* (attributs), suivi d'un ensemble facultatif de *struct_modifier* s (modificateurs de struct). suivi d'un `partial` modificateur facultatif, suivi du mot clé `struct` et d'un *identificateur* qui nomme le struct, suivi d'une spécification facultative *type_parameter_list* (paramètres de type),

suivie d'une spécification facultative *Struct_interfaces* (modificateur partielle), suivie d'une spécification facultative *type_parameter_constraints_clause* s (contraintes de paramètre de type), suivie d'une *struct_body* (corps de struct), éventuellement suivie d'un point-virgule.

Modificateurs de struct

Une *struct_declaration* peut éventuellement inclure une séquence de modificateurs de struct :

antlr

```
struct_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | struct_modifier_unsafe
    ;
```

Il s'agit d'une erreur au moment de la compilation pour que le même modificateur apparaisse plusieurs fois dans une déclaration de struct.

Les modificateurs d'une déclaration de struct ont la même signification que ceux d'une déclaration de classe (déclarations de classe).

Modificateur partiel

Le `partial` modificateur indique que ce *struct_declaration* est une déclaration de type partiel. Plusieurs déclarations de struct partielles portant le même nom dans un espace de noms englobant ou une déclaration de type sont combinées pour former une déclaration de struct, suivant les règles spécifiées dans les types partiels.

Interfaces de struct

Une déclaration de struct peut inclure une spécification *struct_interfaces* , auquel cas le struct est dit d'implémenter directement les types d'interfaces donnés.

antlr

```
struct_interfaces
    : ':' interface_type_list
    ;
```

Les implémentations d'interface sont abordées plus en détail dans implémentations d'interface.

Corps du struct

La *struct_body* d'un struct définit les membres du struct.

antlr

```
struct_body
    : '{' struct_member_declaration* '}'
    ;
```

Membres de struct

Les membres d'un struct se composent des membres introduits par son *struct_member_declaration* s et des membres hérités du type `System.ValueType` .

antlr

```
struct_member_declaration
    : constant_declaration
    | field_declaration
    | method_declaration
    | property_declaration
    | event_declaration
    | indexer_declaration
    | operator_declaration
    | constructor_declaration
    | static_constructor_declaration
    | type_declaration
    | struct_member_declaration_unsafe
    ;
```

À l'exception des différences constatées dans les différences entre les classes et les structs, les descriptions des membres de classe fournis dans les membres de classe via des fonctions Async s'appliquent également aux membres de la structure.

Différences entre classe et struct

Les structs diffèrent des classes de plusieurs manières importantes :

- Les structs sont des types valeur (sémantique de valeur).

- Tous les types struct héritent implicitement de la classe `System.ValueType` (héritage).
- L'assignation à une variable d'un type struct crée une copie de la valeur assignée (affectation).
- La valeur par défaut d'un struct correspond à la valeur produite en affectant à tous les champs de type valeur leur valeur par défaut et à tous les champs de type référence `null` (valeurs par défaut).
- Les opérations de boxing et d'unboxing sont utilisées pour effectuer une conversion entre un type struct et `object` (boxing et unboxing).
- La signification de `this` est différente pour les structs (cet accès).
- Les déclarations de champ d'instance pour un struct ne sont pas autorisées à inclure des initialiseurs de variable (initialiseurs de champ).
- Un struct n'est pas autorisé à déclarer un constructeur d'instance sans paramètre (constructeurs).
- Un struct n'est pas autorisé à déclarer un destructeur (destructeurs).

Sémantique de valeur

Les structs sont des types valeur (types valeur) et sont dits sémantiques de valeur. En revanche, les classes sont des types référence (types référence) et sont dites avoir une sémantique de référence.

Une variable d'un type struct contient directement les données de la structure, alors qu'une variable d'un type de classe contient une référence aux données, la dernière appelée objet. Lorsqu'un struct `B` contient un champ d'instance de type `A` et `A` qu'il s'agit d'un type struct, il s'agit d'une erreur de compilation qui `A` dépend de ou d' `B` un type construit à partir de `B` . Un struct `X` * **dépend directement de** _ un struct `Y` si `X` contient un champ d'instance de type `Y` . Compte tenu de cette définition, l'ensemble complet des structs dont dépend un struct est la fermeture transitive de la relation _ *_dépend directement de_**. Par exemple

```
C#
```

```
struct Node
{
    int data;
    Node next; // error, Node directly depends on itself
}
```

est une erreur, car `Node` contient un champ d'instance de son propre type. Autre exemple

C#

```
struct A { B b; }

struct B { C c; }

struct C { A a; }
```

est une erreur, car chacun des types `A` , et dépend l'un de l' `B` `C` autre.

Avec les classes, deux variables peuvent faire référence au même objet et, par conséquent, les opérations sur une variable peuvent affecter l'objet référencé par l'autre variable. Avec les structs, les variables ont chacune leur propre copie des données (sauf dans le cas des `ref` `out` variables de paramètre et), et il n'est pas possible pour les opérations sur l'une d'affecter l'autre. En outre, étant donné que les structs ne sont pas des types référence, il n'est pas possible que les valeurs d'un type struct soient `null` .

Compte tenu de la déclaration

C#

```
struct Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

fragment de code

C#

```
Point a = new Point(10, 10);
Point b = a;
a.x = 100;
System.Console.WriteLine(b.x);
```

génère la valeur `10` . L'assignation de `a` à `b` crée une copie de la valeur et `b` n'est donc pas affectée par l'assignation à `a.x` . Ayant `Point` été déclarée en tant que classe, la sortie est `100` due au fait que `a` et `b` référence le même objet.

Héritage

Tous les types struct héritent implicitement de la classe `System.ValueType` , qui, à son tour, hérite de la classe `object` . Une déclaration de struct peut spécifier une liste d'interfaces implémentées, mais il n'est pas possible pour une déclaration de struct de spécifier une classe de base.

Les types struct ne sont jamais abstraits et sont toujours implicitement sealed. `abstract` `sealed` Par conséquent, les modificateurs et ne sont pas autorisés dans une déclaration de struct.

Étant donné que l'héritage n'est pas pris en charge pour les structs, l'accessibilité déclarée d'un membre de struct ne peut pas être `protected` ou `protected internal` .

Les fonctions membres d'un struct ne peuvent pas être `abstract` ou `virtual` , et le `override` modificateur est autorisé uniquement à substituer des méthodes héritées de `System.ValueType` .

Affectation

L'assignation à une variable d'un type struct crée une copie de la valeur assignée. Cela diffère d'une assignation à une variable d'un type de classe, qui copie la référence, mais pas l'objet identifié par la référence.

Semblable à une assignation, lorsqu'un struct est passé en tant que paramètre de valeur ou retourné comme résultat d'un membre de fonction, une copie de la structure est créée. Un struct peut être passé par référence à un membre de fonction à l'aide d'un `ref` `out` paramètre ou.

Quand une propriété ou un indexeur d'un struct est la cible d'une assignation, l'expression d'instance associée à l'accès à la propriété ou à l'indexeur doit être classée en tant que variable. Si l'expression d'instance est classée en tant que valeur, une erreur de compilation se produit. Cela est décrit plus en détail dans la section affectation simple.

Valeurs par défaut

Comme décrit dans valeurs par défaut, plusieurs types de variables sont automatiquement initialisés à leur valeur par défaut lors de leur création. Pour les variables de types classe et d'autres types référence, cette valeur par défaut est `null` . Toutefois, étant donné que les structs sont des types valeur qui ne peuvent pas être `null` , la valeur par défaut d'un struct correspond à la valeur produite en affectant à

tous les champs de type valeur leur valeur par défaut et à tous les champs de type référence la valeur `null` .

En faisant référence à la `Point` structure déclarée ci-dessus, l'exemple

```
C#

Point[] a = new Point[100];
```

Initialise chaque `Point` dans le tableau à la valeur produite en définissant les `x` `y` champs et sur zéro.

La valeur par défaut d'un struct correspond à la valeur retournée par le constructeur par défaut du struct (constructeurs par défaut). Contrairement à une classe, un struct n'est pas autorisé à déclarer un constructeur d'instance sans paramètre. Au lieu de cela, chaque struct a implicitement un constructeur d'instance sans paramètre qui retourne toujours la valeur qui résulte de l'affectation de la valeur par défaut à tous les champs de type de valeur et de tous les champs de type référence à `null` .

Les structs doivent être conçus pour considérer l'état d'initialisation par défaut comme un état valide. Dans l'exemple

```
C#

using System;

struct KeyValuePair
{
    string key;
    string value;

    public KeyValuePair(string key, string value) {
        if (key == null || value == null) throw new ArgumentException();
        this.key = key;
        this.value = value;
    }
}
```

le constructeur d'instance défini par l'utilisateur est protégé contre les valeurs NULL uniquement lorsqu'il est appelé explicitement. Dans les cas où une `KeyValuePair` variable est soumise à l'initialisation de la valeur par défaut, les `key` champs et ont la valeur `value` null et la structure doit être préparée pour gérer cet État.

Boxing et unboxing

Une valeur d'un type de classe peut être convertie en type `object` ou en un type interface implémenté par la classe en traitant simplement la référence comme un autre type au moment de la compilation. De même, une valeur de type `object` ou une valeur d'un type d'interface peut être reconvertie en un type de classe sans modifier la référence (mais bien évidemment, une vérification de type au moment de l'exécution est nécessaire dans ce cas).

Étant donné que les structs ne sont pas des types référence, ces opérations sont implémentées différemment pour les types struct. Quand une valeur d'un type struct est convertie en type `object` ou en un type interface implémenté par le struct, une opération boxing est effectuée. De même, lorsqu'une valeur de type `object` ou une valeur d'un type d'interface est reconvertie en type struct, une opération de conversion unboxing a lieu. Une différence essentielle par rapport aux mêmes opérations sur les types de classe est que le boxing et l'unboxing copient la valeur de struct à l'intérieur ou à l'extérieur de l'instance boxed. Ainsi, à la suite d'une opération boxing ou unboxing, les modifications apportées au struct unboxed ne sont pas reflétées dans le struct boxed.

Lorsqu'un type struct se substitue à une méthode virtuelle héritée de `System.Object` (telle que `Equals` , `GetHashCode` ou `ToString`), l'appel de la méthode virtuelle via une instance du type struct ne provoque pas de conversion boxing. Cela est vrai même lorsque le struct est utilisé comme paramètre de type et que l'appel se produit via une instance du type de paramètre de type. Par exemple :

C#

```
using System;

struct Counter
{
    int value;

    public override string ToString() {
        value++;
        return value.ToString();
    }
}

class Program
{
    static void Test<T>() where T: new() {
        T x = new T();
        Console.WriteLine(x.ToString());
        Console.WriteLine(x.ToString());
        Console.WriteLine(x.ToString());
    }
```

```
    static void Main() {
        Test<Counter>();
    }
}
```

La sortie du programme est la suivante :

Console

```
1
2
3
```

Bien qu'il s'agisse d'un mauvais style pour `ToString` avoir des effets secondaires, l'exemple montre qu'aucune conversion boxing ne s'est produite pour les trois appels de `x.ToString()` .

De même, le boxing ne se produit jamais implicitement lors de l'accès à un membre sur un paramètre de type restreint. Supposons, par exemple, `ICounter` qu'une interface contient une méthode `Increment` qui peut être utilisée pour modifier une valeur. Si `ICounter` est utilisé en tant que contrainte, l'implémentation de la `Increment` méthode est appelée avec une référence à la variable qui `Increment` a été appelée sur, jamais une copie boxed.

C#

```
using System;

interface ICounter
{
    void Increment();
}

struct Counter: ICounter
{
    int value;

    public override string ToString() {
        return value.ToString();
    }

    void ICounter.Increment() {
        value++;
    }
}

class Program
{
```

```
    static void Test<T>() where T: ICounter, new() {
        T x = new T();
        Console.WriteLine(x);
        x.Increment();                  // Modify x
        Console.WriteLine(x);
        ((ICounter)x).Increment();      // Modify boxed copy of x
        Console.WriteLine(x);
    }

    static void Main() {
        Test<Counter>();
    }
}
```

Le premier appel à `Increment` modifie la valeur dans la variable `x` . Cela n'est pas équivalent au deuxième appel à `Increment` , qui modifie la valeur dans une copie boxed de `x` . Ainsi, la sortie du programme est la suivante :

Console

```
0
1
1
```

Pour plus d'informations sur le boxing et l'unboxing, consultez conversion boxing et unboxing.

Signification de ce

Dans un constructeur d'instance ou un membre de fonction d'instance d'une classe, `this` est classé comme une valeur. Ainsi, bien que `this` puisse être utilisé pour faire référence à l'instance pour laquelle la fonction membre a été appelée, il n'est pas possible d'assigner à `this` dans une fonction membre d'une classe.

Dans un constructeur d'instance d'un struct, `this` correspond à un `out` paramètre du type struct, et dans une fonction membre d'instance d'un struct, `this` correspond à un `ref` paramètre du type struct. Dans les deux cas, `this` est classée en tant que variable et il est possible de modifier l'ensemble du struct pour lequel la fonction membre a été appelée en affectant à `this` ou en passant ce en tant que `ref` `out` paramètre ou.

Initialiseurs de champ

Comme décrit dans valeurs par défaut, la valeur par défaut d'un struct se compose de la valeur qui résulte de l'affectation de la valeur par défaut à tous les champs de type de

valeur et de tous les champs de type référence à `null` . Pour cette raison, un struct n'autorise pas les déclarations de champ d'instance à inclure des initialiseurs de variable. Cette restriction s'applique uniquement aux champs d'instance. Les champs statiques d'un struct sont autorisés à inclure des initialiseurs de variable.

L'exemple

C#

```
struct Point
{
    public int x = 1;  // Error, initializer not permitted
    public int y = 1;  // Error, initializer not permitted
}
```

est erroné, car les déclarations de champ d'instance incluent des initialiseurs de variable.

Constructeurs

Contrairement à une classe, un struct n'est pas autorisé à déclarer un constructeur d'instance sans paramètre. Au lieu de cela, chaque struct a implicitement un constructeur d'instance sans paramètre qui retourne toujours la valeur qui résulte de l'affectation de la valeur par défaut à tous les champs de type de valeur et de tous les champs de type référence à la valeur null (constructeurs par défaut). Un struct peut déclarer des constructeurs d'instance ayant des paramètres. Par exemple

C#

```
struct Point
{
    int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Compte tenu de la déclaration ci-dessus, les instructions

C#

```
Point p1 = new Point();
Point p2 = new Point(0, 0);
```

tous deux créent un `Point` avec `x` et sont `y` initialisés à zéro.

Un constructeur d'instance de struct n'est pas autorisé à inclure un initialiseur de constructeur du formulaire `base(...)` .

Si le constructeur d'instance de struct ne spécifie pas d'initialiseur de constructeur, la `this` variable correspond à un `out` paramètre du type struct, et semblable à un `out` paramètre, `this` doit être assignée définitivement (assignation définie) à chaque emplacement où le constructeur retourne. Si le constructeur d'instance de struct spécifie un initialiseur de constructeur, la `this` variable correspond à un `ref` paramètre du type struct, et semblable à un `ref` paramètre, `this` est considéré comme définitivement assigné à l'entrée dans le corps du constructeur. Examinez l'implémentation du constructeur d'instance ci-dessous :

C#

```
struct Point
{
    int x, y;

    public int X {
        set { x = value; }
    }

    public int Y {
        set { y = value; }
    }

    public Point(int x, int y) {
        X = x;        // error, this is not yet definitely assigned
        Y = y;        // error, this is not yet definitely assigned
    }
}
```

Aucune fonction membre d'instance (y compris les accesseurs set des propriétés `X` et `Y`) ne peut être appelée tant que tous les champs du struct en cours de construction n'ont pas été définitivement assignés. La seule exception concerne les propriétés implémentées automatiquement (Propriétés implémentées automatiquement). Les règles d'assignation définie (expressions d'assignation simples) exemptent spécifiquement l'assignation à une propriété automatique d'un type struct au sein d'un constructeur d'instance de ce type struct : une telle assignation est considérée comme une assignation définie du champ de stockage masqué de la propriété automatique. Ainsi, les éléments suivants sont autorisés :

C#

```
struct Point
{
    public int X { get; set; }
    public int Y { get; set; }

    public Point(int x, int y) {
        X = x;      // allowed, definitely assigns backing field
        Y = y;      // allowed, definitely assigns backing field
    }
```

Destructeurs

Un struct n'est pas autorisé à déclarer un destructeur.

Constructeurs statiques

Les constructeurs statiques pour les structs suivent la plupart des règles de la même façon que pour les classes. L'exécution d'un constructeur statique pour un type struct est déclenchée par le premier des événements suivants pour se produire au sein d'un domaine d'application :

- Un membre statique du type struct est référencé.
- Un constructeur déclaré explicitement du type struct est appelé.

La création de valeurs par défaut (valeurs par défaut) de types struct ne déclenche pas le constructeur statique. (Par exemple, il s'agit de la valeur initiale des éléments d'un tableau).

Exemples de structs

L'exemple suivant montre deux exemples significatifs d'utilisation de `struct` types pour créer des types qui peuvent être utilisés de la même façon que les types prédéfinis du langage, mais avec la sémantique modifiée.

Type d'entier de base de données

Le `DBInt` struct ci-dessous implémente un type entier qui peut représenter l'ensemble complet des valeurs du `int` type, ainsi qu'un État supplémentaire qui indique une valeur inconnue. Un type avec ces caractéristiques est couramment utilisé dans les bases de données.

C#

```
using System;

public struct DBInt
{
    // The Null member represents an unknown DBInt value.

    public static readonly DBInt Null = new DBInt();

    // When the defined field is true, this DBInt represents a known value
    // which is stored in the value field. When the defined field is false,
    // this DBInt represents an unknown value, and the value field is 0.

    int value;
    bool defined;

    // Private instance constructor. Creates a DBInt with a known value.

    DBInt(int value) {
        this.value = value;
        this.defined = true;
    }

    // The IsNull property is true if this DBInt represents an unknown
value.

    public bool IsNull { get { return !defined; } }

    // The Value property is the known value of this DBInt, or 0 if this
    // DBInt represents an unknown value.

    public int Value { get { return value; } }

    // Implicit conversion from int to DBInt.

    public static implicit operator DBInt(int x) {
        return new DBInt(x);
    }

    // Explicit conversion from DBInt to int. Throws an exception if the
    // given DBInt represents an unknown value.

    public static explicit operator int(DBInt x) {
        if (!x.defined) throw new InvalidOperationException();
        return x.value;
    }

    public static DBInt operator +(DBInt x) {
        return x;
    }

    public static DBInt operator -(DBInt x) {
        return x.defined ? -x.value : Null;
    }
```

```
    public static DBInt operator +(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value + y.value: Null;
    }

    public static DBInt operator -(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value - y.value: Null;
    }

    public static DBInt operator *(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value * y.value: Null;
    }

    public static DBInt operator /(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value / y.value: Null;
    }

    public static DBInt operator %(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value % y.value: Null;
    }

    public static DBBool operator ==(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value == y.value: DBBool.Null;
    }

    public static DBBool operator !=(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value != y.value: DBBool.Null;
    }

    public static DBBool operator >(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value > y.value: DBBool.Null;
    }

    public static DBBool operator <(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value < y.value: DBBool.Null;
    }

    public static DBBool operator >=(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value >= y.value: DBBool.Null;
    }

    public static DBBool operator <=(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value <= y.value: DBBool.Null;
    }

    public override bool Equals(object obj) {
        if (!(obj is DBInt)) return false;
        DBInt x = (DBInt)obj;
        return value == x.value && defined == x.defined;
    }

    public override int GetHashCode() {
        return value;
    }

    public override string ToString() {
```

```
        return defined? value.ToString(): "DBInt.Null";
    }
}
```

Type booléen de base de données

Le `DBBool` struct ci-dessous implémente un type logique à trois valeurs. Les valeurs possibles de ce type sont `DBBool.True` , `DBBool.False` et `DBBool.Null` , où le `Null` membre indique une valeur inconnue. Ces types logiques à trois valeurs sont couramment utilisés dans les bases de données.

C#

```
using System;

public struct DBBool
{
    // The three possible DBBool values.

    public static readonly DBBool Null = new DBBool(0);
    public static readonly DBBool False = new DBBool(-1);
    public static readonly DBBool True = new DBBool(1);

    // Private field that stores -1, 0, 1 for False, Null, True.

    sbyte value;

    // Private instance constructor. The value parameter must be -1, 0, or
1.

    DBBool(int value) {
        this.value = (sbyte)value;
    }

    // Properties to examine the value of a DBBool. Return true if this
    // DBBool has the given value, false otherwise.

    public bool IsNull { get { return value == 0; } }

    public bool IsFalse { get { return value < 0; } }

    public bool IsTrue { get { return value > 0; } }

    // Implicit conversion from bool to DBBool. Maps true to DBBool.True and
    // false to DBBool.False.

    public static implicit operator DBBool(bool x) {
        return x? True: False;
    }

    // Explicit conversion from DBBool to bool. Throws an exception if the
```

```
    // given DBBool is Null, otherwise returns true or false.

    public static explicit operator bool(DBBool x) {
        if (x.value == 0) throw new InvalidOperationException();
        return x.value > 0;
    }

    // Equality operator. Returns Null if either operand is Null, otherwise
    // returns True or False.

    public static DBBool operator ==(DBBool x, DBBool y) {
        if (x.value == 0 || y.value == 0) return Null;
        return x.value == y.value? True: False;
    }

    // Inequality operator. Returns Null if either operand is Null,
otherwise
    // returns True or False.

    public static DBBool operator !=(DBBool x, DBBool y) {
        if (x.value == 0 || y.value == 0) return Null;
        return x.value != y.value? True: False;
    }

    // Logical negation operator. Returns True if the operand is False, Null
    // if the operand is Null, or False if the operand is True.

    public static DBBool operator !(DBBool x) {
        return new DBBool(-x.value);
    }

    // Logical AND operator. Returns False if either operand is False,
    // otherwise Null if either operand is Null, otherwise True.

    public static DBBool operator &(DBBool x, DBBool y) {
        return new DBBool(x.value < y.value? x.value: y.value);
    }

    // Logical OR operator. Returns True if either operand is True,
otherwise
    // Null if either operand is Null, otherwise False.

    public static DBBool operator |(DBBool x, DBBool y) {
        return new DBBool(x.value > y.value? x.value: y.value);
    }

    // Definitely true operator. Returns true if the operand is True, false
    // otherwise.

    public static bool operator true(DBBool x) {
        return x.value > 0;
    }

    // Definitely false operator. Returns true if the operand is False,
false
```

```
    // otherwise.

    public static bool operator false(DBBool x) {
        return x.value < 0;
    }

    public override bool Equals(object obj) {
        if (!(obj is DBBool)) return false;
        return value == ((DBBool)obj).value;
    }

    public override int GetHashCode() {
        return value;
    }

    public override string ToString() {
        if (value > 0) return "DBBool.True";
        if (value < 0) return "DBBool.False";
        return "DBBool.Null";
    }
}
```

Tableaux

Article • 13/05/2021

Un tableau est une structure de données qui contient un certain nombre de variables accessibles par le biais d'index calculés. Les variables contenues dans un tableau, également appelé éléments du tableau, sont tous du même type, et ce type est appelé type d'élément du tableau.

Un tableau a un rang qui détermine le nombre d'index associés à chaque élément de tableau. Le rang d'un tableau est également appelé les dimensions du tableau. Un tableau avec le rang 1 est appelé un ***tableau unidimensionnel*** _. Un tableau avec un rang supérieur à un est appelé tableau à plusieurs *dimensions** *. Les tableaux multidimensionnels dimensionnés spécifiques sont souvent appelés tableaux à deux dimensions, tableaux à trois dimensions, etc.

Chaque dimension d'un tableau a une longueur associée qui est un nombre entier supérieur ou égal à zéro. Les longueurs de dimensions ne font pas partie du type du tableau, mais elles sont plutôt établies quand une instance du type de tableau est créée au moment de l'exécution. La longueur d'une dimension détermine la plage d'index valide pour cette dimension : pour une dimension de longueur, les `N` index peuvent être compris entre `0` et `N - 1` inclusifs. Le nombre total d'éléments dans un tableau est le produit des longueurs de chaque dimension du tableau. Si une ou plusieurs dimensions d'un tableau ont une longueur de zéro, le tableau est considéré comme vide.

Le type d'élément d'un tableau peut être de n'importe quel type, y compris un type tableau.

Types de tableaux

Un type de tableau est écrit sous la forme d'un *non_array_type* suivi d'un ou plusieurs *rank_specifier* s :

antlr

```
array_type
    : non_array_type rank_specifier+
    ;

non_array_type
    : type
    ;

rank_specifier
```

```
    : '[' dim_separator* ']'
    ;

dim_separator
    : ','
    ;
```

Un *non_array_type* est un *type* qui n'est pas lui-même un *array_type*.

Le rang d'un type tableau est donné par le *rank_specifier* le plus à gauche dans le *array_type*: un *rank_specifier* indique que le tableau est un tableau avec un rang égal à un et le nombre de `,` jetons «» dans le *rank_specifier*.

Le type d'élément d'un type tableau est le type qui résulte de la suppression du *rank_specifier* le plus à gauche :

- Un type tableau de la forme `T[R]` est un tableau avec `R` un rang et un type d'élément non tableau `T` .
- Un type tableau de la forme `T[R][R1]...[Rn]` est un tableau avec le rang `R` et un type d'élément `T[R1]...[Rn]` .

En effet, les *rank_specifier* s sont lues de gauche à droite avant le type d'élément non tableau final. Le type `int[][,,][,]` est un tableau unidimensionnel de tableaux à trois dimensions de tableaux à deux dimensions de `int` .

Au moment de l'exécution, une valeur d'un type de tableau peut être `null` ou une référence à une instance de ce type de tableau.

Type System. Array

Le type `System.Array` est le type de base abstrait de tous les types tableau. Une conversion de référence implicite (conversions de référence implicites) existe à partir de n'importe quel type de tableau vers `System.Array` , et une conversion de référence explicite (conversions de références explicites) existe depuis `System.Array` vers n'importe quel type de tableau. Notez qu' `System.Array` il ne s'agit pas d'un *array_type*. Au lieu de cela, il s'agit d'un *class_type* duquel tous les *array_type* sont dérivés.

Au moment de l'exécution, une valeur de type `System.Array` peut être `null` ou une référence à une instance de n'importe quel type de tableau.

Tableaux et interface IList générique

Un tableau unidimensionnel `T[]` implémente l'interface `System.Collections.Generic.IList<T>` (`IList<T>` pour Short) et ses interfaces de base. En conséquence, il existe une conversion implicite de `T[]` en `IList<T>` et ses interfaces de base. En outre, s'il existe une conversion de référence implicite de `S` en, `T` puis `S[]` implémente `IList<T>` et qu'il existe une conversion de référence implicite de `S[]` en `IList<T>` et ses interfaces de base (conversions de référence implicites). S'il existe une conversion de référence explicite de `S` en, `T` il existe une conversion de référence explicite de `S[]` en `IList<T>` et ses interfaces de base (conversions de référence explicites). Par exemple :

C#

```
using System.Collections.Generic;

class Test
{
    static void Main() {
        string[] sa = new string[5];
        object[] oa1 = new object[5];
        object[] oa2 = sa;

        IList<string> lst1 = sa;                    // Ok
        IList<string> lst2 = oa1;                   // Error, cast needed
        IList<object> lst3 = sa;                    // Ok
        IList<object> lst4 = oa1;                   // Ok

        IList<string> lst5 = (IList<string>)oa1;    // Exception
        IList<string> lst6 = (IList<string>)oa2;    // Ok
    }
}
```

L'assignation `lst2 = oa1` génère une erreur au moment de la compilation, car la conversion de `object[]` en `IList<string>` est une conversion explicite, et non implicite. Le Cast `(IList<string>)oa1` entraîne la levée d'une exception au moment de l'exécution, car `oa1` référence un `object[]` et non un `string[]` . Toutefois, la conversion n' `(IList<string>)oa2` entraîne pas la levée d'une exception dans la mesure où `oa2` référence un `string[]` .

Lorsqu'il existe une conversion de référence implicite ou explicite de `S[]` en `IList<T>` , il existe également une conversion de référence explicite de `IList<T>` et ses interfaces de base en `S[]` (conversions de référence explicites).

Lorsqu'un type `S[]` de tableau implémente `IList<T>` , certains membres de l'interface implémentée peuvent lever des exceptions. Le comportement précis de l'implémentation de l'interface dépasse le cadre de cette spécification.

Création de tableau

Les instances de tableau sont créées par *array_creation_expression* s (expressions de création de tableau) ou par des déclarations de variable locale ou de champ qui incluent un *array_initializer* (initialiseurs de tableau).

Lorsqu'une instance de tableau est créée, le rang et la longueur de chaque dimension sont établis, puis restent constants pour toute la durée de vie de l'instance. En d'autres termes, il n'est pas possible de modifier le rang d'une instance de tableau existante, pas plus qu'il n'est possible de redimensionner ses dimensions.

Une instance de tableau est toujours d'un type tableau. Le `System.Array` type est un type abstrait qui ne peut pas être instancié.

Les éléments des tableaux créés par *array_creation_expression* s sont toujours initialisés à leur valeur par défaut (valeurs par défaut).

Accès aux éléments des tableaux

Les éléments de tableau sont accessibles à l'aide d'expressions *element_access* (accès aux tableaux) du formulaire `A[I1, I2, ..., In]` , où `A` est une expression d'un type tableau et chaque `Ix` est une expression de type,,, `int` `uint` `long` `ulong` ou peut être implicitement convertie en un ou plusieurs de ces types. Le résultat d'un accès à un élément de tableau est une variable, à savoir l'élément de tableau sélectionné par les index.

Les éléments d'un tableau peuvent être énumérés à l'aide d'une `foreach` instruction (l'instruction foreach).

Membres de tableau

Chaque type de tableau hérite des membres déclarés par le `System.Array` type.

Covariance de tableau

Pour deux *reference_type* s `A` et `B` , si une conversion de référence implicite (conversions de référence implicites) ou une conversion de référence explicite (conversions de références explicites) existe entre et `A` `B` , la même conversion de référence existe également à partir du type de tableau `A[R]` vers le type de tableau `B[R]` , où `R` est une *rank_specifier* donnée (mais identique pour les deux types de tableau). Cette relation est

connue sous le nom de ***covariance de tableau***. La covariance de tableau en particulier signifie qu'une valeur d'un type de tableau `A[R]` peut en fait être une référence à une instance d'un type de tableau `B[R]` , à condition qu'il existe une conversion de référence implicite de `B` en `A` .

En raison de la covariance de tableau, les assignations aux éléments de tableaux de types référence incluent un contrôle au moment de l'exécution qui garantit que la valeur assignée à l'élément de tableau est en fait un type autorisé (assignation simple). Par exemple :

C#

```
class Test
{
    static void Fill(object[] array, int index, int count, object value) {
        for (int i = index; i < index + count; i++) array[i] = value;
    }

    static void Main() {
        string[] strings = new string[100];
        Fill(strings, 0, 100, "Undefined");
        Fill(strings, 0, 10, null);
        Fill(strings, 90, 10, 0);
    }
}
```

L'assignation à `array[i]` dans la `Fill` méthode comprend implicitement un contrôle au moment de l'exécution qui garantit que l'objet référencé par `value` est soit `null` ou une instance compatible avec le type d'élément réel de `array` . Dans `Main` , les deux premiers appels de se `Fill` déroulent correctement, mais le troisième appel entraîne la `System.ArrayTypeMismatchException` levée d'une exception lors de l'exécution de la première assignation à `array[i]` . L'exception se produit parce qu'un boxed `int` ne peut pas être stocké dans un `string` tableau.

La covariance de tableau ne s'étend pas aux tableaux de *Value_type* s. Par exemple, il n'existe aucune conversion qui autorise le traitement d'un en `int[]` tant que `object[]` .

Initialiseurs de tableau

Les initialiseurs de tableau peuvent être spécifiés dans les déclarations de champ (champs), les déclarations de variables locales (déclarations de variables locales) et les expressions de création de tableau (expressions de création de tableau) :

antlr

```
array_initializer
    : '{' variable_initializer_list? '}'
    | '{' variable_initializer_list ',' '}'
    ;

variable_initializer_list
    : variable_initializer (',' variable_initializer)*
    ;

variable_initializer
    : expression
    | array_initializer
    ;
```

Un initialiseur de tableau se compose d'une séquence d'initialiseurs de variable, placés entre des `{` jetons « » et « » `}` et séparés par des `,` jetons «». Chaque initialiseur de variable est une expression ou, dans le cas d'un tableau multidimensionnel, un initialiseur de tableau imbriqué.

Le contexte dans lequel un initialiseur de tableau est utilisé détermine le type du tableau en cours d'initialisation. Dans une expression de création de tableau, le type de tableau précède immédiatement l'initialiseur ou est déduit des expressions dans l'initialiseur de tableau. Dans une déclaration de champ ou de variable, le type de tableau est le type du champ ou de la variable déclaré (e). Lorsqu'un initialiseur de tableau est utilisé dans une déclaration de champ ou de variable, par exemple :

C#

```
int[] a = {0, 2, 4, 6, 8};
```

Il s'agit simplement d'un raccourci pour une expression de création de tableau équivalente :

C#

```
int[] a = new int[] {0, 2, 4, 6, 8};
```

Pour un tableau unidimensionnel, l'initialiseur de tableau doit se composer d'une séquence d'expressions qui sont compatibles avec le type d'élément du tableau. Les expressions initialisent les éléments de tableau par ordre de croissance, en commençant par l'élément à l'index zéro. Le nombre d'expressions dans l'initialiseur de tableau détermine la longueur de l'instance de tableau en cours de création. Par exemple, l'initialiseur de tableau ci-dessus crée une `int[]` instance de longueur 5, puis initialise l'instance avec les valeurs suivantes :

C#

```
a[0] = 0; a[1] = 2; a[2] = 4; a[3] = 6; a[4] = 8;
```

Pour un tableau multidimensionnel, l'initialiseur de tableau doit avoir autant de niveaux d'imbrication qu'il y a de dimensions dans le tableau. Le niveau d'imbrication le plus externe correspond à la dimension la plus à gauche et le niveau d'imbrication le plus profond correspond à la dimension la plus à droite. La longueur de chaque dimension du tableau est déterminée par le nombre d'éléments au niveau d'imbrication correspondant dans l'initialiseur de tableau. Pour chaque initialiseur de tableau imbriqué, le nombre d'éléments doit être le même que celui des autres initialiseurs de tableau au même niveau. L'exemple :

C#

```
int[,] b = {{0, 1}, {2, 3}, {4, 5}, {6, 7}, {8, 9}};
```

crée un tableau à deux dimensions avec une longueur de cinq pour la dimension la plus à gauche et une longueur de deux pour la dimension la plus à droite :

C#

```
int[,] b = new int[5, 2];
```

puis initialise l'instance de tableau avec les valeurs suivantes :

C#

```
b[0, 0] = 0; b[0, 1] = 1;
b[1, 0] = 2; b[1, 1] = 3;
b[2, 0] = 4; b[2, 1] = 5;
b[3, 0] = 6; b[3, 1] = 7;
b[4, 0] = 8; b[4, 1] = 9;
```

Si une dimension autre que la plus à droite est donnée avec la longueur zéro, les dimensions suivantes sont supposées avoir également une longueur égale à zéro. L'exemple :

C#

```
int[,] c = {};
```

crée un tableau à deux dimensions avec une longueur de zéro pour la dimension la plus à gauche et la dimension la plus à droite :

```
C#
```

```
int[,] c = new int[0, 0];
```

Quand une expression de création de tableau comprend à la fois des longueurs de dimensions explicites et un initialiseur de tableau, les longueurs doivent être des expressions constantes et le nombre d'éléments à chaque niveau d'imbrication doit correspondre à la longueur de dimension correspondante. Voici quelques exemples :

```
C#
```

```
int i = 3;
int[] x = new int[3] {0, 1, 2};        // OK
int[] y = new int[i] {0, 1, 2};        // Error, i not a constant
int[] z = new int[3] {0, 1, 2, 3};     // Error, length/initializer mismatch
```

Ici, l'initialiseur de `y` génère une erreur de compilation, car l'expression de longueur de dimension n'est pas une constante, et l'initialiseur de `z` génère une erreur au moment de la compilation, car la longueur et le nombre d'éléments dans l'initialiseur ne sont pas conformes.

Interfaces

Article • 13/05/2021

Une interface définit un contrat. Une classe ou un struct qui implémente une interface doit adhérer à son contrat. Une interface peut hériter de plusieurs interfaces de base, et une classe ou un struct peut implémenter plusieurs interfaces.

Les interfaces peuvent contenir des méthodes, des propriétés, des événements et des indexeurs. L'interface elle-même ne fournit pas d'implémentations pour les membres qu'elle définit. L'interface spécifie simplement les membres qui doivent être fournis par les classes ou les structs qui implémentent l'interface.

Déclarations d'interfaces

Une *interface_declaration* est une *type_declaration* (déclarations de type) qui déclare un nouveau type d'interface.

antlr

```
interface_declaration
    : attributes? interface_modifier* 'partial'? 'interface'
      identifier variant_type_parameter_list? interface_base?
      type_parameter_constraints_clause* interface_body ';'?
    ;
```

Un *interface_declaration* se compose d'un ensemble facultatif *d'attributs* (attributs), suivi d'un ensemble facultatif de *interface_modifier* s (modificateurs d'interface), suivi d'un `partial` modificateur facultatif, suivi du mot clé `interface` et d'un *identificateur* qui nomme l'interface, suivi d'une spécification facultative *variant_type_parameter_list* (listes de paramètres de type Variant), suivies d'une spécification facultative *interface_base* (interfaces de base), suivies d'une spécification facultative *type_parameter_constraints_clause* s (contraintes de paramètre de type), suivie d'une *interface_body* (corps d'interface), éventuellement suivie d'un point-

Modificateurs d'interface

Une *interface_declaration* peut éventuellement inclure une séquence de modificateurs d'interface :

antlr

```
interface_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | interface_modifier_unsafe
    ;
```

Il s'agit d'une erreur de compilation pour que le même modificateur apparaisse plusieurs fois dans une déclaration d'interface.

Le `new` modificateur est autorisé uniquement sur les interfaces définies dans une classe. Il spécifie que l'interface masque un membre hérité du même nom, comme décrit dans le nouveau modificateur.

Les `public` `protected` `internal` modificateurs,, et `private` contrôlent l'accessibilité de l'interface. En fonction du contexte dans lequel la déclaration d'interface se produit, seuls certains de ces modificateurs peuvent être autorisés (accessibilité déclarée).

Modificateur partiel

Le `partial` modificateur indique que ce *interface_declaration* est une déclaration de type partiel. Plusieurs déclarations d'interface partielles portant le même nom dans un espace de noms englobant ou une déclaration de type sont combinées pour former une déclaration d'interface, suivant les règles spécifiées dans les types partiels.

Listes de paramètres de type Variant

Les listes de paramètres de type Variant ne peuvent se produire que sur les types interface et Delegate. La différence par rapport aux *type_parameter_list* ordinaires est le *variance_annotation* facultatif sur chaque paramètre de type.

antlr

```
variant_type_parameter_list
    : '<' variant_type_parameters '>'
    ;

variant_type_parameters
    : attributes? variance_annotation? type_parameter
    | variant_type_parameters ',' attributes? variance_annotation?
type_parameter
    ;
```

```
variance_annotation
    : 'in'
    | 'out'
    ;
```

Si l'annotation de variance est `out` , le paramètre de type est dit *__covariant__ _. Si l'annotation de variance est `in` , le paramètre de type est dit comme *contravariant*. S'il n'y a pas d'annotation de variance, le paramètre de type est dit : _ *indifférent* *.

Dans l'exemple

C#

```
interface C<out X, in Y, Z>
{
  X M(Y y);
  Z P { get; set; }
}
```

`X` est covariant, `Y` est contravariant et `Z` est indifférent.

Sécurité des écarts

L'occurrence d'annotations de variance dans la liste de paramètres de type d'un type restreint les endroits où les types peuvent se produire dans la déclaration de type.

Un type `T` est une ***sortie non sécurisée*** si l'un des éléments suivants est présent :

- `T` est un paramètre de type contravariant
- `T` est un type de tableau avec un type d'élément de sortie non sécurisé
- `T` est une interface ou un type délégué `S<A1,...,Ak>` construit à partir d'un type générique, `S<X1,...,Xk>` où pour au moins l'un `Ai` des éléments suivants :
  - `Xi` est covariant ou indifférent et `Ai` est une sortie non sécurisée.
  - `Xi` est contravariant ou indifférent et `Ai` est sécurisé pour l'entrée.

Un type `T` est une ***entrée non sécurisée*** si l'un des éléments suivants est présent :

- `T` est un paramètre de type covariant
- `T` est un type de tableau avec un type d'élément d'entrée non sécurisé
- `T` est une interface ou un type délégué `S<A1,...,Ak>` construit à partir d'un type générique, `S<X1,...,Xk>` où pour au moins l'un `Ai` des éléments suivants :
  - `Xi` est covariant ou indifférent et `Ai` est une entrée non sécurisée.
  - `Xi` est contravariant ou invariant et `Ai` est une sortie non sécurisée.

Intuitivement, un type non sécurisé de sortie est interdit dans une position de sortie, et un type non sécurisé d'entrée est interdit dans une position d'entrée.

Un type est ***Safe-Safe*** _ *s'il ne s'agit pas d'une sortie non sécurisée et* _ *d'entrée-Safe* s'il n'est pas de type non sécurisé.

Conversion d'écart

L'objectif des annotations de variance est de fournir des conversions plus strictes (mais toujours de type sécurisé) en types d'interfaces et de délégués. À cette fin, les définitions des conversions implicites (conversionsimplicites) et explicites (conversions explicites) utilisent la notion de variance-convertibilité, qui est définie comme suit :

Un type `T<A1,...,An>` est convertible en un type `T<B1,...,Bn>` si `T` est une interface ou un type délégué déclaré avec les paramètres de type Variant `T<X1,...,Xn>` , et pour chaque paramètre de type variant l' `Xi` un des éléments suivants :

- `Xi` est covariant et une référence implicite ou une conversion d'identité existe `Ai` pour `Bi`
- `Xi` est contravariant et une référence implicite ou une conversion d'identité existe `Bi` pour `Ai`
- `Xi` est indifférent et une conversion d'identité existe de `Ai` en `Bi`

Interfaces de base

Une interface peut hériter de zéro, un ou plusieurs types d'interface, appelés ***interfaces de base explicites*** de l'interface. Lorsqu'une interface possède une ou plusieurs interfaces de base explicites, dans la déclaration de cette interface, l'identificateur d'interface est suivi d'un signe deux-points et d'une liste de types d'interfaces de base séparés par des virgules.

antlr

```
interface_base
    : ':' interface_type_list
    ;
```

Pour un type d'interface construit, les interfaces de base explicites sont formées en acceptant les déclarations d'interface de base explicites sur la déclaration de type générique et en substituant, pour chaque *type_parameter* dans la déclaration de l'interface de base, les *type_argument* correspondants du type construit.

Les interfaces de base explicites d'une interface doivent être au moins aussi accessibles que l'interface elle-même (contraintes d'accessibilité). Par exemple, il s'agit d'une erreur au moment de la compilation pour spécifier une `private` `internal` interface ou dans le *interface_base* d'une `public` interface.

Il s'agit d'une erreur au moment de la compilation pour qu'une interface hérite directement ou indirectement de lui-même.

Les ***interfaces de base*** d'une interface sont les interfaces de base explicites et leurs interfaces de base. En d'autres termes, l'ensemble des interfaces de base est la fermeture transitive complète des interfaces de base explicites, leurs interfaces de base explicites, et ainsi de suite. Une interface hérite de tous les membres de ses interfaces de base. Dans l'exemple

C#

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

interface IListBox: IControl
{
    void SetItems(string[] items);
}

interface IComboBox: ITextBox, IListBox {}
```

les interfaces de base de `IComboBox` sont `IControl` , `ITextBox` et `IListBox` .

En d'autres termes, l' `IComboBox` interface ci-dessus hérite des membres et, ainsi que de `SetText` `SetItems` `Paint` .

Chaque interface de base d'une interface doit être sécurisée en sortie (sécurité des variations). Une classe ou un struct qui implémente une interface implémente également implicitement toutes les interfaces de base de l'interface.

Corps de l'interface

La *interface_body* d'une interface définit les membres de l'interface.

antlr

```
interface_body
    : '{' interface_member_declaration* '}'
    ;
```

Membres d'interfaces

Les membres d'une interface sont les membres hérités des interfaces de base et les membres déclarés par l'interface elle-même.

antlr

```
interface_member_declaration
    : interface_method_declaration
    | interface_property_declaration
    | interface_event_declaration
    | interface_indexer_declaration
    ;
```

Une déclaration d'interface peut déclarer zéro ou plusieurs membres. Les membres d'une interface doivent être des méthodes, des propriétés, des événements ou des indexeurs. Une interface ne peut pas contenir des constantes, des champs, des opérateurs, des constructeurs d'instance, des destructeurs ou des types, ni une interface contenant des membres statiques de n'importe quel type.

Tous les membres d'interface ont implicitement un accès public. Il s'agit d'une erreur au moment de la compilation pour que les déclarations de membres d'interface incluent tous les modificateurs. En particulier, les membres d'interface ne peuvent pas être déclarés avec les modificateurs,,,,,, `abstract` `public` `protected` `internal` `private` `virtual` `override` ou `static` .

L'exemple

C#

```
public delegate void StringListEvent(IStringList sender);

public interface IStringList
{
    void Add(string s);
    int Count { get; }
    event StringListEvent Changed;
    string this[int index] { get; set; }
}
```

déclare une interface qui contient un chacun des genres possibles de membres : une méthode, une propriété, un événement et un indexeur.

Un *interface_declaration* crée un nouvel espace de déclaration (déclarations) et le *interface_member_declaration* s contenu immédiatement dans l' *interface_declaration* introduire de nouveaux membres dans cet espace de déclaration. Les règles suivantes s'appliquent à *interface_member_declaration* s :

- Le nom d'une méthode doit être différent des noms de toutes les propriétés et des événements déclarés dans la même interface. En outre, la signature (signatures et surcharge) d'une méthode doit être différente des signatures de toutes les autres méthodes déclarées dans la même interface, et deux méthodes déclarées dans la même interface peuvent ne pas avoir de signatures qui diffèrent uniquement par `ref` et `out` .
- Le nom d'une propriété ou d'un événement doit être différent des noms de tous les autres membres déclarés dans la même interface.
- La signature d'un indexeur doit se distinguer de tous les autres indexeurs déclarés dans la même interface.

Les membres hérités d'une interface ne font pas spécifiquement partie de l'espace de déclaration de l'interface. Par conséquent, une interface est autorisée à déclarer un membre avec le même nom ou la même signature qu'un membre hérité. Dans ce cas, le membre de l'interface dérivée est dit de masquer le membre de l'interface de base. Le masquage d'un membre hérité n'est pas considéré comme une erreur, mais il entraîne l'émission d'un avertissement par le compilateur. Pour supprimer l'avertissement, la déclaration du membre d'interface dérivé doit inclure un `new` modificateur pour indiquer que le membre dérivé est destiné à masquer le membre de base. Cette rubrique est traitée plus en détail dans Masquer par héritage.

Si un `new` modificateur est inclus dans une déclaration qui ne masque pas un membre hérité, un avertissement est émis à cet effet. Cet avertissement est supprimé en supprimant le `new` modificateur.

Notez que les membres de la classe ne `object` sont pas, strictement parlant, des membres de toute interface (membres d'interface). Toutefois, les membres de la classe `object` sont disponibles via la recherche de membres dans n'importe quel type d'interface (recherche de membre).

Méthodes d'interface

Les méthodes d'interface sont déclarées à l'aide de *interface_method_declaration* s :

```
```

antlr

```
interface_method_declaration
    : attributes? 'new'? return_type identifier type_parameter_list
      '(' formal_parameter_list? ')' type_parameter_constraints_clause* ';'
    ;
```

Les *attributs*, les *return_type*, les *identificateurs* et les *formal_parameter_list* d'une déclaration de méthode d'interface ont la même signification que ceux d'une déclaration de méthode dans une classe (méthodes). Une déclaration de méthode d'interface n'est pas autorisée à spécifier un corps de méthode, et la déclaration se termine donc toujours par un point-virgule.

Chaque type de paramètre formel d'une méthode d'interface doit être de type sécurisé d'entrée (la sécurité de l'écart) et le type de retour doit être soit `void` ou sécurisé de sortie. En outre, chaque contrainte de type classe, contrainte de type interface et contrainte de paramètre de type sur tout paramètre de type de la méthode doit être de type sécurisé.

Ces règles garantissent que toute utilisation covariante ou contravariant de l'interface reste de type sécurisé. Par exemple,

C#

```
interface I<out T> { void M<U>() where U : T; }
```

est non conforme, car l'utilisation de `T` en tant que contrainte de paramètre de type sur `U` n'est pas sécurisée pour l'entrée.

Si cette restriction n'est pas en place, il serait possible de violer la sécurité de type de la manière suivante :

C#

```
class B {}
class D : B{}
class E : B {}
class C : I<D> { public void M<U>() {...} }
...
I<B> b = new C();
b.M<E>();
```

Il s'agit en fait d'un appel à `C.M<E>` . Mais cet appel nécessite que `E` dérive de `D` , donc la sécurité de type serait violée ici.

Propriétés de l'interface

Les propriétés d'interface sont déclarées à l'aide de *interface_property_declaration* s :

```antlr
interface_property_declaration
    : attributes? 'new'? type identifier '{' interface_accessors '}'
    ;

interface_accessors
    : attributes? 'get' ';'
    | attributes? 'set' ';'
    | attributes? 'get' ';' attributes? 'set' ';'
    | attributes? 'set' ';' attributes? 'get' ';'
    ;
```

Les *attributs*, le *type* et l' *identificateur* d'une déclaration de propriété d'interface ont la même signification que ceux d'une déclaration de propriété dans une classe (Propriétés).

Les accesseurs d'une déclaration de propriété d'interface correspondent aux accesseurs d'une déclaration de propriété de classe (accesseurs), sauf que le corps de l'accesseur doit toujours être un point-virgule. Ainsi, les accesseurs indiquent simplement si la propriété est en lecture-écriture, en lecture seule ou en écriture seule.

Le type d'une propriété d'interface doit être sécurisé en sortie s'il existe un accesseur get et doit être de type sécurisé d'entrée en cas d'accesseur Set.

Événements d'interface

Les événements d'interface sont déclarés à l'aide de *interface_event_declaration* s :

```antlr
interface_event_declaration
    : attributes? 'new'? 'event' type identifier ';'
    ;
```

Les *attributs*, le *type* et l' *identificateur* d'une déclaration d'événement d'interface ont la même signification que ceux d'une déclaration d'événement dans une classe (événements).

Le type d'un événement d'interface doit être de type sécurisé.

Indexeurs d'interface

Les indexeurs d'interface sont déclarés à l'aide de *interface_indexer_declaration* s :

antlr

```
interface_indexer_declaration
    : attributes? 'new'? type 'this' '[' formal_parameter_list ']' '{'
interface_accessors '}'
    ;
```

Les *attributs*, le *type* et la *formal_parameter_list* d'une déclaration d'indexeur d'interface ont la même signification que ceux d'une déclaration d'indexeur dans une classe (indexeurs).

Les accesseurs d'une déclaration d'indexeur d'interface correspondent aux accesseurs d'une déclaration d'indexeur de classe (indexeurs), sauf que le corps de l'accesseur doit toujours être un point-virgule. Ainsi, les accesseurs indiquent simplement si l'indexeur est en lecture-écriture, en lecture seule ou en écriture seule.

Tous les types de paramètres formels d'un indexeur d'interface doivent être de type sécurisé. En outre, tous `out` les `ref` types de paramètres formels et doivent également être de type sécurisé de sortie. Notez que même `out` les paramètres doivent être de valeur sécurisée, en raison d'une limitation de la plateforme d'exécution sous-jacente.

Le type d'un indexeur d'interface doit être sécurisé en sortie s'il existe un accesseur get et doit être de type sécurisé d'entrée en cas d'accesseur Set.

Accès au membre d'interface

Les membres d'interface sont accessibles par le biais de l'accès aux membres (accès aux membres) et des expressions d'accès à l'indexeur (accès à l'indexeur) du formulaire `I.M` et `I[A]` , où `I` est un type d'interface, `M` est une méthode, une propriété ou un événement de ce type d'interface, et `A` est une liste d'arguments d'indexeur.

Pour les interfaces qui sont strictement à héritage simple (chaque interface dans la chaîne d'héritage possède exactement zéro ou une interface de base directe), les effets de la recherche de membre (recherche de membre), de l'appel de méthode (appels de méthode) et des règles d'accès à l'indexeur (accès à l'indexeur) sont exactement les mêmes que pour les classes et les structs : les membres plus dérivés masquent les membres moins dérivés portant le même nom ou la même signature. Toutefois, pour les interfaces d'héritage multiple, des ambiguïtés peuvent se produire lorsque deux ou plusieurs interfaces de base non liées déclarent des membres avec le même nom ou la

même signature. Cette section présente plusieurs exemples de ces situations. Dans tous les cas, les casts explicites peuvent être utilisés pour résoudre les ambiguïtés.

Dans l'exemple

C#

```
interface IList
{
    int Count { get; set; }
}

interface ICounter
{
    void Count(int i);
}

interface IListCounter: IList, ICounter {}

class C
{
    void Test(IListCounter x) {
        x.Count(1);                  // Error
        x.Count = 1;                 // Error
        ((IList)x).Count = 1;        // Ok, invokes IList.Count.set
        ((ICounter)x).Count(1);      // Ok, invokes ICounter.Count
    }
}
```

les deux premières instructions provoquent des erreurs au moment de la compilation, car la recherche de membre (recherche de membre) de `Count` dans `IListCounter` est ambiguë. Comme l'illustre l'exemple, l'ambiguïté est résolue en effectuant un cast `x` vers le type d'interface de base approprié. De tels casts n'ont aucun coût d'exécution : ils consistent simplement à afficher l'instance sous la forme d'un type moins dérivé au moment de la compilation.

Dans l'exemple

C#

```
interface IInteger
{
    void Add(int i);
}

interface IDouble
{
    void Add(double d);
}
```

```
interface INumber: IInteger, IDouble {}

class C
{
    void Test(INumber n) {
        n.Add(1);               // Invokes IInteger.Add
        n.Add(1.0);             // Only IDouble.Add is applicable
        ((IInteger)n).Add(1);   // Only IInteger.Add is a candidate
        ((IDouble)n).Add(1);    // Only IDouble.Add is a candidate
    }
}
```

l'appel `n.Add(1)` sélectionne `IInteger.Add` en appliquant les règles de résolution de surcharge de la résolution de surcharge. De même, l'appel `n.Add(1.0)` sélectionne `IDouble.Add` . Lorsque des casts explicites sont insérés, il n'y a qu'une seule méthode candidate, et donc aucune ambiguïté.

Dans l'exemple

C#

```
interface IBase
{
    void F(int i);
}

interface ILeft: IBase
{
    new void F(int i);
}

interface IRight: IBase
{
    void G();
}

interface IDerived: ILeft, IRight {}

class A
{
    void Test(IDerived d) {
        d.F(1);                 // Invokes ILeft.F
        ((IBase)d).F(1);        // Invokes IBase.F
        ((ILeft)d).F(1);        // Invokes ILeft.F
        ((IRight)d).F(1);       // Invokes IBase.F
    }
}
```

le `IBase.F` membre est masqué par le `ILeft.F` membre. L'appel `d.F(1)` est donc sélectionné `ILeft.F` , même `IBase.F` s'il semble ne pas être masqué dans le chemin

d'accès qui en résulte `IRight` .

La règle intuitive de masquage dans des interfaces à héritage multiple est simplement la suivante : si un membre est masqué dans un chemin d'accès, il est masqué dans tous les chemins d'accès. Dans la mesure où le chemin d'accès de `IDerived` à est `ILeft` `IBase` `IBase.F` masqué, le membre est également masqué dans le chemin d'accès de à `IDerived` `IRight` `IBase` .

Noms complets de membres d'interfaces

Un membre d'interface est parfois référencé par son ***nom*** complet. Le nom qualifié complet d'un membre d'interface se compose du nom de l'interface dans laquelle le membre est déclaré, suivi d'un point, suivi du nom du membre. Le nom qualifié complet d'un membre fait référence à l'interface dans laquelle le membre est déclaré. Par exemple, étant donné les déclarations

C#

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}
```

le nom qualifié complet de `Paint` est `IControl.Paint` et le nom qualifié complet de `SetText` est `ITextBox.SetText` .

Dans l'exemple ci-dessus, il n'est pas possible de faire référence à `Paint` As `ITextBox.Paint` .

Quand une interface fait partie d'un espace de noms, le nom qualifié complet d'un membre d'interface comprend le nom de l'espace de noms. Par exemple

C#

```
namespace System
{
    public interface ICloneable
    {
        object Clone();
```

```
    }
}
```

Ici, le nom qualifié complet de la `Clone` méthode est `System.ICloneable.Clone` .

Implémentations d'interfaces

Les interfaces peuvent être implémentées par des classes et des structs. Pour indiquer qu'une classe ou un struct implémente directement une interface, l'identificateur d'interface est inclus dans la liste de la classe de base de la classe ou du struct. Par exemple :

C#

```
interface ICloneable
{
    object Clone();
}

interface IComparable
{
    int CompareTo(object other);
}

class ListEntry: ICloneable, IComparable
{
    public object Clone() {...}
    public int CompareTo(object other) {...}
}
```

Une classe ou un struct qui implémente directement une interface implémente également directement toutes les interfaces de base de l'interface de manière implicite. Cela est vrai même si la classe ou structure ne répertorie pas explicitement toutes les interfaces de base dans la liste de classes de base. Par exemple :

C#

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

class TextBox: ITextBox
```

```
{
    public void Paint() {...}
    public void SetText(string text) {...}
}
```

Ici, la classe `TextBox` implémente à la fois `IControl` et `ITextBox` .

Quand une classe `C` implémente directement une interface, toutes les classes dérivées de C implémentent également l'interface implicitement. Les interfaces de base spécifiées dans une déclaration de classe peuvent être des types d'interfaces construits (types construits). Une interface de base ne peut pas être un paramètre de type seul, bien qu'il puisse impliquer les paramètres de type qui se trouvent dans la portée. Le code suivant illustre comment une classe peut implémenter et étendre les types construits :

C#

```
class C<U,V> {}

interface I1<V> {}

class D: C<string,int>, I1<string> {}

class E<T>: C<int,T>, I1<T> {}
```

Les interfaces de base d'une déclaration de classe générique doivent satisfaire à la règle d'unicité décrite dans unicité des interfaces implémentées.

Implémentations de membres d'interface explicites

Dans le cadre de l'implémentation d'interfaces, une classe ou un struct peut déclarer des ***implémentations de membres d'interface explicites***. Une implémentation de membre d'interface explicite est une méthode, une propriété, un événement ou une déclaration d'indexeur qui fait référence à un nom de membre d'interface qualifié complet. Par exemple

C#

```
interface IList<T>
{
    T[] GetElements();
}

interface IDictionary<K,V>
{
    V this[K key];
    void Add(K key, V value);
```

```
}

class List<T>: IList<T>, IDictionary<int,T>
{
    T[] IList<T>.GetElements() {...}
    T IDictionary<int,T>.this[int index] {...}
    void IDictionary<int,T>.Add(int index, T value) {...}
}
```

Ici `IDictionary<int,T>.this` et `IDictionary<int,T>.Add` sont des implémentations de membres d'interface explicites.

Dans certains cas, le nom d'un membre d'interface peut ne pas convenir à la classe d'implémentation, auquel cas le membre d'interface peut être implémenté à l'aide de l'implémentation de membre d'interface explicite. Une classe qui implémente une abstraction de fichier, par exemple, implémenterait probablement une `Close` fonction membre qui a pour effet de libérer la ressource de fichier, et d'implémenter la `Dispose` méthode de l' `IDisposable` interface à l'aide de l'implémentation de membre d'interface explicite :

C#

```
interface IDisposable
{
    void Dispose();
}

class MyFile: IDisposable
{
    void IDisposable.Dispose() {
        Close();
    }

    public void Close() {
        // Do what's necessary to close the file
        System.GC.SuppressFinalize(this);
    }
}
```

Il n'est pas possible d'accéder à une implémentation de membre d'interface explicite via son nom qualifié complet dans un appel de méthode, un accès à une propriété ou un accès à un indexeur. Une implémentation de membre d'interface explicite est accessible uniquement via une instance d'interface et est dans ce cas référencé simplement par son nom de membre.

Il s'agit d'une erreur de compilation pour qu'une implémentation de membre d'interface explicite inclue des modificateurs d'accès, et il s'agit d'une erreur de compilation pour

inclure les modificateurs `abstract` , `virtual` , `override` ou `static` .

Les implémentations de membres d'interface explicites ont des caractéristiques d'accessibilité différentes de celles des autres membres. Étant donné que les implémentations de membres d'interface explicites ne sont jamais accessibles par leur nom qualifié complet dans un appel de méthode ou un accès à une propriété, ils sont dans un sens privé. Toutefois, étant donné qu'elles sont accessibles par le biais d'une instance d'interface, elles sont dans un sens aussi publiques.

Les implémentations de membres d'interface explicites remplissent deux fonctions principales :

- Étant donné que les implémentations de membres d'interface explicites ne sont pas accessibles via des instances de classe ou de struct, elles permettent d'exclure des implémentations d'interface de l'interface publique d'une classe ou d'un struct. Cela s'avère particulièrement utile quand une classe ou un struct implémente une interface interne qui n'a aucun intérêt pour un consommateur de cette classe ou structure.
- Les implémentations de membres d'interface explicites permettent de lever l'ambiguïté des membres d'interface avec la même signature. Sans implémentations de membres d'interface explicites, il serait impossible pour une classe ou un struct d'avoir différentes implémentations de membres d'interface avec la même signature et le même type de retour, car il serait impossible pour une classe ou un struct d'avoir une implémentation au niveau de tous les membres d'interface ayant la même signature, mais avec des types de retour différents.

Pour qu'une implémentation de membre d'interface explicite soit valide, la classe ou le struct doit nommer une interface dans sa liste de classes de base qui contient un membre dont les types de nom complet, de type et de paramètre correspondent exactement à ceux de l'implémentation de membre d'interface explicite. Par conséquent, dans la classe suivante

C#

```
class Shape: ICloneable
{
    object ICloneable.Clone() {...}
    int IComparable.CompareTo(object other) {...}    // invalid
}
```

la déclaration de `IComparable.CompareTo` génère une erreur de compilation, car `IComparable` n'est pas répertorié dans la liste de classes de base de `Shape` et n'est pas une interface de base de `ICloneable` . De même, dans les déclarations

C#

```
class Shape: ICloneable
{
    object ICloneable.Clone() {...}
}

class Ellipse: Shape
{
    object ICloneable.Clone() {...}    // invalid
}
```

la déclaration de `ICloneable.Clone` dans `Ellipse` génère une erreur de compilation, car `ICloneable` n'est pas explicitement répertorié dans la liste de classes de base de `Ellipse` .

Le nom qualifié complet d'un membre d'interface doit faire référence à l'interface dans laquelle le membre a été déclaré. Par conséquent, dans les déclarations

C#

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

class TextBox: ITextBox
{
    void IControl.Paint() {...}
    void ITextBox.SetText(string text) {...}
}
```

l'implémentation de membre d'interface explicite de `Paint` doit être écrite sous la forme `IControl.Paint` .

Unicité des interfaces implémentées

Les interfaces implémentées par une déclaration de type générique doivent rester uniques pour tous les types construits possibles. Sans cette règle, il est impossible de déterminer la méthode correcte à appeler pour certains types construits. Par exemple,

supposons qu'une déclaration de classe générique était autorisée à être écrite comme suit :

C#

```
interface I<T>
{
    void F();
}

class X<U,V>: I<U>, I<V>                    // Error: I<U> and I<V> conflict
{
    void I<U>.F() {...}
    void I<V>.F() {...}
}
```

Cela était autorisé, il serait impossible de déterminer le code à exécuter dans le cas suivant :

C#

```
I<int> x = new X<int,int>();
x.F();
```

Pour déterminer si la liste d'interfaces d'une déclaration de type générique est valide, les étapes suivantes sont effectuées :

- Indiquez `L` la liste des interfaces directement spécifiées dans une déclaration générique de classe, de struct ou d'interface `C` .
- Ajoutez à `L` toutes les interfaces de base des interfaces déjà présentes dans `L` .
- Supprimez les doublons de `L` .
- Si un type construit peut être créé à partir de, `C` après que les arguments de type ont été remplacés par `L` , deux interfaces dans sont `L` identiques, alors la déclaration de `C` n'est pas valide. Les déclarations de contrainte ne sont pas prises en compte lors de la détermination de tous les types construits possibles.

Dans la déclaration de classe `X` ci-dessus, la liste `L` d'interfaces se compose de `I<U>` et `I<V>` . La déclaration n'est pas valide, car tout type construit avec `U` et `V` étant du même type entraînerait la création de ces deux interfaces de types identiques.

Il est possible d'unifier les interfaces spécifiées à différents niveaux d'héritage :

C#

```
interface I<T>
{
    void F();
}

class Base<U>: I<U>
{
    void I<U>.F() {...}
}

class Derived<U,V>: Base<U>, I<V>    // Ok
{
    void I<V>.F() {...}
}
```

Ce code est valide même si `Derived<U,V>` implémente à la fois `I<U>` et `I<V>` . Code

C#

```
I<int> x = new Derived<int,int>();
x.F();
```

appelle la méthode dans `Derived` , car `Derived<int,int>` réimplémente efficacement `I<int>` (réimplémentationde l'interface).

Implémentation de méthodes génériques

Quand une méthode générique implémente implicitement une méthode d'interface, les contraintes fournies pour chaque paramètre de type de méthode doivent être équivalentes dans les deux déclarations (une fois tous les paramètres de type d'interface remplacés par les arguments de type appropriés), où les paramètres de type de méthode sont identifiés par des positions ordinales, de gauche à droite.

Quand une méthode générique implémente explicitement une méthode d'interface, toutefois, aucune contrainte n'est autorisée sur la méthode d'implémentation. Au lieu de cela, les contraintes sont héritées de la méthode d'interface

C#

```
interface I<A,B,C>
{
    void F<T>(T t) where T: A;
    void G<T>(T t) where T: B;
    void H<T>(T t) where T: C;
}
```

```
class C: I<object,C,string>
{
    public void F<T>(T t) {...}                    // Ok
    public void G<T>(T t) where T: C {...}         // Ok
    public void H<T>(T t) where T: string {...}    // Error
}
```

La méthode `C.F<T>` implémente implicitement `I<object,C,string>.F<T>` . Dans ce cas, `C.F<T>` n'est pas requis (ni autorisé) pour spécifier la contrainte, `T:object` car `object` est une contrainte implicite sur tous les paramètres de type. La méthode `C.G<T>` implémente implicitement, `I<object,C,string>.G<T>` car les contraintes correspondent à celles de l'interface, après le remplacement des paramètres de type d'interface par les arguments de type correspondants. La contrainte pour la méthode `C.H<T>` est une erreur, car les types sealed (`string` dans ce cas) ne peuvent pas être utilisés en tant que contraintes. L'omission de la contrainte serait également une erreur, car les contraintes des implémentations de méthodes d'interface implicites sont obligatoires pour correspondre à. Par conséquent, il est impossible d'implémenter implicitement `I<object,C,string>.H<T>` . Cette méthode d'interface ne peut être implémentée qu'à l'aide d'une implémentation de membre d'interface explicite :

C#

```
class C: I<object,C,string>
{
    ...

    public void H<U>(U u) where U: class {...}

    void I<object,C,string>.H<T>(T t) {
        string s = t;    // Ok
        H<T>(t);
    }
}
```

Dans cet exemple, l'implémentation de membre d'interface explicite appelle une méthode publique qui a des contraintes strictement plus faibles. Notez que l'assignation de `t` à `s` est valide `T` , car hérite d'une contrainte de `T:string` , même si cette contrainte n'est pas exprimable dans le code source.

Mappage d'interface

Une classe ou un struct doit fournir des implémentations de tous les membres des interfaces répertoriées dans la liste de la classe de base de la classe ou du struct. Le

processus de localisation des implémentations de membres d'interface dans une classe ou un struct d'implémentation est appelé ***mappage d'interface***.

Le mappage d'interface pour une classe ou un struct `C` localise une implémentation pour chaque membre de chaque interface spécifiée dans la liste de classes de base de `C` . L'implémentation d'un membre d'interface particulier `I.M` , où `I` est l'interface dans laquelle le membre `M` est déclaré, est déterminée en examinant chaque classe ou struct, en `S` commençant par et en `C` répétant pour chaque classe de base consécutive de `C` , jusqu'à ce qu'une correspondance soit trouvée :

- Si `S` contient une déclaration d'une implémentation de membre d'interface explicite qui correspond à `I` et `M` , ce membre est l'implémentation de `I.M` .
- Sinon, si `S` contient une déclaration d'un membre public non statique qui correspond `M` , ce membre est l'implémentation de `I.M` . Si plusieurs membres correspondent, il n'est pas spécifié quel membre est l'implémentation de `I.M` . Cette situation peut se produire uniquement si `S` est un type construit où les deux membres comme déclarés dans le type générique ont des signatures différentes, mais que les arguments de type rendent leurs signatures identiques.

Une erreur de compilation se produit si les implémentations ne peuvent pas être localisées pour tous les membres de toutes les interfaces spécifiées dans la liste de classes de base de `C` . Notez que les membres d'une interface incluent les membres qui sont hérités des interfaces de base.

Dans le cadre du mappage d'interface, un membre de classe `A` correspond à un membre d'interface `B` lorsque :

- `A` et `B` sont des méthodes, et le nom, le type et les listes de paramètres formels de `A` et `B` sont identiques.
- `A` et `B` sont des propriétés, le nom et le type de `A` et `B` sont identiques et `A` ont les mêmes accesseurs que `B` (`A` est autorisé à avoir des accesseurs supplémentaires s'il ne s'agit pas d'une implémentation de membre d'interface explicite).
- `A` et `B` sont des événements, et le nom et le type de `A` et `B` sont identiques.
- `A` et `B` sont des indexeurs, les listes de paramètres de type et formels de `A` et `B` sont identiques, et `A` ont les mêmes accesseurs que `B` (`A` est autorisé à avoir des accesseurs supplémentaires s'il ne s'agit pas d'une implémentation de membre d'interface explicite).

Les implications notables de l'algorithme de mappage d'interface sont les suivantes :

- Les implémentations de membres d'interface explicites sont prioritaires sur les autres membres de la même classe ou du même struct lors de la détermination du membre de classe ou de struct qui implémente un membre d'interface.
- Ni les membres non publics ni les membres statiques ne participent au mappage d'interface.

Dans l'exemple

C#

```
interface ICloneable
{
    object Clone();
}

class C: ICloneable
{
    object ICloneable.Clone() {...}
    public object Clone() {...}
}
```

le `ICloneable.Clone` membre de `C` devient l'implémentation de `Clone` dans `ICloneable` , car les implémentations de membres d'interface explicites ont la priorité sur les autres membres.

Si une classe ou un struct implémente au moins deux interfaces contenant un membre avec le même nom, le même type et les mêmes types de paramètres, il est possible de mapper chacun de ces membres d'interface sur un membre de classe ou de struct unique. Par exemple

C#

```
interface IControl
{
    void Paint();
}

interface IForm
{
    void Paint();
}

class Page: IControl, IForm
{
    public void Paint() {...}
}
```

Ici, les `Paint` méthodes de `IControl` et `IForm` sont mappées sur la `Paint` méthode dans `Page` . Il est bien sûr également possible d'avoir des implémentations de membres d'interface explicites distinctes pour les deux méthodes.

Si une classe ou un struct implémente une interface qui contient des membres masqués, certains membres doivent nécessairement être implémentés via des implémentations de membres d'interface explicites. Par exemple

C#

```
interface IBase
{
    int P { get; }
}

interface IDerived: IBase
{
    new int P();
}
```

Une implémentation de cette interface requiert au moins une implémentation de membre d'interface explicite et prend l'une des formes suivantes :

C#

```
class C: IDerived
{
    int IBase.P { get {...} }
    int IDerived.P() {...}
}

class C: IDerived
{
    public int P { get {...} }
    int IDerived.P() {...}
}

class C: IDerived
{
    int IBase.P { get {...} }
    public int P() {...}
}
```

Lorsqu'une classe implémente plusieurs interfaces qui ont la même interface de base, il ne peut y avoir qu'une seule implémentation de l'interface de base. Dans l'exemple

C#

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

interface IListBox: IControl
{
    void SetItems(string[] items);
}

class ComboBox: IControl, ITextBox, IListBox
{
    void IControl.Paint() {...}
    void ITextBox.SetText(string text) {...}
    void IListBox.SetItems(string[] items) {...}
}
```

Il n'est pas possible d'avoir des implémentations distinctes pour le `IControl` nommé dans la liste de classes de base, le `IControl` hérité par `ITextBox` et le `IControl` hérité par `IListBox` . En effet, il n'existe aucune notion d'identité distincte pour ces interfaces. Au lieu de cela, les implémentations de `ITextBox` et `IListBox` partagent la même implémentation de `IControl` , et `ComboBox` sont simplement considérées comme implémentant trois interfaces, `IControl` , `ITextBox` et `IListBox` .

Les membres d'une classe de base participent au mappage d'interface. Dans l'exemple

C#

```
interface Interface1
{
    void F();
}

class Class1
{
    public void F() {}
    public void G() {}
}

class Class2: Class1, Interface1
{
    new public void G() {}
}
```

la méthode `F` dans `Class1` est utilisée dans l' `Class2` implémentation de `Interface1` .

Héritage de l'implémentation d'interface

Une classe hérite de toutes les implémentations d'interface fournies par ses classes de base.

Sans la ***réimplémentation*** explicite d'une interface, une classe dérivée ne peut en aucune façon modifier les mappages d'interface qu'elle hérite de ses classes de base. Par exemple, dans les déclarations

C#

```
interface IControl
{
    void Paint();
}

class Control: IControl
{
    public void Paint() {...}
}

class TextBox: Control
{
    new public void Paint() {...}
}
```

la `Paint` méthode dans `TextBox` masque la `Paint` méthode dans `Control` , mais elle ne modifie pas le mappage de `Control.Paint` sur `IControl.Paint` , et les appels à `Paint` via des instances de classe et d'interface ont les effets suivants

C#

```
Control c = new Control();
TextBox t = new TextBox();
IControl ic = c;
IControl it = t;
c.Paint();             // invokes Control.Paint();
t.Paint();             // invokes TextBox.Paint();
ic.Paint();            // invokes Control.Paint();
it.Paint();            // invokes Control.Paint();
```

Toutefois, lorsqu'une méthode d'interface est mappée sur une méthode virtuelle dans une classe, il est possible pour les classes dérivées de substituer la méthode virtuelle et

de modifier l'implémentation de l'interface. Par exemple, la réécriture des déclarations ci-dessus à

C#

```
interface IControl
{
    void Paint();
}

class Control: IControl
{
    public virtual void Paint() {...}
}

class TextBox: Control
{
    public override void Paint() {...}
}
```

les effets suivants seront observés

C#

```
Control c = new Control();
TextBox t = new TextBox();
IControl ic = c;
IControl it = t;
c.Paint();            // invokes Control.Paint();
t.Paint();            // invokes TextBox.Paint();
ic.Paint();           // invokes Control.Paint();
it.Paint();           // invokes TextBox.Paint();
```

Étant donné que les implémentations de membres d'interface explicites ne peuvent pas être déclarées comme virtuelles, il n'est pas possible de se substituer à une implémentation de membre d'interface explicite. Toutefois, il est parfaitement valide pour une implémentation de membre d'interface explicite d'appeler une autre méthode, et cette autre méthode peut être déclarée virtuelle pour permettre aux classes dérivées de la substituer. Par exemple

C#

```
interface IControl
{
    void Paint();
}

class Control: IControl
{
```

```
    void IControl.Paint() { PaintControl(); }
    protected virtual void PaintControl() {...}
}

class TextBox: Control
{
    protected override void PaintControl() {...}
}
```

Ici, les classes dérivées de `Control` peuvent spécialiser l'implémentation de `IControl.Paint` en substituant la `PaintControl` méthode.

Réimplémentation de l'interface

Une classe qui hérite d'une implémentation d'interface est autorisée à ***réimplémenter*** l'interface en l'incluant dans la liste de classes de base.

Une nouvelle implémentation d'une interface suit exactement les mêmes règles de mappage d'interface qu'une implémentation initiale d'une interface. Ainsi, le mappage d'interface hérité n'a aucun effet sur le mappage d'interface établi pour la réimplémentation de l'interface. Par exemple, dans les déclarations

C#

```
interface IControl
{
    void Paint();
}

class Control: IControl
{
    void IControl.Paint() {...}
}

class MyControl: Control, IControl
{
    public void Paint() {}
}
```

le fait que `Control` mappe `IControl.Paint` sur `Control.IControl.Paint` n'affecte pas la réimplémentation dans `MyControl` , qui est mappée `IControl.Paint` sur `MyControl.Paint` .

Les déclarations de membres publics héritées et les déclarations de membres d'interface explicites héritées participent au processus de mappage d'interface pour les interfaces réimplémentées. Par exemple

C#

```
interface IMethods
{
    void F();
    void G();
    void H();
    void I();
}

class Base: IMethods
{
    void IMethods.F() {}
    void IMethods.G() {}
    public void H() {}
    public void I() {}
}

class Derived: Base, IMethods
{
    public void F() {}
    void IMethods.H() {}
}
```

Ici, l'implémentation de `IMethods` dans `Derived` mappe les méthodes d'interface sur `Derived.F` , `Base.IMethods.G` , `Derived.IMethods.H` et `Base.I` .

Lorsqu'une classe implémente une interface, elle implémente implicitement également toutes les interfaces de base de cette interface. De même, une nouvelle implémentation d'une interface est implicitement une nouvelle implémentation de toutes les interfaces de base de l'interface. Par exemple

C#

```
interface IBase
{
    void F();
}

interface IDerived: IBase
{
    void G();
}

class C: IDerived
{
    void IBase.F() {...}
    void IDerived.G() {...}
}

class D: C, IDerived
```

```
{
    public void F() {...}
    public void G() {...}
}
```

Ici, la nouvelle implémentation de `IDerived` réimplémente également, en `IBase` mappant `IBase.F` `D.F` .

Classes et interfaces abstraites

À l'instar d'une classe non abstraite, une classe abstraite doit fournir des implémentations de tous les membres des interfaces répertoriées dans la liste de la classe de base de la classe. Toutefois, une classe abstraite est autorisée à mapper des méthodes d'interface sur des méthodes abstraites. Par exemple

C#

```
interface IMethods
{
    void F();
    void G();
}

abstract class C: IMethods
{
    public abstract void F();
    public abstract void G();
}
```

Ici, l'implémentation de `IMethods` Maps `F` et `G` sur les méthodes abstraites, qui doit être substituée dans les classes non abstraites qui dérivent de `C` .

Notez que les implémentations de membres d'interface explicites ne peuvent pas être abstraites, mais que les implémentations de membres d'interface explicites sont bien entendu autorisées à appeler des méthodes abstraites. Par exemple

C#

```
interface IMethods
{
    void F();
    void G();
}

abstract class C: IMethods
{
    void IMethods.F() { FF(); }
```

```
    void IMethods.G() { GG(); }
    protected abstract void FF();
    protected abstract void GG();
}
```

Ici, les classes non abstraites qui dérivent de sont `C` nécessaires pour substituer `FF` et, ce qui `GG` fournit l'implémentation réelle de `IMethods` .

Énumérations

Article • 13/05/2021

Un ***type enum*** est un type valeur distinct (types valeur) qui déclare un ensemble de constantes nommées.

L'exemple

```C#
enum Color
{
    Red,
    Green,
    Blue
}
```

déclare un type Enum nommé `Color` avec les membres `Red` , `Green` et `Blue` .

Déclarations d'énumérations

Une déclaration enum déclare un nouveau type enum. Une déclaration enum commence par le mot clé `enum` et définit le nom, l'accessibilité, le type sous-jacent et les membres de l'enum.

```antlr
enum_declaration
    : attributes? enum_modifier* 'enum' identifier enum_base? enum_body ';'?
    ;

enum_base
    : ':' integral_type
    ;

enum_body
    : '{' enum_member_declarations? '}'
    | '{' enum_member_declarations ',' '}'
    ;
```

Chaque type enum a un type intégral correspondant appelé le ***type sous-jacent*** du type enum. Ce type sous-jacent doit être en mesure de représenter toutes les valeurs d'énumérateur définies dans l'énumération. Une déclaration enum peut déclarer explicitement un type sous-jacent de,,,, `byte` `sbyte` `short` `ushort` `int` `uint` `long` ou

`ulong` . Notez que `char` ne peut pas être utilisé en tant que type sous-jacent. Une déclaration enum qui ne déclare pas explicitement un type sous-jacent a un type sous-jacent de `int` .

L'exemple

C#

```
enum Color: long
{
    Red,
    Green,
    Blue
}
```

déclare une énumération avec un type sous-jacent de `long` . Un développeur peut choisir d'utiliser un type sous-jacent de `long` , comme dans l'exemple, pour permettre l'utilisation de valeurs qui se trouvent dans la plage de `long` , mais pas dans la plage de `int` , ou pour conserver cette option à l'avenir.

Modificateurs d'énumération

Une *enum_declaration* peut éventuellement inclure une séquence de modificateurs enum :

antlr

```
enum_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    ;
```

Il s'agit d'une erreur au moment de la compilation pour que le même modificateur apparaisse plusieurs fois dans une déclaration Enum.

Les modificateurs d'une déclaration enum ont la même signification que ceux d'une déclaration de classe (modificateurs de classe). Notez, toutefois, que les `abstract` `sealed` modificateurs et ne sont pas autorisés dans une déclaration Enum. Les enums ne peuvent pas être abstraits et n'autorisent pas la dérivation.

Membres d’énumération

Le corps d’une déclaration de type enum définit zéro, un ou plusieurs membres enum, qui sont les constantes nommées du type enum. Deux membres enum ne peuvent pas avoir le même nom.

antlr

```
enum_member_declarations
    : enum_member_declaration (',' enum_member_declaration)*
    ;

enum_member_declaration
    : attributes? identifier ('=' constant_expression)?
    ;
```

Chaque membre enum a une valeur de constante associée. Le type de cette valeur est le type sous-jacent pour l’enum conteneur. La valeur de constante pour chaque membre enum doit être comprise dans la plage du type sous-jacent pour l’enum. L’exemple

C#

```
enum Color: uint
{
    Red = -1,
    Green = -2,
    Blue = -3
}
```

génère une erreur au moment de la compilation, car les `-1` valeurs `-2` de constante, et ne `-3` sont pas dans la plage du type intégral sous-jacent `uint` .

Plusieurs membres enum peuvent partager la même valeur associée. L’exemple

C#

```
enum Color
{
    Red,
    Green,
    Blue,

    Max = Blue
}
```

affiche une énumération dans laquelle deux membres enum-- `Blue` et `Max` --ont la même valeur associée.

La valeur associée d'un membre enum est assignée implicitement ou explicitement. Si la déclaration du membre enum a un initialiseur *constant_expression* , la valeur de cette expression constante, implicitement convertie dans le type sous-jacent de l'enum, est la valeur associée du membre Enum. Si la déclaration du membre enum n'a pas d'initialiseur, sa valeur associée est définie implicitement, comme suit :

- Si le membre enum est le premier membre enum déclaré dans le type enum, sa valeur associée est zéro.
- Dans le cas contraire, la valeur associée du membre enum est obtenue en accroissant la valeur associée du membre enum qui précède textuellement. Cette valeur augmentée doit être comprise dans la plage de valeurs qui peuvent être représentées par le type sous-jacent ; sinon, une erreur de compilation se produit.

L'exemple

C#

```
using System;

enum Color
{
    Red,
    Green = 10,
    Blue
}

class Test
{
    static void Main() {
        Console.WriteLine(StringFromColor(Color.Red));
        Console.WriteLine(StringFromColor(Color.Green));
        Console.WriteLine(StringFromColor(Color.Blue));
    }

    static string StringFromColor(Color c) {
        switch (c) {
            case Color.Red:
                return String.Format("Red = {0}", (int) c);

            case Color.Green:
                return String.Format("Green = {0}", (int) c);

            case Color.Blue:
                return String.Format("Blue = {0}", (int) c);

            default:
                return "Invalid color";
```

```
        }
    }
}
```

imprime les noms des membres enum et leurs valeurs associées. La sortie est la suivante :

Console

```
Red = 0
Green = 10
Blue = 11
```

pour les raisons suivantes :

- `Red` la valeur zéro est assignée automatiquement au membre enum (car il n'a pas d'initialiseur et est le premier membre enum);
- la `Green` valeur est explicitement donnée au membre enum `10` ;
- et le membre enum reçoit `Blue` automatiquement une valeur supérieure à celle du membre qui le précède textuellement.

La valeur associée d'un membre enum ne peut pas, directement ou indirectement, utiliser la valeur de son propre membre enum associé. Outre cette restriction de circularité, les initialiseurs de membres enum peuvent librement faire référence à d'autres initialiseurs de membres enum, indépendamment de leur position textuelle. Dans un initialiseur de membre enum, les valeurs d'autres membres enum sont toujours traitées comme ayant le type de leur type sous-jacent, de sorte que les casts ne sont pas nécessaires pour faire référence à d'autres membres enum.

L'exemple

C#

```
enum Circular
{
    A = B,
    B
}
```

génère une erreur au moment de la compilation, car les déclarations de `A` et `B` sont circulaires. `A` dépend `B` explicitement de et `B` dépend `A` implicitement de.

Les membres enum sont nommés et étendus d'une manière exactement analogue aux champs dans les classes. La portée d'un membre enum est le corps de son type enum

contenant. Au sein de cette étendue, les membres enum peuvent être référencés par leur nom simple. À partir de tout autre code, le nom d'un membre enum doit être qualifié avec le nom de son type enum. Les membres enum n'ont pas d'accessibilité déclarée : un membre enum est accessible si son type enum conteneur est accessible.

Type System.Enum

Le type `System.Enum` est la classe de base abstraite de tous les types ENUM (il est distinct et différent du type sous-jacent du type enum), et les membres hérités de `System.Enum` sont disponibles dans n'importe quel type enum. Une conversion boxing (conversions boxing) existe à partir de n'importe quel type enum en `System.Enum` , et une conversion unboxing (conversions unboxing) existe depuis `System.Enum` vers un type enum.

Notez qu' `System.Enum` il ne s'agit pas d'un *enum_type*. Au lieu de cela, il s'agit d'un *class_type* duquel tous les *enum_type* sont dérivés. Le type `System.Enum` hérite du type `System.ValueType` (type System. ValueType) qui, à son tour, hérite du type `object` . Au moment de l'exécution, une valeur de type `System.Enum` peut être `null` ou une référence à une valeur boxed de tout type enum.

Valeurs et opérations des énumérations

Chaque type enum définit un type distinct ; une conversion d'énumération explicite (conversions d'énumération explicites) est requise pour la conversion entre un type enum et un type intégral, ou entre deux types ENUM. L'ensemble de valeurs qu'un type enum peut utiliser n'est pas limité par ses membres enum. En particulier, toute valeur du type sous-jacent d'une énumération peut être convertie en type enum, et est une valeur valide distincte de ce type enum.

Les membres enum ont le type de leur type enum conteneur (sauf dans d'autres initialiseurs de membres enum : consultez enum members). La valeur d'un membre enum déclaré dans le type enum `E` avec la valeur associée `v` est `(E)v` .

Les opérateurs suivants peuvent être utilisés sur des valeurs de types enum : `==` , `!=` ,,, `<` `>` `<=` , `>=` (opérateurs de comparaison d'énumération), binaire `+` (opérateur d'addition), binaire `-` (opérateur de soustraction), `^` , `&` , `|` (opérateurs logiques d'énumération), `~` (opérateur de complément au niveau du bit), `++` et `--` (opérateurs suffixés d'incrémentation et de décrémentation).

Chaque type enum dérive automatiquement de la classe `System.Enum` (qui dérive à son tour de `System.ValueType` et `object`). Ainsi, les méthodes et les propriétés héritées de cette classe peuvent être utilisées sur les valeurs d'un type enum.

Délégués

Article • 13/05/2021

Les délégués permettent aux scénarios que d'autres langages, tels que C++, Pascal et Modula, aient résolu les pointeurs de fonction. Contrairement aux pointeurs de fonction C++, toutefois, les délégués sont entièrement orientés objet et contrairement aux pointeurs C++ vers les fonctions membres, les délégués encapsulent à la fois une instance d'objet et une méthode.

Une déclaration Delegate définit une classe dérivée de la classe `System.Delegate` . Une instance de délégué encapsule une liste d'appel, qui est une liste d'une ou plusieurs méthodes, chacune d'elles étant désignée sous le terme d'entité pouvant être appelée. Pour les méthodes d'instance, une entité pouvant être appelée se compose d'une instance et d'une méthode sur cette instance. Pour les méthodes statiques, une entité pouvant être appelée se compose simplement d'une méthode. L'appel d'une instance de délégué avec un jeu d'arguments approprié entraîne l'appel de chacune des entités pouvant être appelées par le délégué avec le jeu d'arguments donné.

Une propriété intéressante et utile d'une instance de délégué est qu'elle ne connaît pas ou ne tient pas compte des classes des méthodes qu'elle encapsule ; Il est important que ces méthodes soient compatibles (déclarations déléguées) avec le type du délégué. Les délégués sont alors parfaitement adaptés à l'appel « anonyme ».

Déclarations des délégués

Un *delegate_declaration* est un *type_declaration* (déclarations de type) qui déclare un nouveau type délégué.

antlr

```
delegate_declaration
    : attributes? delegate_modifier* 'delegate' return_type
      identifier variant_type_parameter_list?
      '(' formal_parameter_list? ')' type_parameter_constraints_clause* ';'
    ;

delegate_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | delegate_modifier_unsafe
    ;
```

Il s'agit d'une erreur au moment de la compilation pour que le même modificateur apparaisse plusieurs fois dans une déclaration Delegate.

Le `new` modificateur est autorisé uniquement sur les délégués déclarés dans un autre type, auquel cas il spécifie qu'un tel délégué masque un membre hérité du même nom, comme décrit dans le nouveau modificateur.

Les `public` `protected` `internal` modificateurs,, et `private` contrôlent l'accessibilité du type délégué. En fonction du contexte dans lequel la déclaration du délégué se produit, certains de ces modificateurs peuvent ne pas être autorisés (accessibilité déclarée).

Le nom de type du délégué est *identifier*.

Le *formal_parameter_list* facultatif spécifie les paramètres du délégué et *return_type* indique le type de retour du délégué.

Le *variant_type_parameter_list* facultatif (listes de paramètres de type Variant) spécifie les paramètres de type pour le délégué lui-même.

Le type de retour d'un type délégué doit être `void` , ou sécurisé de sortie (la sécurité de l'écart).

Tous les types de paramètres formels d'un type délégué doivent être de type sécurisé d'entrée. En outre, les `out` `ref` types de paramètres ou doivent également être de type sécurisé de sortie. Notez que même `out` les paramètres doivent être de valeur sécurisée, en raison d'une limitation de la plateforme d'exécution sous-jacente.

Les types délégués en C# sont équivalents au nom, et non structurellement équivalents. Plus précisément, deux types délégués différents ayant les mêmes listes de paramètres et type de retour sont considérés comme des types délégués différents. Toutefois, les instances de deux types délégués distincts mais structurellement équivalents peuvent être considérées comme égales (opérateurs d'égalité de délégué).

Par exemple :

C#

```
delegate int D1(int i, double d);

class A
{
    public static int M1(int a, double b) {...}
}

class B
```

```
{
    delegate int D2(int c, double d);
    public static int M1(int f, double g) {...}
    public static void M2(int k, double l) {...}
    public static int M3(int g) {...}
    public static void M4(int g) {...}
}
```

Les méthodes `A.M1` et `B.M1` sont compatibles avec les types délégués `D1` et, dans la `D2` mesure où ils ont le même type de retour et la même liste de paramètres ; Toutefois, ces types délégués sont deux types différents. ils ne sont donc pas interchangeables. Les méthodes `B.M2` , `B.M3` et `B.M4` sont incompatibles avec les types délégués `D1` et `D2` , dans la mesure où ils ont des types de retour ou des listes de paramètres différents.

Comme d'autres déclarations de type générique, les arguments de type doivent être fournis pour créer un type délégué construit. Les types de paramètres et le type de retour d'un type délégué construit sont créés en substituant, pour chaque paramètre de type dans la déclaration du délégué, l'argument de type correspondant du type délégué construit. Le type de retour et les types de paramètres résultants sont utilisés pour déterminer quelles méthodes sont compatibles avec un type délégué construit. Par exemple :

C#

```
delegate bool Predicate<T>(T value);

class X
{
    static bool F(int i) {...}
    static bool G(string s) {...}
}
```

La méthode `X.F` est compatible avec le type délégué `Predicate<int>` et la méthode `X.G` est compatible avec le type délégué `Predicate<string>` .

La seule façon de déclarer un type délégué est d'utiliser une *delegate_declaration*. Un type délégué est un type de classe dérivé de `System.Delegate` . Les types délégués étant implicitement `sealed` , il n'est pas possible de dériver un type d'un type délégué. Il n'est pas non plus possible de dériver un type de classe non délégué de `System.Delegate` . Notez que `System.Delegate` n'est pas lui-même un type délégué ; il s'agit d'un type de classe dont tous les types délégués sont dérivés.

C# fournit une syntaxe spéciale pour l'instanciation et l'appel de délégués. À l'exception de l'instanciation, toute opération qui peut être appliquée à une classe ou une instance

de classe peut également être appliquée à une instance ou une classe déléguée, respectivement. En particulier, il est possible d'accéder aux membres du `System.Delegate` type via la syntaxe d'accès aux membres habituelle.

L'ensemble de méthodes encapsulé par une instance de délégué est appelé une liste d'appel. Lorsqu'une instance de délégué est créée (déléguer la compatibilité) à partir d'une seule méthode, elle encapsule cette méthode et sa liste d'appel contient une seule entrée. Toutefois, lorsque deux instances de délégués non NULL sont combinées, leurs listes d'appel sont concaténées, dans l'opérande de l'ordre de gauche, puis dans l'opérande de droite, pour former une nouvelle liste d'appel, qui contient au moins deux entrées.

Les délégués sont combinés à l'aide des opérateurs binaires `+` (opérateur d'addition) et `+=` (assignation composée). Un délégué peut être supprimé d'une combinaison de délégués à l'aide de `-` l'opérateur binaire (soustraction) et des `-=` opérateurs (assignation composée). Les délégués peuvent être comparés pour déterminer leur égalité (opérateurs d'égalité de délégué).

L'exemple suivant illustre l'instanciation d'un certain nombre de délégués et les listes d'appel correspondantes :

C#

```
delegate void D(int x);

class C
{
    public static void M1(int i) {...}
    public static void M2(int i) {...}

}

class Test
{
    static void Main() {
        D cd1 = new D(C.M1);      // M1
        D cd2 = new D(C.M2);      // M2
        D cd3 = cd1 + cd2;        // M1 + M2
        D cd4 = cd3 + cd1;        // M1 + M2 + M1
        D cd5 = cd4 + cd3;        // M1 + M2 + M1 + M1 + M2
    }

}
```

Lorsque `cd1` et `cd2` sont instanciés, ils encapsulent chacun une méthode. Lorsque `cd3` est instancié, il a une liste d'appel de deux méthodes, `M1` et `M2` , dans cet ordre. `cd4`la

liste d'appel de contient `M1` , `M2` et `M1` , dans cet ordre. Enfin, `cd5` la liste d'appel de contient `M1` , `M2` , `M1` , `M1` , et `M2` , dans cet ordre. Pour obtenir plus d'exemples de combinaison (et de suppression) de délégués, consultez appel de délégué.

Compatibilité des délégués

Une méthode ou un délégué `M` est ***compatible*** avec un type délégué `D` si toutes les conditions suivantes sont vraies :

- `D` et `M` ont le même nombre de paramètres, et chaque paramètre dans `D` a le même `ref` `out` modificateur ou que le paramètre correspondant dans `M` .
- Pour chaque paramètre de valeur (paramètre sans `ref` ou `out` modificateur), une conversion d'identité (conversion d'identité) ou une conversion de référence implicite (conversions de référence implicites) existe à partir du type de paramètre dans `D` le type de paramètre correspondant dans `M` .
- Pour chaque `ref` `out` paramètre ou, le type de paramètre dans `D` est le même que le type de paramètre dans `M` .
- Une conversion d'identité ou de référence implicite existe du type de retour de `M` au type de retour de `D` .

Instanciation des délégués

Une instance d'un délégué est créée par un *delegate_creation_expression* (expressions de création de délégué) ou une conversion vers un type délégué. L'instance de délégué nouvellement créée fait référence à l'une des deux :

- Méthode statique référencée dans le *delegate_creation_expression* ou
- L'objet cible (qui ne peut pas être `null`) et la méthode d'instance référencée dans le *delegate_creation_expression*, ou
- Autre délégué.

Par exemple :

C#

```
delegate void D(int x);

class C
{
    public static void M1(int i) {...}
    public void M2(int i) {...}
}
```

```
class Test
{
    static void Main() {
        D cd1 = new D(C.M1);        // static method
        C t = new C();
        D cd2 = new D(t.M2);        // instance method
        D cd3 = new D(cd2);        // another delegate
    }
}
```

Une fois instanciés, les instances de délégué font toujours référence aux mêmes objet et méthode cible. Rappelez-vous que lorsque deux délégués sont combinés, ou lorsqu'un nouveau délégué est supprimé d'un autre, un nouveau délégué est obtenu avec sa propre liste d'appel ; les listes d'appel des délégués combinés ou supprimés restent inchangées.

Appel des délégués

C# fournit une syntaxe spéciale pour appeler un délégué. Lorsqu'une instance de délégué non null dont la liste d'appel contient une entrée est appelée, elle appelle la méthode à l'aide des mêmes arguments qu'elle a été donnée et retourne la même valeur que la méthode référencée. (Pour plus d'informations sur l'appel de délégué, consultez appel de délégué .) Si une exception se produit pendant l'appel d'un tel délégué et que cette exception n'est pas interceptée dans la méthode qui a été appelée, la recherche d'une clause catch d'exception continue dans la méthode qui a appelé le délégué, comme si cette méthode avait directement appelé la méthode à laquelle ce délégué a fait face.

L'appel d'une instance de délégué dont la liste d'appel contient plusieurs entrées se poursuit en appelant chaque méthode de la liste d'appel, de manière synchrone, dans l'ordre. Chaque méthode appelée reçoit le même jeu d'arguments que celui qui a été donné à l'instance de délégué. Si un tel appel de délégué comprend des paramètres de référence (paramètres de référence), chaque appel de méthode se produit avec une référence à la même variable ; les modifications apportées à cette variable par une méthode dans la liste d'appel seront visibles par les méthodes plus loin dans la liste d'appel. Si l'appel de délégué comprend des paramètres de sortie ou une valeur de retour, la valeur finale provient de l'appel du dernier délégué de la liste.

Si une exception se produit pendant le traitement de l'appel d'un délégué de ce type, et si cette exception n'est pas interceptée dans la méthode qui a été appelée, la recherche d'une clause catch d'exception continue dans la méthode qui a appelé le délégué, et toutes les méthodes situées plus loin dans la liste d'invocation ne sont pas appelées.

Toute tentative d'appel d'une instance de délégué dont la valeur est null entraîne une exception de type `System.NullReferenceException` .

L'exemple suivant montre comment instancier, combiner, supprimer et appeler des délégués :

C#

```
using System;

delegate void D(int x);

class C
{
    public static void M1(int i) {
        Console.WriteLine("C.M1: " + i);
    }

    public static void M2(int i) {
        Console.WriteLine("C.M2: " + i);
    }

    public void M3(int i) {
        Console.WriteLine("C.M3: " + i);
    }
}

class Test
{
    static void Main() {
        D cd1 = new D(C.M1);
        cd1(-1);                // call M1

        D cd2 = new D(C.M2);
        cd2(-2);                // call M2

        D cd3 = cd1 + cd2;
        cd3(10);                // call M1 then M2

        cd3 += cd1;
        cd3(20);                // call M1, M2, then M1

        C c = new C();
        D cd4 = new D(c.M3);
        cd3 += cd4;
        cd3(30);                // call M1, M2, M1, then M3

        cd3 -= cd1;             // remove last M1
        cd3(40);                // call M1, M2, then M3

        cd3 -= cd4;
        cd3(50);                // call M1 then M2
```

```
        cd3 -= cd2;
        cd3(60);                // call M1

        cd3 -= cd2;             // impossible removal is benign
        cd3(60);                // call M1

        cd3 -= cd1;             // invocation list is empty so cd3 is null

        cd3(70);                // System.NullReferenceException thrown

        cd3 -= cd1;             // impossible removal is benign
    }
}
```

Comme indiqué dans l'instruction `cd3 += cd1;` , un délégué peut être présent plusieurs fois dans une liste d'appel. Dans ce cas, elle est simplement appelée une fois par occurrence. Dans une liste d'appel telle que celle-ci, lorsque ce délégué est supprimé, la dernière occurrence dans la liste d'appel est celle qui est réellement supprimée.

Juste avant l'exécution de l'instruction finale, `cd3 -= cd1;` , le délégué `cd3` fait référence à une liste d'appel vide. Toute tentative de suppression d'un délégué d'une liste vide (ou de suppression d'un délégué inexistant d'une liste non vide) n'est pas une erreur.

La sortie produite est la suivante :

Console

```
C.M1: -1
C.M2: -2
C.M1: 10
C.M2: 10
C.M1: 20
C.M2: 20
C.M1: 20
C.M1: 30
C.M2: 30
C.M1: 30
C.M3: 30
C.M1: 40
C.M2: 40
C.M3: 40
C.M1: 50
C.M2: 50
C.M1: 60
C.M1: 60
```

Exceptions

Article • 13/05/2021

Les exceptions en C# offrent un moyen structuré, uniforme et de type sécurisé de gestion des conditions d'erreur au niveau du système et au niveau de l'application. Le mécanisme d'exception en C# est très similaire à celui de C++, avec quelques différences importantes :

- En C#, toutes les exceptions doivent être représentées par une instance d'un type de classe dérivé de `System.Exception` . En C++, toute valeur de tout type peut être utilisée pour représenter une exception.
- En C#, un bloc finally (instruction try) peut être utilisé pour écrire un code d'arrêt qui s'exécute à la fois dans une exécution normale et dans des conditions exceptionnelles. Ce code est difficile à écrire en C++ sans duplication de code.
- En C#, les exceptions au niveau système telles que le dépassement de capacité, la division par zéro et les déréférencements null ont des classes d'exception bien définies et sont associées à des conditions d'erreur au niveau de l'application.

Causes des exceptions

Une exception peut être levée de deux façons différentes.

- Une `throw` instruction (instruction throw) lève une exception immédiatement et sans condition. Le contrôle n'atteint jamais l'instruction qui suit immédiatement `throw` .
- Certaines conditions exceptionnelles qui se produisent pendant le traitement des instructions et expressions C# provoquent une exception dans certaines circonstances lorsque l'opération ne peut pas être effectuée normalement. Par exemple, une opération de division entière (opérateur de division) lève une `System.DivideByZeroException` si le dénominateur est égal à zéro. Consultez classes d'exceptions communes pour obtenir la liste des différentes exceptions qui peuvent se produire de cette façon.

System.Exception, classe

La `System.Exception` classe est le type de base de toutes les exceptions. Cette classe possède quelques propriétés notables que toutes les exceptions partagent :

- `Message` est une propriété en lecture seule de type `string` qui contient une description explicite de la raison de l'exception.
- `InnerException` est une propriété en lecture seule de type `Exception` . Si sa valeur n'est pas null, elle fait référence à l'exception qui a provoqué l'exception actuelle, autrement dit, l'exception actuelle a été levée dans un bloc catch qui gère le `InnerException` . Sinon, sa valeur est null, ce qui indique que cette exception n'a pas été provoquée par une autre exception. Le nombre d'objets d'exception chaînés ensemble de cette manière peut être arbitraire.

La valeur de ces propriétés peut être spécifiée dans les appels au constructeur d'instance pour `System.Exception` .

Gestion des exceptions

Les exceptions sont gérées par une `try` instruction (instruction try).

Quand une exception se produit, le système recherche la clause la plus proche `catch` qui peut gérer l'exception, comme déterminé par le type au moment de l'exécution de l'exception. Tout d'abord, la méthode actuelle recherche une instruction englobante lexicale `try` , et les clauses catch associées de l'instruction try sont considérées dans l'ordre. En cas d'échec, la méthode qui a appelé la méthode actuelle fait l'objet d'une recherche d'une instruction englobante lexicale `try` qui encadre le point de l'appel à la méthode actuelle. Cette recherche se poursuit jusqu'à ce qu'une `catch` clause pouvant gérer l'exception actuelle soit trouvée, en nommant une classe d'exception qui est de la même classe, ou une classe de base, du type au moment de l'exécution de l'exception levée. Une `catch` clause qui ne nomme pas une classe d'exception peut gérer n'importe quelle exception.

Une fois qu'une clause catch correspondante est trouvée, le système se prépare à transférer le contrôle à la première instruction de la clause catch. Avant le début de l'exécution de la clause catch, le système exécute pour la première fois, dans l'ordre, toutes les `finally` clauses associées à des instructions Try plus imbriquées que celle qui a intercepté l'exception.

Si aucune clause catch correspondante n'est trouvée, l'une des deux situations suivantes se produit :

- Si la recherche d'une clause catch correspondante atteint un constructeur statique (constructeurs statiques) ou un initialiseur de champ statique, une `System.TypeInitializationException` est levée au point qui a déclenché l'appel du

constructeur statique. L'exception interne de `System.TypeInitializationException` contient l'exception qui a été levée à l'origine.
- Si la recherche de clauses catch correspondantes atteint le code qui a démarré initialement le thread, l'exécution du thread se termine. L'impact d'une telle terminaison est défini par l'implémentation.

Les exceptions qui se produisent pendant l'exécution du destructeur méritent une mention spéciale. Si une exception se produit pendant l'exécution du destructeur et que cette exception n'est pas interceptée, l'exécution de ce destructeur est terminée et le destructeur de la classe de base (le cas échéant) est appelé. S'il n'existe aucune classe de base (comme dans le cas du `object` type) ou s'il n'existe aucun destructeur de classe de base, l'exception est ignorée.

Classes d'exceptions courantes

Les exceptions suivantes sont levées par certaines opérations C#.

`System.ArithmeticException`	Classe de base pour les exceptions qui se produisent pendant des opérations arithmétiques, telles que `System.DivideByZeroException` et `System.OverflowException`.
`System.ArrayTypeMismatchException`	Levée lorsqu'un magasin dans un tableau échoue parce que le type réel de l'élément stocké est incompatible avec le type réel du tableau.
`System.DivideByZeroException`	Levée lorsqu'une tentative de division d'une valeur intégrale par zéro se produit.
`System.IndexOutOfRangeException`	Levée lors d'une tentative d'indexation d'un tableau via un index inférieur à zéro ou en dehors des limites du tableau.
`System.InvalidCastException`	Levée lorsqu'une conversion explicite d'un type de base ou d'une interface en type dérivé échoue au moment de l'exécution.
`System.NullReferenceException`	Levée lorsqu'une `null` référence est utilisée de manière à ce que l'objet référencé soit requis.
`System.OutOfMemoryException`	Levée lors de l'échec d'une tentative d'allocation de mémoire (via `new`).
`System.OverflowException`	Levée quand une opération dans un contexte `checked` engendre un dépassement.

`System.StackOverflowException`	Levée lorsque la pile d'exécution est épuisée par un trop grand nombre d'appels de méthode en attente ; en général, une récurrence très profonde ou non liée.
`System.TypeInitializationException`	Levée lorsqu'un constructeur statique lève une exception et `catch` qu'il n'existe aucune clause pour l'intercepter.

Attributs

Article • 13/05/2021

Une grande partie du langage C# permet au programmeur de spécifier des informations déclaratives sur les entités définies dans le programme. Par exemple, l'accessibilité d'une méthode dans une classe est spécifiée en la décorant avec le *method_modifier* s `public` „ `protected` `internal` et `private` .

C# permet aux programmeurs d'inventer de nouveaux genres d'informations déclaratives, appelés ***attributs***. Les programmeurs peuvent ensuite attacher des attributs à diverses entités de programme et récupérer les informations d'attribut dans un environnement d'exécution. Par exemple, une infrastructure peut définir un `HelpAttribute` attribut qui peut être placé sur certains éléments de programme (tels que les classes et les méthodes) pour fournir un mappage de ces éléments de programme à leur documentation.

Les attributs sont définis via la déclaration de classes d'attributs (classes d'attributs), qui peuvent avoir des paramètres positionnels et nommés (paramètres positionnels et nommés). Les attributs sont attachés aux entités dans un programme C# à l'aide de spécifications d'attributs (spécification d'attribut) et peuvent être récupérés au moment de l'exécution en tant qu'instances d'attribut (instances d'attribut).

Classes d'attributs

Une classe qui dérive de la classe abstraite `System.Attribute` , qu'elle soit directement ou indirectement, est une *__classe d'attributs__ _. La déclaration d'une classe d'attributs définit un nouveau genre d'*attribut* _ * qui peut être placé sur une déclaration. Par Convention, les classes d'attributs sont nommées avec le suffixe `Attribute` . Les utilisations d'un attribut peuvent inclure ou omettre ce suffixe.

Utilisation des attributs

L'attribut `AttributeUsage` (attribut AttributeUsage) est utilisé pour décrire comment une classe d'attributs peut être utilisée.

`AttributeUsage` possède un paramètre positionnel (paramètres positionnels et nommés) qui permet à une classe d'attributs de spécifier les genres de déclarations sur lesquelles elle peut être utilisée. L'exemple

```
C#
```

```
using System;

[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]
public class SimpleAttribute: Attribute
{
    ...
}
```

définit une classe d'attributs nommée `SimpleAttribute` qui peut être placée sur *class_declaration* s et *interface_declaration* s uniquement. L'exemple

C#

```
[Simple] class Class1 {...}

[Simple] interface Interface1 {...}
```

affiche plusieurs utilisations de l' `Simple` attribut. Bien que cet attribut soit défini avec le nom `SimpleAttribute` , lorsque cet attribut est utilisé, le `Attribute` suffixe peut être omis, ce qui donne le nom abrégé `Simple` . Ainsi, l'exemple ci-dessus est sémantiquement équivalent à ce qui suit :

C#

```
[SimpleAttribute] class Class1 {...}

[SimpleAttribute] interface Interface1 {...}
```

`AttributeUsage` a un paramètre nommé (positionnel et paramètres nommés) appelé `AllowMultiple` , qui indique si l'attribut peut être spécifié plusieurs fois pour une entité donnée. Si `AllowMultiple` pour une classe d'attributs a la valeur true, cette classe d'attributs est une **classe d'attributs à plusieurs utilisation** _ et peut être spécifiée plusieurs fois sur une entité. Si `AllowMultiple` pour une classe d'attributs a la valeur false ou qu'elle n'est pas spécifiée, cette classe d'attributs est une classe d'attributs _ * à *usage unique** *, et peut être spécifiée au plus une fois sur une entité.

L'exemple

C#

```
using System;

[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
public class AuthorAttribute: Attribute
{
```

```
    private string name;

    public AuthorAttribute(string name) {
        this.name = name;
    }

    public string Name {
        get { return name; }
    }
}
```

définit une classe d'attributs à utilisation multiple nommée `AuthorAttribute` . L'exemple

C#

```
[Author("Brian Kernighan"), Author("Dennis Ritchie")]
class Class1
{
    ...
}
```

montre une déclaration de classe avec deux utilisations de l' `Author` attribut.

`AttributeUsage` a un autre paramètre nommé appelé `Inherited` , qui indique si l'attribut, lorsqu'il est spécifié sur une classe de base, est également hérité par les classes qui dérivent de cette classe de base. Si `Inherited` pour une classe d'attributs a la valeur true, cet attribut est hérité. Si `Inherited` pour une classe d'attributs a la valeur false, cet attribut n'est pas hérité. S'il n'est pas spécifié, sa valeur par défaut est true.

Une classe d'attributs qui `X` n'a pas d' `AttributeUsage` attribut attaché, comme dans

C#

```
using System;

class X: Attribute {...}
```

équivaut à ce qui suit :

C#

```
using System;

[AttributeUsage(
    AttributeTargets.All,
    AllowMultiple = false,
    Inherited = true)
```

```
]
class X: Attribute {...}
```

Paramètres positionnels et nommés

Les classes d'attributs peuvent avoir * paramètres **positionnels** _ et _ *paramètres nommés* *. Chaque constructeur d'instance public d'une classe d'attributs définit une séquence valide de paramètres positionnels pour cette classe d'attributs. Chaque champ et propriété en lecture-écriture public non statique d'une classe d'attributs définit un paramètre nommé pour la classe d'attributs.

L'exemple

C#

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class HelpAttribute: Attribute
{
    public HelpAttribute(string url) {        // Positional parameter
        ...
    }

    public string Topic {                     // Named parameter
        get {...}
        set {...}
    }

    public string Url {
        get {...}
    }
}
```

définit une classe d'attributs nommée `HelpAttribute` qui a un paramètre positionnel, `url` , et un paramètre nommé, `Topic` . Bien qu'il ne soit pas statique et public, la propriété `Url` ne définit pas de paramètre nommé, car il n'est pas accessible en lecture-écriture.

Cette classe d'attributs peut être utilisée comme suit :

C#

```
[Help("http://www.mycompany.com/.../Class1.htm")]
class Class1
{
    ...
```

```
}

[Help("http://www.mycompany.com/.../Misc.htm", Topic = "Class2")]
class Class2
{
    ...
}
```

Types de paramètre d'attribut

Les types de paramètres positionnels et nommés pour une classe d'attributs sont limités aux ***types de paramètres d'attribut***, qui sont les suivants :

- L'un des types suivants : `bool` , `byte` , `char` , `double` , `float` , `int` , `long` , `sbyte` , `short` , `string` , `uint` , `ulong` , `ushort` .
- Le type `object`.
- Le type `System.Type`.
- Un type enum, à condition qu'il ait une accessibilité publique et que les types dans lesquels il est imbriqué (le cas échéant) disposent également d'un accès public (spécification d'attribut).
- Tableaux unidimensionnels des types ci-dessus.
- Un argument de constructeur ou un champ public qui ne possède pas l'un de ces types, ne peut pas être utilisé en tant que paramètre positionnel ou nommé dans une spécification d'attribut.

Spécification des attributs

*La **spécification d'attribut** _ est l'application d'un attribut précédemment défini à une déclaration. Un attribut est une partie des informations déclaratives supplémentaires spécifiées pour une déclaration. Les attributs peuvent être spécifiés au niveau de la portée globale (pour spécifier des attributs sur l'assembly ou le module conteneur) et pour _type_declaration * s (déclarations de type). *class_member_declaration* s (contraintes de paramètre de type), *interface_member_declaration* s (membres d'interface), *struct_member_declaration* s (membres de struct), *enum_member_declaration* s (membres enum), *Accessor_declarations* (accesseurs), *event_accessor_declarations* (événements de type champ) et *formal_parameter_list* s (paramètres de méthode).

Les attributs sont spécifiés dans les ***sections d'attribut***. Une section d'attribut se compose d'une paire de crochets, qui entourent une liste séparée par des virgules d'un ou de plusieurs attributs. L'ordre dans lequel les attributs sont spécifiés dans une telle liste et l'ordre dans lequel les sections attachées à la même entité de programme sont

organisées ne sont pas significatifs. Par exemple, les spécifications d'attribut `[A][B]` ,, `[B][A]` `[A,B]` et `[B,A]` sont équivalentes.

antlr

```
global_attributes
    : global_attribute_section+
    ;

global_attribute_section
    : '[' global_attribute_target_specifier attribute_list ']'
    | '[' global_attribute_target_specifier attribute_list ',' ']'
    ;

global_attribute_target_specifier
    : global_attribute_target ':'
    ;

global_attribute_target
    : 'assembly'
    | 'module'
    ;

attributes
    : attribute_section+
    ;

attribute_section
    : '[' attribute_target_specifier? attribute_list ']'
    | '[' attribute_target_specifier? attribute_list ',' ']'
    ;

attribute_target_specifier
    : attribute_target ':'
    ;

attribute_target
    : 'field'
    | 'event'
    | 'method'
    | 'param'
    | 'property'
    | 'return'
    | 'type'
    ;

attribute_list
    : attribute (',' attribute)*
    ;

attribute
    : attribute_name attribute_arguments?
    ;
```

```
attribute_name
    : type_name
    ;

attribute_arguments
    : '(' positional_argument_list? ')'
    | '(' positional_argument_list ',' named_argument_list ')'
    | '(' named_argument_list ')'
    ;

positional_argument_list
    : positional_argument (',' positional_argument)*
    ;

positional_argument
    : attribute_argument_expression
    ;

named_argument_list
    : named_argument (','  named_argument)*
    ;

named_argument
    : identifier '=' attribute_argument_expression
    ;

attribute_argument_expression
    : expression
    ;
```

Un attribut se compose d'un *ATTRIBUTE_NAME* et d'une liste facultative d'arguments positionnels et nommés. Les arguments positionnels (le cas échéant) précèdent les arguments nommés. Un argument positionnel se compose d'un *attribute_argument_expression*; un argument nommé se compose d'un nom, suivi d'un signe égal, suivi d'un *attribute_argument_expression*, qui, ensemble, sont limités par les mêmes règles que l'assignation simple. L'ordre des arguments nommés n'est pas significatif.

Le *ATTRIBUTE_NAME* identifie une classe d'attributs. Si la forme de *ATTRIBUTE_NAME* est *type_name* , ce nom doit faire référence à une classe d'attributs. Sinon, une erreur de compilation se produit. L'exemple

C#

```
class Class1 {}

[Class1] class Class2 {}    // Error
```

génère une erreur de compilation, car il tente d'utiliser `Class1` en tant que classe d'attributs lorsque `Class1` n'est pas une classe d'attributs.

Certains contextes autorisent la spécification d'un attribut sur plusieurs cibles. Un programme peut spécifier explicitement la cible en incluant un *attribute_target_specifier*. Lorsqu'un attribut est placé au niveau global, une *global_attribute_target_specifier* est requise. Dans tous les autres emplacements, une valeur par défaut raisonnable est appliquée, mais un *attribute_target_specifier* peut être utilisé pour confirmer ou substituer la valeur par défaut dans certains cas ambigus (ou pour simplement confirmer la valeur par défaut dans les cas non ambigus). Par conséquent, en général, les *attribute_target_specifier* s peuvent être omises, sauf au niveau global. Les contextes potentiellement ambigus sont résolus comme suit :

- Un attribut spécifié au niveau de la portée globale peut s'appliquer à l'assembly cible ou au module cible. Aucune valeur par défaut n'existe pour ce contexte, donc un *attribute_target_specifier* est toujours requis dans ce contexte. La présence de l' `assembly` *attribute_target_specifier* indique que l'attribut s'applique à l'assembly cible ; la présence du `module` *attribute_target_specifier* indique que l'attribut s'applique au module cible.
- Un attribut spécifié dans une déclaration de délégué peut s'appliquer au délégué déclaré ou à sa valeur de retour. En l'absence d'un *attribute_target_specifier*, l'attribut s'applique au délégué. La présence de l' `type` *attribute_target_specifier* indique que l'attribut s'applique au délégué ; la présence de l' `return` *attribute_target_specifier* indique que l'attribut s'applique à la valeur de retour.
- Un attribut spécifié dans une déclaration de méthode peut s'appliquer à la méthode déclarée ou à sa valeur de retour. En l'absence d'un *attribute_target_specifier*, l'attribut s'applique à la méthode. La présence de l' `method` *attribute_target_specifier* indique que l'attribut s'applique à la méthode ; la présence du `return` *attribute_target_specifier* indique que l'attribut s'applique à la valeur de retour.
- Un attribut spécifié dans une déclaration d'opérateur peut s'appliquer à l'opérateur déclaré ou à sa valeur de retour. En l'absence d'un *attribute_target_specifier*, l'attribut s'applique à l'opérateur. La présence de l' `method` *attribute_target_specifier* indique que l'attribut s'applique à l'opérateur ; la présence de l' `return` *attribute_target_specifier* indique que l'attribut s'applique à la valeur de retour.
- Un attribut spécifié dans une déclaration d'événement qui omet des accesseurs d'événement peut s'appliquer à l'événement déclaré, au champ associé (si l'événement n'est pas abstrait) ou aux méthodes Add et Remove associées. En l'absence d'un *attribute_target_specifier*, l'attribut s'applique à l'événement. La présence de l' `event` *attribute_target_specifier* indique que l'attribut s'applique à

l'événement ; la présence de l' `field` *attribute_target_specifier* indique que l'attribut s'applique au champ ; et la présence de la `method` *attribute_target_specifier* indique que l'attribut s'applique aux méthodes.

- Un attribut spécifié dans une déclaration d'accesseur get pour une déclaration de propriété ou d'indexeur peut s'appliquer à la méthode associée ou à sa valeur de retour. En l'absence d'un *attribute_target_specifier*, l'attribut s'applique à la méthode. La présence de l' `method` *attribute_target_specifier* indique que l'attribut s'applique à la méthode ; la présence du `return` *attribute_target_specifier* indique que l'attribut s'applique à la valeur de retour.
- Un attribut spécifié sur un accesseur Set pour une déclaration de propriété ou d'indexeur peut s'appliquer à la méthode associée ou à son paramètre seul implicite. En l'absence d'un *attribute_target_specifier*, l'attribut s'applique à la méthode. La présence de l' `method` *attribute_target_specifier* indique que l'attribut s'applique à la méthode ; la présence de l' `param` *attribute_target_specifier* indique que l'attribut s'applique au paramètre ; la présence de la `return` *attribute_target_specifier* indique que l'attribut s'applique à la valeur de retour.
- Un attribut spécifié dans une déclaration d'accesseur Add ou Remove pour une déclaration d'événement peut s'appliquer à la méthode associée ou à son paramètre solitaire. En l'absence d'un *attribute_target_specifier*, l'attribut s'applique à la méthode. La présence de l' `method` *attribute_target_specifier* indique que l'attribut s'applique à la méthode ; la présence de l' `param` *attribute_target_specifier* indique que l'attribut s'applique au paramètre ; la présence de la `return` *attribute_target_specifier* indique que l'attribut s'applique à la valeur de retour.

Dans d'autres contextes, l'inclusion d'un *attribute_target_specifier* est autorisée, mais inutile. Par exemple, une déclaration de classe peut inclure ou omettre le spécificateur `type` :

```
C#
```

```
[type: Author("Brian Kernighan")]
class Class1 {}

[Author("Dennis Ritchie")]
class Class2 {}
```

La spécification d'un *attribute_target_specifier* non valide est une erreur. Par exemple, le spécificateur `param` ne peut pas être utilisé sur une déclaration de classe :

```
C#
```

```
[param: Author("Brian Kernighan")]        // Error
class Class1 {}
```

Par Convention, les classes d'attributs sont nommées avec le suffixe `Attribute` . Une *ATTRIBUTE_NAME* de la forme *type_name* peut inclure ou omettre ce suffixe. Si une classe d'attributs est trouvée avec et sans ce suffixe, une ambiguïté est présente et une erreur de compilation se produit. Si le *ATTRIBUTE_NAME* est épelé de telle sorte que son *identificateur* le plus à droite soit un identificateur textuel (identificateurs), seul un attribut sans suffixe est mis en correspondance, ce qui permet de résoudre cette ambiguïté. L'exemple

C#

```
using System;

[AttributeUsage(AttributeTargets.All)]
public class X: Attribute
{}

[AttributeUsage(AttributeTargets.All)]
public class XAttribute: Attribute
{}

[X]                     // Error: ambiguity
class Class1 {}

[XAttribute]            // Refers to XAttribute
class Class2 {}

[@X]                    // Refers to X
class Class3 {}

[@XAttribute]           // Refers to XAttribute
class Class4 {}
```

montre deux classes d'attributs nommées `X` et `XAttribute` . L'attribut `[X]` est ambigu, car il peut faire référence à `X` ou à `XAttribute` . L'utilisation d'un identificateur textuel permet de spécifier l'intention exacte dans de tels cas rares. L'attribut `[XAttribute]` n'est pas ambigu (même s'il s'agit d'une classe d'attributs nommée `XAttributeAttribute` !). Si la déclaration pour la classe `X` est supprimée, les deux attributs font référence à la classe d'attributs nommée `XAttribute` , comme suit :

C#

```
using System;
```

```
[AttributeUsage(AttributeTargets.All)]
public class XAttribute: Attribute
{}

[X]                     // Refers to XAttribute
class Class1 {}

[XAttribute]            // Refers to XAttribute
class Class2 {}

[@X]                    // Error: no attribute named "X"
class Class3 {}
```

Il s'agit d'une erreur au moment de la compilation pour utiliser une classe d'attributs à usage unique plusieurs fois sur la même entité. L'exemple

C#

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class HelpStringAttribute: Attribute
{
    string value;

    public HelpStringAttribute(string value) {
        this.value = value;
    }

    public string Value {
        get {...}
    }
}

[HelpString("Description of Class1")]
[HelpString("Another description of Class1")]
public class Class1 {}
```

génère une erreur au moment de la compilation, car il tente d'utiliser `HelpString` , qui est une classe d'attributs à usage unique, plusieurs fois sur la déclaration de `Class1` .

Une expression `E` est une *attribute_argument_expression* si toutes les instructions suivantes sont vraies :

- Le type de `E` est un type de paramètre d'attribut (types de paramètres d'attribut).
- Au moment de la compilation, la valeur de `E` peut être résolue en l'un des éléments suivants :
  - Valeur constante.
  - Objet `System.Type`.

- Tableau unidimensionnel de *attribute_argument_expression* s.

Par exemple :

C#

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class TestAttribute: Attribute
{
    public int P1 {
        get {...}
        set {...}
    }

    public Type P2 {
        get {...}
        set {...}
    }

    public object P3 {
        get {...}
        set {...}
    }
}

[Test(P1 = 1234, P3 = new int[] {1, 3, 5}, P2 = typeof(float))]
class MyClass {}
```

Un *typeof_expression* (opérateur typeof) utilisé comme expression d'argument d'attribut peut faire référence à un type non générique, à un type construit fermé ou à un type générique indépendant, mais ne peut pas faire référence à un type ouvert. Cela permet de s'assurer que l'expression peut être résolue au moment de la compilation.

C#

```
class A: Attribute
{
    public A(Type t) {...}
}

class G<T>
{
    [A(typeof(T))] T t;                  // Error, open type in attribute
}

class X
{
    [A(typeof(List<int>))] int x;        // Ok, closed constructed type
```

```
    [A(typeof(List<>))] int y;          // Ok, unbound generic type
}
```

Instances d'attribut

Une ***instance d'attribut*** est une instance qui représente un attribut au moment de l'exécution. Un attribut est défini avec une classe d'attributs, des arguments positionnels et des arguments nommés. Une instance d'attribut est une instance de la classe d'attributs qui est initialisée avec les arguments positionnels et nommés.

La récupération d'une instance d'attribut implique le traitement au moment de la compilation et au moment de l'exécution, comme décrit dans les sections suivantes.

Compilation d'un attribut

La compilation d'un *attribut* avec une classe d'attributs `T` , *positional_argument_list* `P` et *named_argument_list* `N` , se compose des étapes suivantes :

- Suivez les étapes de traitement au moment de la compilation pour compiler une *object_creation_expression* du formulaire `new T(P)` . Ces étapes entraînent une erreur au moment de la compilation ou déterminent un constructeur `C` d'instance sur `T` qui peut être appelé au moment de l'exécution.
- Si `C` n'a pas d'accessibilité publique, une erreur de compilation se produit.
- Pour chaque *named_argument* `Arg` dans `N` :
  - Indiquez `Name` l' *identificateur* de la *named_argument* `Arg` .
  - `Name` doit identifier une propriété ou un champ public en lecture-écriture non statique sur `T` . Si n' `T` a pas de champ ou de propriété de ce type, une erreur de compilation se produit.
- Conservez les informations suivantes pour l'instanciation au moment de l'exécution de l'attribut : la classe d'attributs `T` , le constructeur `C` d'instance sur `T` , le *positional_argument_list* `P` et le *named_argument_list* `N` .

Récupération au moment de l'exécution d'une instance d'attribut

La compilation d'un *attribut* produit une classe d'attributs `T` , un constructeur `C` d'instance sur `T` , une *positional_argument_list* `P` et un *named_argument_list* `N` . À partir de ces informations, une instance d'attribut peut être récupérée au moment de l'exécution à l'aide des étapes suivantes :

- Suivez les étapes de traitement au moment de l'exécution pour exécuter une *object_creation_expression* du formulaire `new T(P)` , à l'aide du constructeur d'instance `C` déterminé au moment de la compilation. Ces étapes entraînent une exception ou produisent une instance `O` de `T` .
- Pour chaque *named_argument* `Arg` dans `N` , dans l'ordre :
  - Indiquez `Name` l' *identificateur* de la *named_argument* `Arg` . Si `Name` n'identifie pas une propriété ou un champ en lecture/écriture public non statique sur `O` , une exception est levée.
  - `Value`À la suite de l'évaluation de la *attribute_argument_expression* de `Arg` .
  - Si `Name` identifie un champ sur `O` , définissez ce champ sur `Value` .
  - Sinon, `Name` identifie une propriété sur `O` . Définissez cette propriété sur `Value`.
  - Le résultat est `O` , une instance de la classe d'attributs `T` qui a été initialisée avec le *positional_argument_list* `P` et le *named_argument_list* `N` .

Attributs réservés

Un petit nombre d'attributs affecte le langage d'une certaine façon. Parmi ces attributs, citons notamment :

- `System.AttributeUsageAttribute` (Attribut AttributeUsage), qui est utilisé pour décrire la façon dont une classe d'attributs peut être utilisée.
- `System.Diagnostics.ConditionalAttribute` (L'attribut conditionnel), qui est utilisé pour définir des méthodes conditionnelles.
- `System.ObsoleteAttribute` (Attribut obsolète), qui est utilisé pour marquer un membre comme obsolète.
- `System.Runtime.CompilerServices.CallerLineNumberAttribute`, `System.Runtime.CompilerServices.CallerFilePathAttribute` et `System.Runtime.CompilerServices.CallerMemberNameAttribute` (attributs des informationsde l'appelant), qui sont utilisés pour fournir des informations sur le contexte d'appel aux paramètres facultatifs.

Attribut AttributeUsage

L'attribut `AttributeUsage` est utilisé pour décrire la manière dont la classe d'attributs peut être utilisée.

Une classe décorée avec l' `AttributeUsage` attribut doit dériver de `System.Attribute` , directement ou indirectement. Sinon, une erreur de compilation se produit.

```
C#
```

```
namespace System
{
    [AttributeUsage(AttributeTargets.Class)]
    public class AttributeUsageAttribute: Attribute
    {
        public AttributeUsageAttribute(AttributeTargets validOn) {...}
        public virtual bool AllowMultiple { get {...} set {...} }
        public virtual bool Inherited { get {...} set {...} }
        public virtual AttributeTargets ValidOn { get {...} }
    }

    public enum AttributeTargets
    {
        Assembly     = 0x0001,
        Module       = 0x0002,
        Class        = 0x0004,
        Struct       = 0x0008,
        Enum         = 0x0010,
        Constructor  = 0x0020,
        Method       = 0x0040,
        Property     = 0x0080,
        Field        = 0x0100,
        Event        = 0x0200,
        Interface    = 0x0400,
        Parameter    = 0x0800,
        Delegate     = 0x1000,
        ReturnValue  = 0x2000,

        All = Assembly | Module | Class | Struct | Enum | Constructor |
            Method | Property | Field | Event | Interface | Parameter |
            Delegate | ReturnValue
    }
}
```

L'attribut Conditional

L'attribut `Conditional` active la définition des *__méthodes conditionnelles__ _ et _ *classes d'attributs conditionnelles* *.

C#

```
namespace System.Diagnostics
{
    [AttributeUsage(AttributeTargets.Method | AttributeTargets.Class,
AllowMultiple = true)]
    public class ConditionalAttribute: Attribute
    {
        public ConditionalAttribute(string conditionString) {...}
        public string ConditionString { get {...} }
```

```
    }
}
```

Méthodes conditionnelles

Une méthode décorée avec l' `Conditional` attribut est une méthode conditionnelle. L' `Conditional` attribut indique une condition en testant un symbole de compilation conditionnelle. Les appels à une méthode conditionnelle sont inclus ou omis, selon que ce symbole est défini ou non au point de l'appel. Si le symbole est défini, l'appel est inclus ; dans le cas contraire, l'appel (y compris l'évaluation du récepteur et des paramètres de l'appel) est omis.

Une méthode conditionnelle est soumise aux restrictions suivantes :

- La méthode conditionnelle doit être une méthode dans un *class_declaration* ou *struct_declaration*. Une erreur de compilation se produit si l' `Conditional` attribut est spécifié sur une méthode dans une déclaration d'interface.
- La méthode conditionnelle doit avoir un type de retour `void` .
- La méthode conditionnelle ne doit pas être marquée avec le `override` modificateur. Toutefois, une méthode conditionnelle peut être marquée avec le `virtual` modificateur. Les substitutions d'une telle méthode sont implicitement conditionnelles et ne doivent pas être explicitement marquées avec un `Conditional` attribut.
- La méthode conditionnelle ne doit pas être une implémentation d'une méthode d'interface. Sinon, une erreur de compilation se produit.

En outre, une erreur de compilation se produit si une méthode conditionnelle est utilisée dans un *delegate_creation_expression*. L'exemple

C#

```
#define DEBUG

using System;
using System.Diagnostics;

class Class1
{
    [Conditional("DEBUG")]
    public static void M() {
        Console.WriteLine("Executed Class1.M");
    }
}

class Class2
```

```
{
    public static void Test() {
        Class1.M();
    }
}
```

déclare `Class1.M` en tant que méthode conditionnelle. `Class2`la `Test` méthode de appelle cette méthode. Étant donné que le symbole de compilation conditionnelle `DEBUG` est défini, si `Class2.Test` est appelé, il appellera `M` . Si le symbole `DEBUG` n'a pas été défini, `Class2.Test` n'appelle pas `Class1.M` .

Il est important de noter que l'inclusion ou l'exclusion d'un appel à une méthode conditionnelle est contrôlée par les symboles de compilation conditionnelle au point de l'appel. Dans l'exemple

Fichier `class1.cs` :

C#

```
using System.Diagnostics;

class Class1
{
    [Conditional("DEBUG")]
    public static void F() {
        Console.WriteLine("Executed Class1.F");
    }
}
```

Fichier `class2.cs` :

C#

```
#define DEBUG

class Class2
{
    public static void G() {
        Class1.F();                // F is called
    }
}
```

Fichier `class3.cs` :

C#

```
#undef DEBUG

class Class3
{
    public static void H() {
        Class1.F();                // F is not called
    }
}
```

les classes `Class2` et `Class3` contiennent chacune des appels à la méthode conditionnelle `Class1.F` , qui est conditionnelle selon qu' `DEBUG` elle est définie ou non. Étant donné que ce symbole est défini dans le contexte de `Class2` , mais pas `Class3` , l'appel à `F` dans `Class2` est inclus, tandis que l'appel à `F` dans `Class3` est omis.

L'utilisation de méthodes conditionnelles dans une chaîne d'héritage peut prêter à confusion. Les appels effectués à une méthode conditionnelle par `base` le biais de, du formulaire `base.M` , sont soumis aux règles d'appel de méthode conditionnelles normales. Dans l'exemple

Fichier `class1.cs` :

C#

```
using System;
using System.Diagnostics;

class Class1
{
    [Conditional("DEBUG")]
    public virtual void M() {
        Console.WriteLine("Class1.M executed");
    }
}
```

Fichier `class2.cs` :

C#

```
using System;

class Class2: Class1
{
    public override void M() {
        Console.WriteLine("Class2.M executed");
        base.M();                        // base.M is not called!
    }
}
```

Fichier `class3.cs` :

```C#
#define DEBUG

using System;

class Class3
{
    public static void Test() {
        Class2 c = new Class2();
        c.M();                              // M is called
    }
}
```

`Class2` comprend un appel à l' `M` défini dans sa classe de base. Cet appel est omis, car la méthode de base est conditionnelle en fonction de la présence du symbole `DEBUG` , qui n'est pas défini. Ainsi, la méthode écrit uniquement dans la console « `Class2.M executed` ». L'utilisation judicieuse de *pp_declaration* s peut éliminer de tels problèmes.

Classes d'attributs conditionnelles

Une classe d'attributs (classes d'attributs) décorée avec un ou plusieurs `Conditional` attributs est une classe d' ***attributs conditionnels***. Une classe d'attributs conditionnel est donc associée aux symboles de compilation conditionnelle déclarés dans ses `Conditional` attributs. Cet exemple :

```C#
using System;
using System.Diagnostics;
[Conditional("ALPHA")]
[Conditional("BETA")]
public class TestAttribute : Attribute {}
```

déclare `TestAttribute` comme une classe d'attributs conditionnels associée aux symboles de compilation conditionnelle `ALPHA` et `BETA` .

Les spécifications d'attributs (spécification d'attribut) d'un attribut conditionnel sont incluses si un ou plusieurs de ses symboles de compilation conditionnelle associés sont définis au point de spécification, sinon la spécification d'attribut est omise.

Il est important de noter que l'inclusion ou l'exclusion d'une spécification d'attribut d'une classe d'attributs conditionnel est contrôlée par les symboles de compilation conditionnelle au point de la spécification. Dans l'exemple

Fichier `test.cs` :

```
C#

using System;
using System.Diagnostics;

[Conditional("DEBUG")]

public class TestAttribute : Attribute {}
```

Fichier `class1.cs` :

```
C#

#define DEBUG

[Test]                // TestAttribute is specified

class Class1 {}
```

Fichier `class2.cs` :

```
C#

#undef DEBUG

[Test]                 // TestAttribute is not specified

class Class2 {}
```

les classes `Class1` et `Class2` sont toutes décorées avec l'attribut `Test` , qui est conditionnel selon qu'il `DEBUG` est défini ou non. Étant donné que ce symbole est défini dans le contexte de `Class1` , mais pas `Class2` , la spécification de l' `Test` attribut sur `Class1` est incluse, alors que la spécification de l' `Test` attribut sur `Class2` est omise.

Attribut Obsolete

L'attribut `Obsolete` est utilisé pour marquer des types et des membres de types qui ne doivent plus être utilisés.

C#

```
namespace System
{
    [AttributeUsage(
        AttributeTargets.Class |
        AttributeTargets.Struct |
        AttributeTargets.Enum |
        AttributeTargets.Interface |
        AttributeTargets.Delegate |
        AttributeTargets.Method |
        AttributeTargets.Constructor |
        AttributeTargets.Property |
        AttributeTargets.Field |
        AttributeTargets.Event,
        Inherited = false)
    ]
    public class ObsoleteAttribute: Attribute
    {
        public ObsoleteAttribute() {...}
        public ObsoleteAttribute(string message) {...}
        public ObsoleteAttribute(string message, bool error) {...}
        public string Message { get {...} }
        public bool IsError { get {...} }
    }
}
```

Si un programme utilise un type ou un membre qui est décoré avec l' `Obsolete` attribut, le compilateur émet un avertissement ou une erreur. Plus précisément, le compilateur émet un avertissement si aucun paramètre d'erreur n'est fourni, ou si le paramètre d'erreur est fourni et a la valeur `false` . Le compilateur émet une erreur si le paramètre d'erreur est spécifié et a la valeur `true` .

Dans l'exemple

C#

```
[Obsolete("This class is obsolete; use class B instead")]
class A
{
    public void F() {}
}

class B
{
    public void F() {}
}

class Test
{
    static void Main() {
```

```
        A a = new A();          // Warning
        a.F();
    }
}
```

la classe `A` est décorée avec l' `Obsolete` attribut. Chaque utilisation de `A` dans `Main` génère un avertissement qui comprend le message spécifié, «cette classe est obsolète ; Utilisez à la place la classe B.»

Attributs d'informations de l'appelant

À des fins telles que la journalisation et la création de rapports, il est parfois utile pour une fonction membre d'obtenir certaines informations de compilation sur le code appelant. Les attributs d'informations de l'appelant permettent de transmettre ces informations de façon transparente.

Lorsqu'un paramètre facultatif est annoté avec l'un des attributs d'informations de l'appelant, l'omission de l'argument correspondant dans un appel n'entraîne pas nécessairement la substitution de la valeur de paramètre par défaut. Au lieu de cela, si les informations spécifiées sur le contexte d'appel sont disponibles, ces informations sont passées en tant que valeur d'argument.

Par exemple :

C#

```
using System.Runtime.CompilerServices

...

public void Log(
    [CallerLineNumber] int line = -1,
    [CallerFilePath]   string path = null,
    [CallerMemberName] string name = null
)
{
    Console.WriteLine((line < 0) ? "No line" : "Line "+ line);
    Console.WriteLine((path == null) ? "No file path" : path);
    Console.WriteLine((name == null) ? "No member name" : name);
}
```

Un appel à sans arguments permet d' `Log()` imprimer le numéro de ligne et le chemin d'accès de fichier de l'appel, ainsi que le nom du membre dans lequel l'appel s'est produit.

Les attributs d'informations de l'appelant peuvent se produire sur des paramètres facultatifs n'importe où, y compris dans les déclarations de délégué. Toutefois, les attributs d'informations de l'appelant spécifique ont des restrictions sur les types des paramètres qu'ils peuvent attribuer, de sorte qu'il y aura toujours une conversion implicite d'une valeur substituée vers le type de paramètre.

C'est une erreur d'avoir le même attribut d'informations de l'appelant sur un paramètre de la partie qui définit et implémente la partie d'une déclaration de méthode partielle. Seuls les attributs d'informations de l'appelant dans la partie de définition sont appliqués, alors que les attributs d'informations de l'appelant qui se trouvent uniquement dans la partie d'implémentation sont ignorés.

Les informations de l'appelant n'affectent pas la résolution de surcharge. Étant donné que les paramètres facultatifs attribués sont toujours omis du code source de l'appelant, la résolution de surcharge ignore ces paramètres de la même façon qu'il ignore les autres paramètres facultatifs omis (résolution de surcharge).

Les informations de l'appelant ne sont substituées que lorsqu'une fonction est appelée explicitement dans le code source. Les appels implicites tels que les appels de constructeur parent implicite n'ont pas d'emplacement source et ne remplacent pas les informations de l'appelant. En outre, les appels qui sont liés de manière dynamique ne remplacent pas les informations de l'appelant. Lorsqu'un paramètre d'informations d'appelant est omis dans de tels cas, la valeur par défaut spécifiée du paramètre est utilisée à la place.

Les expressions de requête sont une exception. Celles-ci sont considérées comme des expansions syntaxiques, et si les appels qu'elles développent pour omettre les paramètres facultatifs avec les attributs d'informations de l'appelant, les informations de l'appelant sont remplacées. L'emplacement utilisé est l'emplacement de la clause de requête à partir de laquelle l'appel a été généré.

Si plusieurs attributs d'informations de l'appelant sont spécifiés sur un paramètre donné, ils sont préférés dans l'ordre suivant : `CallerLineNumber` , `CallerFilePath` , `CallerMemberName` .

Attribut CallerLineNumber

`System.Runtime.CompilerServices.CallerLineNumberAttribute` Est autorisé sur les paramètres facultatifs lorsqu'il existe une conversion implicite standard (conversions implicites standard) de la valeur de constante `int.MaxValue` vers le type du paramètre. Cela permet de s'assurer que tous les numéros de ligne non négatifs jusqu'à cette valeur peuvent être passés sans erreur.

Si un appel de fonction à partir d'un emplacement dans le code source omet un paramètre facultatif avec `CallerLineNumberAttribute` , un littéral numérique représentant le numéro de ligne de cet emplacement est utilisé comme argument de l'appel à la place de la valeur de paramètre par défaut.

Si l'appel s'étend sur plusieurs lignes, la ligne choisie dépend de l'implémentation.

Notez que le numéro de ligne peut être affecté par les `#line` directives (directives de ligne).

Attribut CallerFilePath

`System.Runtime.CompilerServices.CallerFilePathAttribute`Est autorisé sur les paramètres facultatifs lorsqu'il existe une conversion implicite standard (conversions implicites standard) de `string` vers le type du paramètre.

Si un appel de fonction à partir d'un emplacement dans le code source omet un paramètre facultatif avec `CallerFilePathAttribute` , un littéral de chaîne représentant le chemin de fichier de cet emplacement est utilisé comme argument de l'appel à la place de la valeur de paramètre par défaut.

Le format du chemin d'accès du fichier dépend de l'implémentation.

Notez que le chemin d'accès du fichier peut être affecté par les `#line` directives (directives de ligne).

Attribut CallerMemberName

`System.Runtime.CompilerServices.CallerMemberNameAttribute`Est autorisé sur les paramètres facultatifs lorsqu'il existe une conversion implicite standard (conversions implicites standard) de `string` vers le type du paramètre.

Si un appel de fonction à partir d'un emplacement dans le corps d'un membre de fonction ou d'un attribut appliqué au membre de fonction lui-même ou son type de retour, les paramètres ou les paramètres de type dans le code source omettent un paramètre facultatif avec `CallerMemberNameAttribute` , un littéral de chaîne représentant le nom de ce membre est utilisé comme argument de l'appel à la place de la valeur de paramètre par défaut

Pour les appels qui se produisent dans les méthodes génériques, seul le nom de la méthode lui-même est utilisé, sans la liste des paramètres de type.

Pour les appels qui se produisent dans les implémentations de membres d'interface explicites, seul le nom de méthode lui-même est utilisé, sans la qualification d'interface précédente.

Pour les appels qui se produisent dans les accesseurs de propriété ou d'événement, le nom de membre utilisé est celui de la propriété ou de l'événement lui-même.

Pour les appels qui se produisent dans les accesseurs d'indexeur, le nom de membre utilisé est celui fourni par un `IndexerNameAttribute` (attribut IndexerName) sur le membre de l'indexeur, s'il est présent, ou le nom par défaut dans le `Item` cas contraire.

Pour les appels qui se produisent dans les déclarations de constructeurs d'instance, les constructeurs statiques, les destructeurs et les opérateurs, le nom de membre utilisé est dépendant de l'implémentation.

Attributs pour l'interopérabilité

Remarque : cette section s'applique uniquement à l'implémentation Microsoft .NET de C#.

Interopérabilité avec les composants COM et Win32

Le Runtime .NET fournit un grand nombre d'attributs qui permettent aux programmes C# d'interagir avec les composants écrits à l'aide de DLL COM et Win32. Par exemple, l' `DllImport` attribut peut être utilisé sur une `static extern` méthode pour indiquer que l'implémentation de la méthode doit être trouvée dans une DLL Win32. Ces attributs se trouvent dans l' `System.Runtime.InteropServices` espace de noms et la documentation détaillée pour ces attributs est disponible dans la documentation du Runtime .net.

Interopérabilité avec d'autres langages .NET

Attribut IndexerName

Les indexeurs sont implémentés dans .NET à l'aide de propriétés indexées et ont un nom dans les métadonnées .NET. Si aucun `IndexerName` attribut n'est présent pour un indexeur, le nom `Item` est utilisé par défaut. L' `IndexerName` attribut permet à un développeur de remplacer cette valeur par défaut et de spécifier un autre nom.

```
C#
```

```
namespace System.Runtime.CompilerServices.CSharp
{
    [AttributeUsage(AttributeTargets.Property)]
    public class IndexerNameAttribute: Attribute
    {
        public IndexerNameAttribute(string indexerName) {...}
        public string Value { get {...} }
    }
}
```

Code unsafe

Article • 13/05/2021

Le langage C# principal, tel que défini dans les chapitres précédents, diffère notamment de C et C++ dans son omission des pointeurs en tant que type de données. Au lieu de cela, C# fournit des références et la possibilité de créer des objets qui sont gérés par un garbage collector. Cette conception, couplée à d'autres fonctionnalités, rend C# un langage plus sûr que C ou C++. Dans le langage C# principal, il n'est tout simplement pas possible d'avoir une variable non initialisée, un pointeur « non résolu » ou une expression qui indexe un tableau au-delà de ses limites. Les catégories entières de bogues qui imposent régulièrement les programmes C et C++ sont donc éliminées.

Bien que pratiquement toutes les constructions de type pointeur en C ou C++ aient un équivalent de type référence en C#, néanmoins, il existe des situations où l'accès aux types pointeur devient une nécessité. Par exemple, l'interfaçage avec le système d'exploitation sous-jacent, l'accès à un périphérique mappé en mémoire ou l'implémentation d'un algorithme de temps critique peut ne pas être possible ou pratique sans accès aux pointeurs. Pour répondre à ce besoin, C# offre la possibilité d'écrire du ***code non sécurisé***.

Dans du code non sécurisé, il est possible de déclarer et d'utiliser des pointeurs, d'effectuer des conversions entre les pointeurs et les types intégraux, de prendre l'adresse des variables, et ainsi de suite. Dans un sens, l'écriture de code non sécurisé est très similaire à l'écriture de code C dans un programme C#.

Le code non sécurisé est en fait une fonctionnalité « sécurisée » du point de vue des développeurs et des utilisateurs. Le code unsafe doit être clairement marqué avec le modificateur, de sorte que les `unsafe` développeurs ne peuvent pas utiliser accidentellement des fonctionnalités potentiellement dangereuses, et le moteur d'exécution fonctionne pour s'assurer que le code unsafe ne peut pas être exécuté dans un environnement non fiable.

Contextes non sécurisés

Les fonctionnalités non sécurisées de C# sont uniquement disponibles dans les contextes non sécurisés. Un contexte non sécurisé est introduit en incluant un `unsafe` modificateur dans la déclaration d'un type ou d'un membre, ou en utilisant un *unsafe_statement*:

- Une déclaration d'une classe, d'une structure, d'une interface ou d'un délégué peut inclure un `unsafe` modificateur, auquel cas la totalité de l'étendue textuelle de cette déclaration de type (y compris le corps de la classe, du struct ou de l'interface) est considérée comme un contexte non sécurisé.
- Une déclaration d'un champ, d'une méthode, d'une propriété, d'un événement, d'un indexeur, d'un opérateur, d'un constructeur d'instance, d'un destructeur ou d'un constructeur statique peut inclure un `unsafe` modificateur, auquel cas la totalité de l'étendue textuelle de cette déclaration de membre est considérée comme un contexte non sécurisé.
- Une *unsafe_statement* permet l'utilisation d'un contexte unsafe dans un *bloc*. La totalité de l'étendue textuelle du *bloc* associé est considérée comme un contexte non sécurisé.

Les productions grammaticales associées sont indiquées ci-dessous.

antlr

```
class_modifier_unsafe
    : 'unsafe'
    ;

struct_modifier_unsafe
    : 'unsafe'
    ;

interface_modifier_unsafe
    : 'unsafe'
    ;

delegate_modifier_unsafe
    : 'unsafe'
    ;

field_modifier_unsafe
    : 'unsafe'
    ;

method_modifier_unsafe
    : 'unsafe'
    ;

property_modifier_unsafe
    : 'unsafe'
    ;

event_modifier_unsafe
    : 'unsafe'
    ;

indexer_modifier_unsafe
```

```
    : 'unsafe'
    ;

operator_modifier_unsafe
    : 'unsafe'
    ;

constructor_modifier_unsafe
    : 'unsafe'
    ;

destructor_declaration_unsafe
    : attributes? 'extern'? 'unsafe'? '~' identifier '(' ')' destructor_body
    | attributes? 'unsafe'? 'extern'? '~' identifier '(' ')' destructor_body
    ;

static_constructor_modifiers_unsafe
    : 'extern'? 'unsafe'? 'static'
    | 'unsafe'? 'extern'? 'static'
    | 'extern'? 'static' 'unsafe'?
    | 'unsafe'? 'static' 'extern'?
    | 'static' 'extern'? 'unsafe'?
    | 'static' 'unsafe'? 'extern'?
    ;

embedded_statement_unsafe
    : unsafe_statement
    | fixed_statement
    ;

unsafe_statement
    : 'unsafe' block
    ;
```

Dans l'exemple

C#

```
public unsafe struct Node
{
    public int Value;
    public Node* Left;
    public Node* Right;
}
```

le `unsafe` modificateur spécifié dans la déclaration de struct fait en sorte que la totalité de l'étendue textuelle de la déclaration de struct devienne un contexte non sécurisé. Par conséquent, il est possible de déclarer `Left` les `Right` champs et comme étant d'un type pointeur. L'exemple ci-dessus peut également être écrit

```
```

C#

```
public struct Node
{
    public int Value;
    public unsafe Node* Left;
    public unsafe Node* Right;
}
```

Ici, les `unsafe` modificateurs dans les déclarations de champ entraînent la prise en compte de contextes non sécurisés par ces déclarations.

En dehors de l'établissement d'un contexte non sécurisé, ce qui autorise l'utilisation de types pointeur, le `unsafe` modificateur n'a aucun effet sur un type ou un membre. Dans l'exemple

C#

```
public class A
{
    public unsafe virtual void F() {
        char* p;
        ...
    }
}

public class B: A
{
    public override void F() {
        base.F();
        ...
    }
}
```

le `unsafe` modificateur sur la `F` méthode dans permet `A` simplement à l'étendue textuelle de de `F` devenir un contexte non sécurisé dans lequel les fonctionnalités non sécurisées du langage peuvent être utilisées. Dans, la substitution de `F` dans `B` , il n'est pas nécessaire de spécifier à nouveau le `unsafe` modificateur, sauf si, bien sûr, la `F` méthode dans `B` elle-même a besoin d'accéder à des fonctionnalités non sécurisées.

La situation est légèrement différente lorsqu'un type pointeur fait partie de la signature de la méthode.

C#

```
public unsafe class A
{
```

```
    public virtual void F(char* p) {...}
}

public class B: A
{
    public unsafe override void F(char* p) {...}
}
```

Ici, étant donné que `F` la signature de est incluse dans un type pointeur, elle peut uniquement être écrite dans un contexte non sécurisé. Toutefois, le contexte unsafe peut être introduit soit en rendant la classe entière non sécurisée, comme c'est le cas dans `A` , soit en incluant un `unsafe` modificateur dans la déclaration de méthode, comme c'est le cas dans `B` .

Types de pointeur

Dans un contexte unsafe, un *type* (types) peut être un *pointer_type* , ainsi qu'un *Value_type* ou un *reference_type*. Toutefois, un *pointer_type* peut également être utilisé dans une `typeof` expression (expressions de création d'objets anonymes) en dehors d'un contexte non sécurisé, car cette utilisation n'est pas fiable.

antlr

```
type_unsafe
    : pointer_type
    ;
```

Un *pointer_type* est écrit sous la forme d'un *unmanaged_type* ou du mot clé `void` , suivi d'un `*` jeton :

antlr

```
pointer_type
    : unmanaged_type '*'
    | 'void' '*'
    ;

unmanaged_type
    : type
    ;
```

Le type spécifié avant `*` dans un type pointeur est appelé le ***type référent*** du type pointeur. Il représente le type de la variable vers laquelle la valeur du type pointeur pointe.

Contrairement aux références (valeurs de types référence), les pointeurs ne sont pas suivis par le récupérateur de mémoire : le garbage collector n'a aucune connaissance des pointeurs et des données auxquelles ils pointent. Pour cette raison, un pointeur n'est pas autorisé à pointer vers une référence ou un struct qui contient des références, et le type référent d'un pointeur doit être un *unmanaged_type*.

Un *unmanaged_type* est un type qui n'est pas un type *reference_type* ou construit, et ne contient pas de champs de type *reference_type* ou construits à aucun niveau d'imbrication. En d'autres termes, un *unmanaged_type* est l'un des éléments suivants :

- `sbyte`, `byte` , `short` , `ushort` , `int` , `uint` , `long` , `ulong` , `char` , `float` , `double` , `decimal` ou `bool` .
- Tout *enum_type*.
- Tout *pointer_type*.
- Tout *struct_type* défini par l'utilisateur qui n'est pas un type construit et contient uniquement des champs de *unmanaged_type* s.

La règle intuitive pour mélanger des pointeurs et des références est que les référents de références (objets) sont autorisés à contenir des pointeurs, mais les référents de pointeurs ne sont pas autorisés à contenir des références.

Des exemples de types pointeur sont fournis dans le tableau ci-dessous :

Exemple	Description
`byte*`	Pointeur vers `byte`
`char*`	Pointeur vers `char`
`int**`	Pointeur vers un pointeur vers `int`
`int*[]`	Tableau unidimensionnel de pointeurs vers `int`
`void*`	Pointeur vers un type inconnu

Pour une implémentation donnée, tous les types de pointeurs doivent avoir la même taille et la même représentation.

Contrairement à C et C++, lorsque plusieurs pointeurs sont déclarés dans la même déclaration, en C#, `*` est écrit avec le type sous-jacent uniquement, et non comme un signe de ponctuation de préfixe sur chaque nom de pointeur. Par exemple

C#

```
int* pi, pj;    // NOT as int *pi, *pj;
```

La valeur d'un pointeur ayant `T*` le type représente l'adresse d'une variable de type `T` . L'opérateur d'indirection de pointeur `*` (indirection de pointeur) peut être utilisé pour accéder à cette variable. Par exemple, à partir d'une variable `P` de type `int*` , l'expression `*P` désigne la `int` variable trouvée à l'adresse contenue dans `P` .

À l'instar d'une référence d'objet, un pointeur peut être `null` . L'application de l'opérateur d'indirection à un `null` pointeur entraîne un comportement défini par l'implémentation. Un pointeur avec une valeur `null` est représenté par All-bits-Zero.

Le `void*` type représente un pointeur vers un type inconnu. Étant donné que le type référent est inconnu, l'opérateur d'indirection ne peut pas être appliqué à un pointeur de type `void*` et aucune opération arithmétique n'est effectuée sur ce pointeur. Toutefois, un pointeur de type `void*` peut être casté en tout autre type pointeur (et inversement).

Les types pointeur sont une catégorie distincte de types. Contrairement aux types référence et aux types valeur, les types pointeur n'héritent pas de `object` et aucune conversion n'existe entre les types pointeur et `object` . En particulier, le boxing et l'unboxing (boxing et unboxing) ne sont pas pris en charge pour les pointeurs. Toutefois, les conversions sont autorisées entre les différents types pointeur et entre les types pointeur et les types intégraux. Cela est décrit dans conversions de pointeurs.

Un *pointer_type* ne peut pas être utilisé en tant qu'argument de type (types construits), et l'inférence de type (inférence de type) échoue sur les appels de méthode génériques qui auraient pu déduire un argument de type comme étant un type pointeur.

Un *pointer_type* peut être utilisé comme type de champ volatile (champs volatiles).

Bien que les pointeurs puissent être passés en tant que `ref` `out` paramètres ou, cela peut entraîner un comportement indéfini, puisque le pointeur peut être configuré pour pointer vers une variable locale qui n'existe plus quand la méthode appelée est retournée, ou l'objet fixe auquel elle est utilisée pour pointer, n'est plus résolu. Par exemple :

C#

```
using System;

class Test
{
    static int value = 20;

    unsafe static void F(out int* pi1, ref int* pi2) {
        int i = 10;
        pi1 = &i;
```

```
        fixed (int* pj = &value) {
            // ...
            pi2 = pj;
        }
    }

    static void Main() {
        int i = 10;
        unsafe {
            int* px1;
            int* px2 = &i;

            F(out px1, ref px2);

            Console.WriteLine("*px1 = {0}, *px2 = {1}",
                *px1, *px2);    // undefined behavior
        }
    }
}
```

Une méthode peut retourner une valeur d'un certain type et ce type peut être un pointeur. Par exemple, lorsqu'un pointeur désignant une séquence contiguë de `int` s, le nombre d'éléments de cette séquence et une autre `int` valeur, la méthode suivante retourne l'adresse de cette valeur dans cette séquence, si une correspondance est trouvée ; sinon, elle retourne `null` :

C#

```
unsafe static int* Find(int* pi, int size, int value) {
    for (int i = 0; i < size; ++i) {
        if (*pi == value)
            return pi;
        ++pi;
    }
    return null;
}
```

Dans un contexte non sécurisé, plusieurs constructions sont disponibles pour l'utilisation des pointeurs :

- L' `*` opérateur peut être utilisé pour effectuer une indirection de pointeur (indirection de pointeur).
- L' `->` opérateur peut être utilisé pour accéder à un membre d'un struct via un pointeur (accès aux membres du pointeur).
- L' `[]` opérateur peut être utilisé pour indexer un pointeur (accès à un élément pointeur).

- L' `&` opérateur peut être utilisé pour obtenir l'adresse d'une variable (opérateur d'adresse).
- Les `++` `--` opérateurs et peuvent être utilisés pour incrémenter et décrémenter des pointeurs (incrément et décrémentation de pointeur).
- Les `+` `-` opérateurs et peuvent être utilisés pour effectuer des opérations arithmétiques sur les pointeurs (opérations arithmétiquessur les pointeurs).
- Les `==` `!=` opérateurs,, `<` ,, `>` `<=` et `>=` peuvent être utilisés pour comparer des pointeurs (Comparaison de pointeur).
- L' `stackalloc` opérateur peut être utilisé pour allouer de la mémoire à partir de la pile des appels (mémoires tampons de taille fixe).
- L' `fixed` instruction peut être utilisée pour corriger temporairement une variable afin que son adresse puisse être obtenue (l'instruction fixed).

Variables fixes et déplaçables

L'opérateur address-of (l'opérateur address-of) et l' `fixed` instruction (l'instruction fixed) divisent les variables en deux catégories : *__variables fixes__ _ et _ *variables déplaçables* *.

Les variables fixes résident dans des emplacements de stockage qui ne sont pas affectés par l'opération du garbage collector. (Les variables locales, les paramètres de valeur et les variables créées par des pointeurs de déréférencement sont des exemples de variables fixes.) En revanche, les variables déplaçables résident dans des emplacements de stockage qui peuvent faire l'objet d'un déplacement ou d'une suppression par le garbage collector. (Les exemples de variables déplaçables incluent des champs dans des objets et des éléments de tableaux.)

L' `&` opérateur (l'opérateur d'adresse) autorise l'obtention de l'adresse d'une variable fixe sans restrictions. Toutefois, étant donné qu'une variable déplaçable est sujette à un déplacement ou une suppression par le garbage collector, l'adresse d'une variable déplaçable peut uniquement être obtenue à l'aide d'une `fixed` instruction (l'instruction fixed) et cette adresse reste valide uniquement pour la durée de cette `fixed` instruction.

En termes précis, une variable fixe est l'un des éléments suivants :

- Variable résultant d'un *simple_name* (noms simples) qui fait référence à une variable locale ou à un paramètre de valeur, sauf si la variable est capturée par une fonction anonyme.
- Variable résultant d'un *member_access* (accès aux membres) du formulaire `V.I` , où `V` est une variable fixe d'un *struct_type*.
- Variable résultant d'un *pointer_indirection_expression* (indirection de pointeur) du formulaire, d' `*P` un *pointer_member_access* (accès aux membres du pointeur) du

formulaire `P->I` ou d'un *pointer_element_access* (accès à un élément pointeur) du formulaire `P[E]` .

Toutes les autres variables sont classées comme variables déplaçables.

Notez qu'un champ statique est classé comme une variable déplaçable. Notez également qu'un `ref` `out` paramètre ou est classé comme une variable déplaçable, même si l'argument donné pour le paramètre est une variable fixe. Enfin, Notez qu'une variable produite par le déréférencement d'un pointeur est toujours classifiée comme une variable fixe.

Conversions de pointeurs

Dans un contexte non sécurisé, l'ensemble des conversions implicites disponibles (conversions implicites) est étendu pour inclure les conversions de pointeur implicites suivantes :

- Depuis n'importe quel *pointer_type* vers le type `void*` .
- Du `null` littéral à n'importe quel *pointer_type*.

En outre, dans un contexte non sécurisé, l'ensemble des conversions explicites disponibles (conversions explicites) est étendu pour inclure les conversions de pointeur explicites suivantes :

- De tout *pointer_type* à n'importe quel autre *pointer_type*.
- De `sbyte` ,,,,, `byte` `short` `ushort` `int` `uint` , `long` ou `ulong` à n'importe quel *pointer_type*.
- De n'importe quel *pointer_type* à `sbyte` , `byte` ,, `short` `ushort` ,,, `int` `uint` `long` ou `ulong` .

Enfin, dans un contexte non sécurisé, l'ensemble de conversions implicites standard (conversions implicites standard) comprend la conversion de pointeur suivante :

- Depuis n'importe quel *pointer_type* vers le type `void*` .

Les conversions entre deux types pointeur ne modifient jamais la valeur du pointeur réel. En d'autres termes, une conversion d'un type pointeur vers un autre n'a aucun effet sur l'adresse sous-jacente donnée par le pointeur.

Quand un type pointeur est converti en un autre, si le pointeur résultant n'est pas correctement aligné pour le type pointé, le comportement n'est pas défini si le résultat est déréférencé. En général, le concept « aligné correctement » est transitif : si un pointeur vers le type `A` est correctement aligné pour un pointeur vers `B` le type, qui, à

son tour, est correctement aligné pour un pointeur vers le type `C` , un pointeur vers le type `A` est correctement aligné pour un pointeur vers le type `C` .

Considérez le cas suivant dans lequel une variable ayant un type est accessible via un pointeur vers un type différent :

C#

```
char c = 'A';
char* pc = &c;
void* pv = pc;
int* pi = (int*)pv;
int i = *pi;          // undefined
*pi = 123456;         // undefined
```

Quand un type pointeur est converti en un pointeur vers un octet, le résultat pointe vers l'octet adressé le plus bas de la variable. Les incréments successifs du résultat, jusqu'à la taille de la variable, produisent des pointeurs vers les octets restants de cette variable. Par exemple, la méthode suivante affiche chacun des huit octets d'une valeur double sous la forme d'une valeur hexadécimale :

C#

```
using System;

class Test
{
    unsafe static void Main() {
      double d = 123.456e23;
        unsafe {
           byte* pb = (byte*)&d;
            for (int i = 0; i < sizeof(double); ++i)
               Console.Write("{0:X2} ", *pb++);
            Console.WriteLine();
        }
    }
}
```

Bien entendu, la sortie générée dépend de endianness.

Les mappages entre les pointeurs et les entiers sont définis par l'implémentation. Toutefois, sur les architectures d'UC 32 * et 64 bits avec un espace d'adressage linéaire, les conversions de pointeurs vers ou depuis les types intégraux se comportent en général exactement comme des conversions de `uint` `ulong` valeurs ou, respectivement, vers ou à partir de ces types intégraux.

Tableaux de pointeurs

Dans un contexte non sécurisé, des tableaux de pointeurs peuvent être construits. Seules certaines des conversions qui s'appliquent à d'autres types de tableau sont autorisées sur les tableaux de pointeurs :

- La conversion de référence implicite (conversions de référence implicites) à partir de n'importe quelle *array_type* vers `System.Array` et les interfaces qu'elle implémente s'applique également aux tableaux de pointeurs. Toutefois, toute tentative d'accès aux éléments du tableau via `System.Array` ou les interfaces qu'il implémente entraîne une exception au moment de l'exécution, car les types de pointeurs ne sont pas convertibles en `object` .
- Les conversions de référence implicite et explicite (conversions deréférence implicites, conversions de référence explicites) d'un type de tableau unidimensionnel `S[]` en `System.Collections.Generic.IList<T>` et ses interfaces de base génériques ne s'appliquent jamais aux tableaux de pointeurs, puisque les types pointeur ne peuvent pas être utilisés en tant qu'arguments de type et qu'il n'y a pas de conversion de types pointeur en
- La conversion de référence explicite (conversions de références explicites) à partir de `System.Array` et des interfaces qu'elle implémente à tout *array_type* s'applique aux tableaux de pointeurs.
- Les conversions de référence explicites (conversions de référence explicites) de `System.Collections.Generic.IList<S>` et ses interfaces de base en un type de tableau unidimensionnel ne `T[]` s'appliquent jamais aux tableaux de pointeurs, car les types pointeur ne peuvent pas être utilisés en tant qu'arguments de type, et il n'y a aucune conversion des types pointeur vers les types non pointeur.

Ces restrictions signifient que l'expansion de l' `foreach` instruction sur les tableaux décrite dans l'instruction foreach ne peut pas être appliquée aux tableaux de pointeurs. Au lieu de cela, une instruction foreach de la forme

C#

```
foreach (V v in x) embedded_statement
```

où le type de `x` est un type de tableau de la forme `T[,,...,]` , `N` est le nombre de dimensions moins 1 et `T` ou `V` est un type pointeur, est développé à l'aide de boucles for imbriquées comme suit :

C#

```
{
    T[,,...,] a = x;
    for (int i0 = a.GetLowerBound(0); i0 <= a.GetUpperBound(0); i0++)
    for (int i1 = a.GetLowerBound(1); i1 <= a.GetUpperBound(1); i1++)
    ...
    for (int iN = a.GetLowerBound(N); iN <= a.GetUpperBound(N); iN++) {
        V v = (V)a.GetValue(i0,i1,...,iN);
        embedded_statement
    }
}
```

Les variables `a` , `i0` , `i1` ,..., ne `iN` sont pas visibles par ou accessibles à `x` ou le *embedded_statement* ou tout autre code source du programme. La variable `v` est en lecture seule dans l'instruction incorporée. S'il n'existe pas de conversion explicite (conversions de pointeur) de `T` (le type d'élément) en `V` , une erreur est générée et aucune autre étape n'est effectuée. Si `x` a la valeur `null` , une `System.NullReferenceException` est levée au moment de l'exécution.

Pointeurs dans les expressions

Dans un contexte unsafe, une expression peut produire un résultat d'un type pointeur, mais en dehors d'un contexte unsafe, il s'agit d'une erreur de compilation pour qu'une expression soit d'un type pointeur. En termes précis, en dehors d'un contexte non sécurisé, une erreur de compilation se produit si une *simple_name* (noms simples), *member_access* (accès aux membres), *invocation_expression* (expressions d'appel) ou *element_access* (accès à l'élément) est d'un type pointeur.

Dans un contexte non sécurisé, les productions *primary_no_array_creation_expression* (expressions primaires) et *unary_expression* (opérateurs unaires) autorisent les constructions supplémentaires suivantes :

antlr

```
primary_no_array_creation_expression_unsafe
    : pointer_member_access
    | pointer_element_access
    | sizeof_expression
    ;

unary_expression_unsafe
    : pointer_indirection_expression
    | addressof_expression
    ;
```

Ces constructions sont décrites dans les sections suivantes. La priorité et l'associativité des opérateurs non sécurisés sont implicites par la grammaire.

Indirection de pointeur

Un *pointer_indirection_expression* se compose d'un astérisque (`*`) suivi d'un *unary_expression*.

antlr

```
pointer_indirection_expression
    : '*' unary_expression
    ;
```

L'opérateur unaire `*` dénote l'indirection de pointeur et est utilisé pour obtenir la variable vers laquelle pointe un pointeur. Le résultat de l'évaluation de `*P` , où `P` est une expression d'un type pointeur `T*` , est une variable de type `T` . Il s'agit d'une erreur au moment de la compilation pour appliquer l'opérateur unaire `*` à une expression de type `void*` ou à une expression qui n'est pas d'un type pointeur.

L'effet de l'application de l'opérateur unaire `*` à un `null` pointeur est défini par l'implémentation. En particulier, il n'y a aucune garantie que cette opération lève une `System.NullReferenceException` .

Si une valeur non valide a été assignée au pointeur, le comportement de l'opérateur unaire `*` n'est pas défini. Parmi les valeurs non valides pour le déréférencement d'un pointeur par l'opérateur unaire `*` , une adresse est alignée de manière inappropriée pour le type pointé (Voir l'exemple dans les conversions de pointeur) et l'adresse d'une variable après la fin de sa durée de vie.

À des fins d'analyse d'assignation définie, une variable produite en évaluant une expression de la forme `*P` est considérée comme initialement assignée (variables initialement affectées).

Accès aux membres de pointeur

Un *pointer_member_access* se compose d'un *primary_expression*, suivi d'un `->` jeton «», suivi d'un *identificateur* et d'un *type_argument_list* facultatif.

antlr

```
pointer_member_access
    : primary_expression '->' identifier
    ;
```

Dans un accès de membre de pointeur du formulaire `P->I` , `P` doit être une expression d'un type pointeur autre que `void*` , et `I` doit indiquer un membre accessible du type vers lequel `P` pointe.

Un accès aux membres d'un pointeur du formulaire `P->I` est évalué exactement comme `(*P).I` . Pour obtenir une description de l'opérateur d'indirection de pointeur (`*`), consultez indirection de pointeur. Pour obtenir une description de l'opérateur d'accès aux membres (`.`), consultez accès aux membres.

Dans l'exemple

C#

```
using System;

struct Point
{
    public int x;
    public int y;

    public override string ToString() {
        return "(" + x + "," + y + ")";
    }
}

class Test
{
    static void Main() {
        Point point;
        unsafe {
            Point* p = &point;
            p->x = 10;
            p->y = 20;
            Console.WriteLine(p->ToString());
        }
    }
}
```

l' `->` opérateur est utilisé pour accéder aux champs et appeler une méthode d'un struct via un pointeur. Étant donné que l'opération `P->I` est exactement équivalente à `(*P).I` , la `Main` méthode peut également avoir été écrite :

C#

```
class Test
{
    static void Main() {
        Point point;
        unsafe {
            Point* p = &point;
            (*p).x = 10;
            (*p).y = 20;
            Console.WriteLine((*p).ToString());
        }
    }
}
```

Accès aux éléments de pointeur

Une *pointer_element_access* se compose d'une *primary_no_array_creation_expression* suivie d'une expression placée entre « `[` » et «» `]` .

antlr

```
pointer_element_access
    : primary_no_array_creation_expression '[' expression ']'
    ;
```

Dans un accès à un élément pointeur du formulaire `P[E]` , `P` doit être une expression d'un type pointeur autre que `void*` , et `E` doit être une expression qui peut être convertie implicitement en `int` , `uint` , `long` ou `ulong` .

Un accès à un élément de pointeur du formulaire `P[E]` est évalué exactement comme `*(P + E)` . Pour obtenir une description de l'opérateur d'indirection de pointeur (`*`), consultez indirection de pointeur. Pour obtenir une description de l'opérateur d'addition de pointeurs (`+`), consultez l' opération arithmétique de pointeur.

Dans l'exemple

C#

```
class Test
{
    static void Main() {
        unsafe {
            char* p = stackalloc char[256];
            for (int i = 0; i < 256; i++) p[i] = (char)i;
        }
    }
}
```

un accès à un élément pointeur est utilisé pour initialiser la mémoire tampon de caractères dans une `for` boucle. Étant donné que l'opération `P[E]` est exactement équivalente à `*(P + E)` , l'exemple peut également avoir été écrit :

C#

```
class Test
{
    static void Main() {
        unsafe {
            char* p = stackalloc char[256];
            for (int i = 0; i < 256; i++) *(p + i) = (char)i;
        }
    }
}
```

L'opérateur d'accès à l'élément pointeur ne vérifie pas les erreurs de limites hors limites et le comportement lors de l'accès à un élément hors limites n'est pas défini. C'est le même que C et C++.

address-of, opérateur

Une *addressof_expression* se compose d'un et commercial (`&`) suivi d'un *unary_expression*.

antlr

```
addressof_expression
    : '&' unary_expression
    ;
```

Si une expression `E` est d'un type `T` et est classée en tant que variable fixe (variables fixes et déplaçables), la construction `&E` calcule l'adresse de la variable fournie par `E` . Le type du résultat est `T*` et est classifié comme une valeur. Une erreur de compilation se produit si `E` n'est pas classifié comme une variable, si `E` est classifiée comme une variable locale en lecture seule, ou si indique `E` une variable déplaçable. Dans le dernier cas, une instruction fixed (l'instruction fixed) peut être utilisée pour « corriger » temporairement la variable avant d'obtenir son adresse. Comme indiqué dans l'accès aux membres, en dehors d'un constructeur d'instance ou d'un constructeur statique pour un struct ou une classe qui définit un `readonly` champ, ce champ est considéré comme une valeur, et non comme une variable. Par conséquent, son adresse ne peut pas être prise. De même, l'adresse d'une constante ne peut pas être prise.

L' `&` opérateur ne requiert pas l'assignation définitive de son argument, mais après une `&` opération, la variable à laquelle l'opérateur est appliqué est considérée comme assignée de manière définitive dans le chemin d'exécution dans lequel l'opération se produit. Il incombe au programmeur de s'assurer que l'initialisation correcte de la variable a réellement lieu dans cette situation.

Dans l'exemple

C#

```
using System;

class Test
{
    static void Main() {
        int i;
        unsafe {
            int* p = &i;
            *p = 123;
        }
        Console.WriteLine(i);
    }
}
```

`i` est considéré comme étant définitivement affecté à la suite de l' `&i` opération utilisée pour initialiser `p` . L'assignation à `*p` en effet initialise `i` , mais l'inclusion de cette initialisation est la responsabilité du programmeur, et aucune erreur de compilation ne se produit si l'assignation a été supprimée.

Les règles d'attribution définie pour l' `&` opérateur existent, de sorte que l'initialisation redondante des variables locales peut être évitée. Par exemple, de nombreuses API externes prennent un pointeur vers une structure qui est remplie par l'API. Les appels à ces API passent généralement l'adresse d'une variable de struct locale et sans la règle, l'initialisation redondante de la variable de struct est nécessaire.

Incrémenter et décrémenter des pointeurs

Dans un contexte non sécurisé, les `++` `--` opérateurs et (opérateurs suffixés d'incrémentation et de décrémentation et opérateurs d' incrémentation et de décrémentation de préfixe) peuvent être appliqués aux variables pointeur de tous les types sauf `void*` . Ainsi, pour chaque type de pointeur `T*` , les opérateurs suivants sont définis implicitement :

C#

```
T* operator ++(T* x);
T* operator --(T* x);
```

Les opérateurs produisent les mêmes résultats que `x + 1` et `x - 1` , respectivement (arithmétique de pointeur). En d'autres termes, pour une variable pointeur de type `T*` , l' `++` opérateur ajoute `sizeof(T)` à l'adresse contenue dans la variable, et l' `--` opérateur soustrait `sizeof(T)` de l'adresse contenue dans la variable.

Si une opération d'incrémentation ou de décrémentation de pointeur dépasse le domaine du type pointeur, le résultat est défini par l'implémentation, mais aucune exception n'est générée.

Arithmétique des pointeurs

Dans un contexte non sécurisé, les `+` opérateurs et (opérateur d' `-` addition et opérateur de soustraction) peuvent être appliqués aux valeurs de tous les types pointeur, à l'exception de `void*` . Ainsi, pour chaque type de pointeur `T*` , les opérateurs suivants sont définis implicitement :

C#

```
T* operator +(T* x, int y);
T* operator +(T* x, uint y);
T* operator +(T* x, long y);
T* operator +(T* x, ulong y);

T* operator +(int x, T* y);
T* operator +(uint x, T* y);
T* operator +(long x, T* y);
T* operator +(ulong x, T* y);

T* operator -(T* x, int y);
T* operator -(T* x, uint y);
T* operator -(T* x, long y);
T* operator -(T* x, ulong y);

long operator -(T* x, T* y);
```

En fonction d'une expression `P` d'un type pointeur `T*` et d'une expression `N` de type `int` , `uint` , `long` ou `ulong` , les expressions `P + N` et `N + P` calculent la valeur de pointeur de type `T*` qui résulte de l'ajout `N * sizeof(T)` à l'adresse fournie par `P` . De même, l'expression `P - N` calcule la valeur de pointeur de type `T*` qui résulte de `N * sizeof(T)` la soustraction de l'adresse spécifiée par `P` .

À partir de deux expressions, `P` et `Q` , d'un type pointeur `T*` , l'expression `P - Q` calcule la différence entre les adresses fournies par `P` et, `Q` puis divise cette différence par `sizeof(T)` . Le type du résultat est Always `long` . En effet, `P - Q` est calculé comme `((long)(P) - (long)(Q)) / sizeof(T)` .

Par exemple :

C#

```
using System;

class Test
{
    static void Main() {
        unsafe {
            int* values = stackalloc int[20];
            int* p = &values[1];
            int* q = &values[15];
            Console.WriteLine("p - q = {0}", p - q);
            Console.WriteLine("q - p = {0}", q - p);
        }
    }
}
```

ce qui génère la sortie :

Console

```
p - q = -14
q - p = 14
```

Si une opération arithmétique de pointeur dépasse le domaine du type pointeur, le résultat est tronqué en mode d'implémentation défini, mais aucune exception n'est générée.

Comparaison de pointeurs

Dans un contexte non sécurisé, les `==` `!=` opérateurs,,, `<` `>` , `<=` et `=>` (opérateurs relationnels et de test de type) peuvent être appliqués aux valeurs de tous les types pointeur. Les opérateurs de comparaison de pointeurs sont les suivants :

C#

```
bool operator ==(void* x, void* y);
bool operator !=(void* x, void* y);
bool operator <(void* x, void* y);
```

```
bool operator >(void* x, void* y);
bool operator <=(void* x, void* y);
bool operator >=(void* x, void* y);
```

Étant donné qu'une conversion implicite existe de tout type pointeur vers le `void*` type, les opérandes de tout type pointeur peuvent être comparés à l'aide de ces opérateurs. Les opérateurs de comparaison comparent les adresses données par les deux opérandes comme s'il s'agissait d'entiers non signés.

Opérateur sizeof

L'opérateur `sizeof` retourne le nombre d'octets occupés par une variable d'un type donné. Le type spécifié comme opérande de `sizeof` doit être un *unmanaged_type* (types pointeur).

antlr

```
sizeof_expression
    : 'sizeof' '(' unmanaged_type ')'
    ;
```

Le résultat de l' `sizeof` opérateur est une valeur de type `int` . Pour certains types prédéfinis, l' `sizeof` opérateur produit une valeur constante, comme indiqué dans le tableau ci-dessous.

Expression	Résultat
`sizeof(sbyte)`	`1`
`sizeof(byte)`	`1`
`sizeof(short)`	`2`
`sizeof(ushort)`	`2`
`sizeof(int)`	`4`
`sizeof(uint)`	`4`
`sizeof(long)`	`8`
`sizeof(ulong)`	`8`
`sizeof(char)`	`2`
`sizeof(float)`	`4`

Expression	Résultat
`sizeof(double)`	`8`
`sizeof(bool)`	`1`

Pour tous les autres types, le résultat de l' `sizeof` opérateur est défini par l'implémentation et est classifié comme une valeur, et non comme une constante.

L'ordre dans lequel les membres sont empaquetés dans un struct n'est pas spécifié.

À des fins d'alignement, il peut y avoir un remplissage sans nom au début d'un struct, dans un struct et à la fin du struct. Le contenu des bits utilisés comme remplissage est indéterminé.

En cas d'application à un opérande qui a le type struct, le résultat est le nombre total d'octets dans une variable de ce type, y compris tout remplissage.

fixed, instruction

Dans un contexte non sécurisé, la *embedded_statement* (instructions) production autorise une construction supplémentaire, l' `fixed` instruction, qui est utilisée pour « corriger » une variable déplaçable de telle sorte que son adresse reste constante pendant toute la durée de l'instruction.

antlr

```
fixed_statement
    : 'fixed' '(' pointer_type fixed_pointer_declarators ')'
embedded_statement
    ;

fixed_pointer_declarators
    : fixed_pointer_declarator (','  fixed_pointer_declarator)*
    ;

fixed_pointer_declarator
    : identifier '=' fixed_pointer_initializer
    ;

fixed_pointer_initializer
    : '&' variable_reference
    | expression
    ;
```

Chaque *fixed_pointer_declarator* déclare une variable locale du *pointer_type* donné et initialise cette variable locale avec l'adresse calculée par le *fixed_pointer_initializer* correspondant. Une variable locale déclarée dans une `fixed` instruction est accessible dans les *fixed_pointer_initializer* s se trouvant à droite de la déclaration de cette variable et dans le *embedded_statement* de l' `fixed` instruction. Une variable locale déclarée par une `fixed` instruction est considérée comme étant en lecture seule. Une erreur de compilation se produit si l'instruction incorporée tente de modifier cette variable locale (par le biais de l'assignation ou des `++` `--` opérateurs et) ou de la passer en tant que `ref` `out` paramètre ou.

Un *fixed_pointer_initializer* peut être l'un des éléments suivants :

- Le jeton « `&` » suivi d'un *variable_reference* (règles précises pour déterminer l'assignation définie) à une variable déplaçable (variables fixes et déplaçables) d'un type non managé `T` , à condition que le type `T*` soit implicitement convertible en type pointeur donné dans l' `fixed` instruction. Dans ce cas, l'initialiseur calcule l'adresse de la variable donnée, et la variable est garantie pour rester à une adresse fixe pendant toute la durée de l' `fixed` instruction.
- Expression d'un *array_type* avec des éléments d'un type non managé `T` , à condition que le type `T*` soit implicitement convertible en type pointeur donné dans l' `fixed` instruction. Dans ce cas, l'initialiseur calcule l'adresse du premier élément du tableau, et le tableau entier est garanti qu'il reste à une adresse fixe pendant toute la durée de l' `fixed` instruction. Si l'expression de tableau a la valeur null ou si le tableau n'a aucun élément, l'initialiseur calcule une adresse égale à zéro.
- Expression de type `string` , à condition que le type `char*` soit implicitement convertible en type pointeur donné dans l' `fixed` instruction. Dans ce cas, l'initialiseur calcule l'adresse du premier caractère de la chaîne, et la chaîne entière est garantie pour rester à une adresse fixe pendant toute la durée de l' `fixed` instruction. Le comportement de l' `fixed` instruction est défini par l'implémentation si l'expression de chaîne est null.
- *Simple_name* ou *member_access* qui référence un membre de mémoire tampon de taille fixe d'une variable déplaçable, à condition que le type du membre de la mémoire tampon de taille fixe soit implicitement convertible en type pointeur donné dans l' `fixed` instruction. Dans ce cas, l'initialiseur calcule un pointeur vers le premier élément de la mémoire tampon de taille fixe (mémoires tampons de taille fixe dans les expressions), et la mémoire tampon de taille fixe reste toujours à une adresse fixe pour la durée de l' `fixed` instruction.

Pour chaque adresse calculée par un *fixed_pointer_initializer* l' `fixed` instruction garantit que la variable référencée par l'adresse n'est pas sujette à un déplacement ou une suppression par le garbage collector pendant la durée de l' `fixed` instruction. Par exemple, si l'adresse calculée par un *fixed_pointer_initializer* fait référence à un champ d'un objet ou à un élément d'une instance de tableau, l' `fixed` instruction garantit que l'instance d'objet conteneur n'est pas déplacée ou supprimée pendant la durée de vie de l'instruction.

Il incombe au programmeur de s'assurer que les pointeurs créés par des `fixed` instructions ne survivent pas au-delà de l'exécution de ces instructions. Par exemple, lorsque des pointeurs créés par des `fixed` instructions sont passés à des API externes, il incombe au programmeur de s'assurer que les API ne conservent aucune mémoire de ces pointeurs.

Les objets fixes peuvent entraîner la fragmentation du tas (car ils ne peuvent pas être déplacés). Pour cette raison, les objets doivent être corrigés uniquement lorsque cela est absolument nécessaire, puis uniquement pour la durée la plus faible possible.

L'exemple

C#

```
class Test
{
    static int x;
    int y;

    unsafe static void F(int* p) {
        *p = 1;
    }

    static void Main() {
        Test t = new Test();
        int[] a = new int[10];
        unsafe {
            fixed (int* p = &x) F(p);
            fixed (int* p = &t.y) F(p);
            fixed (int* p = &a[0]) F(p);
            fixed (int* p = a) F(p);
        }
    }
}
```

illustre plusieurs utilisations de l' `fixed` instruction. La première instruction corrige et obtient l'adresse d'un champ statique, la deuxième instruction corrige et obtient l'adresse d'un champ d'instance, et la troisième instruction corrige et obtient l'adresse

d'un élément de tableau. Dans chaque cas, il y aurait eu une erreur à utiliser l' `&` opérateur normal, car les variables sont toutes classées comme variables déplaçables.

La quatrième `fixed` instruction de l'exemple ci-dessus produit un résultat similaire à la troisième.

Cet exemple de l' `fixed` instruction utilise `string` :

C#

```
class Test
{
    static string name = "xx";

    unsafe static void F(char* p) {
        for (int i = 0; p[i] != '\0'; ++i)
            Console.WriteLine(p[i]);
    }

    static void Main() {
        unsafe {
            fixed (char* p = name) F(p);
            fixed (char* p = "xx") F(p);
        }
    }
}
```

Dans un tableau de contexte non sécurisé, les éléments des tableaux unidimensionnels sont stockés dans l'ordre d'index de plus en plus important, en commençant par l'index `0` et en terminant par l'index `Length - 1` . Pour les tableaux multidimensionnels, les éléments de tableau sont stockés de telle sorte que les index de la dimension la plus à droite soient augmentés en premier, puis la dimension gauche suivante, et ainsi de suite vers la gauche. Dans une `fixed` instruction qui obtient un pointeur `p` vers une instance de tableau `a` , les valeurs de pointeur allant de `p` pour `p + a.Length - 1` représenter les adresses des éléments dans le tableau. De même, les variables allant de `p[0]` pour `p[a.Length - 1]` représenter les éléments de tableau réels. Étant donné la façon dont les tableaux sont stockés, nous pouvons traiter un tableau de toute dimension comme s'il était linéaire.

Par exemple :

C#

```
using System;

class Test
{
```

```
    static void Main() {
        int[,,] a = new int[2,3,4];
        unsafe {
            fixed (int* p = a) {
                for (int i = 0; i < a.Length; ++i)    // treat as linear
                    p[i] = i;
            }
        }

        for (int i = 0; i < 2; ++i)
            for (int j = 0; j < 3; ++j) {
                for (int k = 0; k < 4; ++k)
                    Console.Write("[{0},{1},{2}] = {3,2} ", i, j, k,
a[i,j,k]);
                Console.WriteLine();
            }
    }
}
```

ce qui génère la sortie :

Console

```
[0,0,0] =  0 [0,0,1] =  1 [0,0,2] =  2 [0,0,3] =  3
[0,1,0] =  4 [0,1,1] =  5 [0,1,2] =  6 [0,1,3] =  7
[0,2,0] =  8 [0,2,1] =  9 [0,2,2] = 10 [0,2,3] = 11
[1,0,0] = 12 [1,0,1] = 13 [1,0,2] = 14 [1,0,3] = 15
[1,1,0] = 16 [1,1,1] = 17 [1,1,2] = 18 [1,1,3] = 19
[1,2,0] = 20 [1,2,1] = 21 [1,2,2] = 22 [1,2,3] = 23
```

Dans l'exemple

C#

```
class Test
{
    unsafe static void Fill(int* p, int count, int value) {
        for (; count != 0; count--) *p++ = value;
    }

    static void Main() {
        int[] a = new int[100];
        unsafe {
            fixed (int* p = a) Fill(p, 100, -1);
        }
    }
}
```

une `fixed` instruction est utilisée pour corriger un tableau afin que son adresse puisse être passée à une méthode qui prend un pointeur.

Dans l'exemple :

```C#
unsafe struct Font
{
    public int size;
    public fixed char name[32];
}

class Test
{
    unsafe static void PutString(string s, char* buffer, int bufSize) {
        int len = s.Length;
        if (len > bufSize) len = bufSize;
        for (int i = 0; i < len; i++) buffer[i] = s[i];
        for (int i = len; i < bufSize; i++) buffer[i] = (char)0;
    }

    Font f;

    unsafe static void Main()
    {
        Test test = new Test();
        test.f.size = 10;
        fixed (char* p = test.f.name) {
            PutString("Times New Roman", p, 32);
        }
    }
}
```

une instruction fixed est utilisée pour corriger une mémoire tampon de taille fixe d'un struct afin que son adresse puisse être utilisée comme pointeur.

Une `char*` valeur produite par la résolution d'une instance de chaîne pointe toujours vers une chaîne terminée par le caractère null. Dans une instruction fixed qui obtient un pointeur `p` vers une instance de chaîne `s` , les valeurs de pointeur allant de `p` pour représenter les `p + s.Length - 1` adresses des caractères de la chaîne, et la valeur de pointeur `p + s.Length` pointe toujours vers un caractère null (le caractère avec la valeur `'\0'`).

La modification d'objets de type managé par le biais de pointeurs fixes peut engendrer un comportement indéfini. Par exemple, étant donné que les chaînes sont immuables, il incombe au programmeur de s'assurer que les caractères référencés par un pointeur vers une chaîne fixe ne sont pas modifiés.

L'arrêt automatique des chaînes de caractères est particulièrement pratique lors de l'appel d'API externes qui attendent des chaînes de style C. Notez, toutefois, qu'une

instance de chaîne est autorisée à contenir des caractères null. Si ces caractères Null sont présents, la chaîne apparaît tronquée lorsqu'elle est traitée comme un terminé par le caractère null `char*` .

Mémoires tampons de taille fixe

Les mémoires tampons de taille fixe sont utilisées pour déclarer des tableaux en ligne « C style » comme membres de structs, et sont principalement utiles pour l'interfaçage avec les API non managées.

Déclarations de mémoire tampon de taille fixe

Une ***mémoire tampon de taille fixe*** est un membre qui représente le stockage pour une mémoire tampon de longueur fixe des variables d'un type donné. Une déclaration de mémoire tampon de taille fixe introduit une ou plusieurs mémoires tampons de taille fixe d'un type d'élément donné. Les mémoires tampons de taille fixe sont autorisées uniquement dans les déclarations de struct et peuvent se produire uniquement dans des contextes non sécurisés (contextes non sécurisés).

antlr

```
struct_member_declaration_unsafe
    : fixed_size_buffer_declaration
    ;

fixed_size_buffer_declaration
    : attributes? fixed_size_buffer_modifier* 'fixed' buffer_element_type
fixed_size_buffer_declarator+ ';'
    ;

fixed_size_buffer_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'unsafe'
    ;

buffer_element_type
    : type
    ;

fixed_size_buffer_declarator
    : identifier '[' constant_expression ']'
    ;
```

Une déclaration de mémoire tampon de taille fixe peut inclure un jeu d'attributs (attributs), un `new` modificateur (modificateurs), une combinaison valide des quatre modificateurs d'accès (paramètres de type et contraintes) et un `unsafe` modificateur (contextes non sécurisés). Les attributs et les modificateurs s'appliquent à tous les membres déclarés par la déclaration de mémoire tampon de taille fixe. Il s'agit d'une erreur pour que le même modificateur apparaisse plusieurs fois dans une déclaration de mémoire tampon de taille fixe.

Une déclaration de mémoire tampon de taille fixe n'est pas autorisée à inclure le `static` modificateur.

Le type d'élément de mémoire tampon d'une déclaration de mémoire tampon de taille fixe spécifie le type d'élément de la ou des mémoires tampons introduites par la déclaration. Le type d'élément de mémoire tampon doit être l'un des types prédéfinis,,,,,, `sbyte` `byte` ,,,, `short` `ushort` `int` `uint` `long` `ulong` `char` `float` `double` ou `bool` .

Le type d'élément de mémoire tampon est suivi d'une liste de déclarateurs de mémoires tampons de taille fixe, chacun d'entre eux introduisant un nouveau membre. Un déclarateur de mémoire tampon de taille fixe se compose d'un identificateur qui nomme le membre, suivi d'une expression constante placée entre `[` des `]` jetons et. L'expression constante indique le nombre d'éléments dans le membre introduit par ce déclarateur de mémoire tampon de taille fixe. Le type de l'expression constante doit être implicitement convertible en type `int` , et la valeur doit être un entier positif différent de zéro.

Il est garanti que les éléments d'une mémoire tampon de taille fixe sont disposés de façon séquentielle dans la mémoire.

Une déclaration de mémoire tampon de taille fixe qui déclare plusieurs mémoires tampons de taille fixe équivaut à plusieurs déclarations d'une déclaration de mémoire tampon de taille fixe unique avec les mêmes attributs et types d'éléments. Par exemple

C#

```
unsafe struct A
{
   public fixed int x[5], y[10], z[100];
}
```

équivaut à :

C#

```
unsafe struct A
{
   public fixed int x[5];
   public fixed int y[10];
   public fixed int z[100];
}
```

Mémoires tampons de taille fixe dans les expressions

La recherche de membre (opérateurs) d'un membre de mémoire tampon de taille fixe se déroule exactement comme la recherche de membres d'un champ.

Une mémoire tampon de taille fixe peut être référencée dans une expression à l'aide d'un *simple_name* (inférence de type) ou d'une *member_access* (vérification au moment de la compilation de la résolution de surcharge dynamique).

Quand un membre de la mémoire tampon de taille fixe est référencé comme un nom simple, l'effet est le même qu'un accès au membre du formulaire `this.I` , où `I` est le membre de la mémoire tampon de taille fixe.

Dans un accès membre de la forme `E.I` , si `E` est d'un type struct et qu'une recherche de membre de `I` dans ce type struct identifie un membre de taille fixe, `E.I` est évaluée comme suit :

- Si l'expression `E.I` ne se produit pas dans un contexte non sécurisé, une erreur de compilation se produit.
- Si `E` est classifié comme une valeur, une erreur de compilation se produit.
- Sinon, si `E` est une variable déplaçable (variables fixes et déplaçables) et que l'expression `E.I` n'est pas une *fixed_pointer_initializer* (l'instruction fixed), une erreur de compilation se produit.
- Dans le cas contraire, `E` fait référence à une variable fixe et le résultat de l'expression est un pointeur vers le premier élément du membre de la mémoire tampon de taille fixe `I` dans `E` . Le résultat est de type `S*` , où `S` est le type d'élément de `I` et est classifié comme une valeur.

Les éléments suivants de la mémoire tampon de taille fixe sont accessibles à l'aide d'opérations de pointeur à partir du premier élément. Contrairement à l'accès aux tableaux, l'accès aux éléments d'une mémoire tampon de taille fixe est une opération risquée et l'option n'est pas cochée.

L'exemple suivant déclare et utilise un struct avec un membre de mémoire tampon de taille fixe.

C#

```
unsafe struct Font
{
    public int size;
    public fixed char name[32];
}

class Test
{
    unsafe static void PutString(string s, char* buffer, int bufSize) {
        int len = s.Length;
        if (len > bufSize) len = bufSize;
        for (int i = 0; i < len; i++) buffer[i] = s[i];
        for (int i = len; i < bufSize; i++) buffer[i] = (char)0;
    }

    unsafe static void Main()
    {
        Font f;
        f.size = 10;
        PutString("Times New Roman", f.name, 32);
    }
}
```

Vérification de l'attribution définie

Les mémoires tampons de taille fixe ne sont pas soumises à la vérification de l'assignation définie (assignation définie) et les membres de la mémoire tampon de taille fixe sont ignorés à des fins de vérification de l'assignation définie des variables de type struct.

Lorsque la variable de struct conteneur la plus externe d'un membre de mémoire tampon de taille fixe est une variable statique, une variable d'instance d'une instance de classe ou un élément de tableau, les éléments de la mémoire tampon de taille fixe sont automatiquement initialisés à leurs valeurs par défaut (valeurs par défaut). Dans tous les autres cas, le contenu initial d'une mémoire tampon de taille fixe n'est pas défini.

Allocation de piles

Dans un contexte non sécurisé, une déclaration de variable locale (déclarations de variable locale) peut inclure un initialiseur d'allocation de pile qui alloue de la mémoire à partir de la pile des appels.

antlr

```
local_variable_initializer_unsafe
    : stackalloc_initializer
    ;

stackalloc_initializer
    : 'stackalloc' unmanaged_type '[' expression ']'
    ;
```

Le *unmanaged_type* indique le type des éléments qui seront stockés dans l'emplacement nouvellement alloué, et l' *expression* indique le nombre de ces éléments. Prises ensemble, elles spécifient la taille d'allocation requise. Étant donné que la taille d'une allocation de pile ne peut pas être négative, il s'agit d'une erreur de compilation pour spécifier le nombre d'éléments sous la forme d'un *constant_expression* qui correspond à une valeur négative.

Un initialiseur d'allocation de pile de la forme `stackalloc T[E]` doit `T` être un type non managé (types pointeur) et `E` être une expression de type `int` . La construction alloue `E * sizeof(T)` des octets de la pile des appels et retourne un pointeur, de type `T*` , au bloc nouvellement alloué. Si `E` est une valeur négative, le comportement n'est pas défini. Si `E` est égal à zéro, aucune allocation n'est effectuée et le pointeur retourné est défini par l'implémentation. Si la mémoire disponible est insuffisante pour allouer un bloc de la taille donnée, une `System.StackOverflowException` exception est levée.

Le contenu de la mémoire nouvellement allouée n'est pas défini.

Les initialiseurs d'allocation de pile ne sont pas autorisés dans `catch` `finally` les blocs ou (instruction try).

Il n'existe aucun moyen de libérer explicitement la mémoire allouée à l'aide de `stackalloc` . Tous les blocs de mémoire alloués par la pile créés lors de l'exécution d'un membre de fonction sont automatiquement ignorés quand ce membre de fonction retourne. Cela correspond à la `alloca` fonction, une extension couramment trouvée dans les implémentations C et C++.

Dans l'exemple

C#

```
using System;

class Test
{
    static string IntToString(int value) {
        int n = value >= 0? value: -value;
        unsafe {
```

```
            char* buffer = stackalloc char[16];
            char* p = buffer + 16;
            do {
                *--p = (char)(n % 10 + '0');
                n /= 10;
            } while (n != 0);
            if (value < 0) *--p = '-';
            return new string(p, 0, (int)(buffer + 16 - p));
        }
    }

    static void Main() {
        Console.WriteLine(IntToString(12345));
        Console.WriteLine(IntToString(-999));
    }
}
```

un `stackalloc` initialiseur est utilisé dans la `IntToString` méthode pour allouer une mémoire tampon de 16 caractères sur la pile. La mémoire tampon est automatiquement supprimée lorsque la méthode est retournée.

Allocation de mémoire dynamique

À l'exception de l' `stackalloc` opérateur, C# ne fournit aucune construction prédéfinie pour la gestion de la mémoire qui n'est pas récupérée par le garbage collector. Ces services sont généralement fournis par la prise en charge des bibliothèques de classes ou importés directement depuis le système d'exploitation sous-jacent. Par exemple, la `Memory` classe ci-dessous montre comment les fonctions de tas d'un système d'exploitation sous-jacent sont accessibles à partir de C# :

C#

```
using System;
using System.Runtime.InteropServices;

public static unsafe class Memory
{
    // Handle for the process heap. This handle is used in all calls to the
    // HeapXXX APIs in the methods below.
    private static readonly IntPtr s_heap = GetProcessHeap();

    // Allocates a memory block of the given size. The allocated memory is
    // automatically initialized to zero.
    public static void* Alloc(int size)
    {
        void* result = HeapAlloc(s_heap, HEAP_ZERO_MEMORY, (UIntPtr)size);
        if (result == null) throw new OutOfMemoryException();
        return result;
    }
```

```
    // Copies count bytes from src to dst. The source and destination
    // blocks are permitted to overlap.
    public static void Copy(void* src, void* dst, int count)
    {
        byte* ps = (byte*)src;
        byte* pd = (byte*)dst;
        if (ps > pd)
        {
            for (; count != 0; count--) *pd++ = *ps++;
        }
        else if (ps < pd)
        {
            for (ps += count, pd += count; count != 0; count--) *--pd = *--
ps;
        }
    }

    // Frees a memory block.
    public static void Free(void* block)
    {
        if (!HeapFree(s_heap, 0, block)) throw new
InvalidOperationException();
    }

    // Re-allocates a memory block. If the reallocation request is for a
    // larger size, the additional region of memory is automatically
    // initialized to zero.
    public static void* ReAlloc(void* block, int size)
    {
        void* result = HeapReAlloc(s_heap, HEAP_ZERO_MEMORY, block,
(UIntPtr)size);
        if (result == null) throw new OutOfMemoryException();
        return result;
    }

    // Returns the size of a memory block.
    public static int SizeOf(void* block)
    {
        int result = (int)HeapSize(s_heap, 0, block);
        if (result == -1) throw new InvalidOperationException();
        return result;
    }

    // Heap API flags
    private const int HEAP_ZERO_MEMORY = 0x00000008;

    // Heap API functions
    [DllImport("kernel32")]
    private static extern IntPtr GetProcessHeap();

    [DllImport("kernel32")]
    private static extern void* HeapAlloc(IntPtr hHeap, int flags, UIntPtr
size);
```

```
    [DllImport("kernel32")]
    private static extern bool HeapFree(IntPtr hHeap, int flags, void*
block);

    [DllImport("kernel32")]
    private static extern void* HeapReAlloc(IntPtr hHeap, int flags, void*
block, UIntPtr size);

    [DllImport("kernel32")]
    private static extern UIntPtr HeapSize(IntPtr hHeap, int flags, void*
block);
}
```

Vous trouverez ci-dessous un exemple qui utilise la `Memory` classe :

C#

```
class Test
{
    static unsafe void Main()
    {
        byte* buffer = null;
        try
        {
            const int Size = 256;
            buffer = (byte*)Memory.Alloc(Size);
            for (int i = 0; i < Size; i++) buffer[i] = (byte)i;
            byte[] array = new byte[Size];
            fixed (byte* p = array) Memory.Copy(buffer, p, Size);
            for (int i = 0; i < Size; i++) Console.WriteLine(array[i]);
        }
        finally
        {
            if (buffer != null) Memory.Free(buffer);
        }
    }
}
```

L'exemple alloue 256 octets de mémoire par le biais de `Memory.Alloc` et initialise le bloc de mémoire avec des valeurs qui s'incrémentent de 0 à 255. Il alloue ensuite un tableau d'octets d'éléments 256 et utilise `Memory.Copy` pour copier le contenu du bloc de mémoire dans le tableau d'octets. Enfin, le bloc de mémoire est libéré à l'aide de `Memory.Free` et le contenu du tableau d'octets est généré sur la console.

Annex A Grammar

Article • 04/07/2023

This clause is informative.

A.1 General

This annex contains the grammar productions found in the specification, including the optional ones for unsafe code. Productions appear here in the same order in which they appear in the specification.

A.2 Lexical grammar

ANTLR

```
// Source: §6.3.1 General
DEFAULT  : 'default' ;
NULL     : 'null' ;
TRUE     : 'true' ;
FALSE    : 'false' ;
ASTERISK : '*' ;
SLASH    : '/' ;

// Source: §6.3.1 General
input
    : input_section?
    ;

input_section
    : input_section_part+
    ;

input_section_part
    : input_element* New_Line
    | PP_Directive
    ;

input_element
    : Whitespace
    | Comment
    | token
    ;

// Source: §6.3.2 Line terminators
New_Line
    : New_Line_Character
```

```
    | '\u000D\u000A'    // carriage return, line feed
    ;

// Source: §6.3.3 Comments
Comment
    : Single_Line_Comment
    | Delimited_Comment
    ;

fragment Single_Line_Comment
    : '//' Input_Character*
    ;

fragment Input_Character
    // anything but New_Line_Character
    : ~('\u000D' | '\u000A'   | '\u0085' | '\u2028' | '\u2029')
    ;

fragment New_Line_Character
    : '\u000D'  // carriage return
    | '\u000A'  // line feed
    | '\u0085'  // next line
    | '\u2028'  // line separator
    | '\u2029'  // paragraph separator
    ;

fragment Delimited_Comment
    : '/*' Delimited_Comment_Section* ASTERISK+ '/'
    ;

fragment Delimited_Comment_Section
    : SLASH
    | ASTERISK* Not_Slash_Or_Asterisk
    ;

fragment Not_Slash_Or_Asterisk
    : ~('/' | '*')    // Any except SLASH or ASTERISK
    ;

// Source: §6.3.4 White space
Whitespace
    : [\p{Zs}]  // any character with Unicode class Zs
    | '\u0009'  // horizontal tab
    | '\u000B'  // vertical tab
    | '\u000C'  // form feed
    ;

// Source: §6.4.1 General
token
    : identifier
    | keyword
    | Integer_Literal
    | Real_Literal
    | Character_Literal
    | String_Literal
```

```
    | operator_or_punctuator
    ;

// Source: §6.4.2 Unicode character escape sequences
fragment Unicode_Escape_Sequence
    : '\\u' Hex_Digit Hex_Digit Hex_Digit Hex_Digit
    | '\\U' Hex_Digit Hex_Digit Hex_Digit Hex_Digit
            Hex_Digit Hex_Digit Hex_Digit Hex_Digit
    ;

// Source: §6.4.3 Identifiers
identifier
    : Simple_Identifier
    | contextual_keyword
    ;

Simple_Identifier
    : Available_Identifier
    | Escaped_Identifier
    ;

fragment Available_Identifier
    // excluding keywords or contextual keywords, see note below
    : Basic_Identifier
    ;

fragment Escaped_Identifier
    // Includes keywords and contextual keywords prefixed by '@'.
    // See note below.
    : '@' Basic_Identifier
    ;

fragment Basic_Identifier
    : Identifier_Start_Character Identifier_Part_Character*
    ;

fragment Identifier_Start_Character
    : Letter_Character
    | Underscore_Character
    ;

fragment Underscore_Character
    : '_'           // underscore
    | '\\u005' [fF] // Unicode_Escape_Sequence for underscore
    ;

fragment Identifier_Part_Character
    : Letter_Character
    | Decimal_Digit_Character
    | Connecting_Character
    | Combining_Character
    | Formatting_Character
    ;

fragment Letter_Character
```

```
    // Category Letter, all subcategories; category Number, subcategory
letter.
    : [\p{L}\p{Nl}]
    // Only escapes for categories L & Nl allowed. See note below.
    | Unicode_Escape_Sequence
    ;

fragment Combining_Character
    // Category Mark, subcategories non-spacing and spacing combining.
    : [\p{Mn}\p{Mc}]
    // Only escapes for categories Mn & Mc allowed. See note below.
    | Unicode_Escape_Sequence
    ;

fragment Decimal_Digit_Character
    // Category Number, subcategory decimal digit.
    : [\p{Nd}]
    // Only escapes for category Nd allowed. See note below.
    | Unicode_Escape_Sequence
    ;

fragment Connecting_Character
    // Category Punctuation, subcategory connector.
    : [\p{Pc}]
    // Only escapes for category Pc allowed. See note below.
    | Unicode_Escape_Sequence
    ;

fragment Formatting_Character
    // Category Other, subcategory format.
    : [\p{Cf}]
    // Only escapes for category Cf allowed, see note below.
    | Unicode_Escape_Sequence
    ;

// Source: §6.4.4 Keywords
keyword
    : 'abstract' | 'as'       | 'base'       | 'bool'      | 'break'
    | 'byte'     | 'case'     | 'catch'      | 'char'      | 'checked'
    | 'class'    | 'const'    | 'continue'   | 'decimal'   | DEFAULT
    | 'delegate' | 'do'       | 'double'     | 'else'      | 'enum'
    | 'event'    | 'explicit' | 'extern'     | FALSE       | 'finally'
    | 'fixed'    | 'float'    | 'for'        | 'foreach'   | 'goto'
    | 'if'       | 'implicit' | 'in'         | 'int'       | 'interface'
    | 'internal' | 'is'       | 'lock'       | 'long'      | 'namespace'
    | 'new'      | NULL       | 'object'     | 'operator'  | 'out'
    | 'override' | 'params'   | 'private'    | 'protected' | 'public'
    | 'readonly' | 'ref'      | 'return'     | 'sbyte'     | 'sealed'
    | 'short'    | 'sizeof'   | 'stackalloc' | 'static'    | 'string'
    | 'struct'   | 'switch'   | 'this'       | 'throw'     | TRUE
    | 'try'      | 'typeof'   | 'uint'       | 'ulong'     | 'unchecked'
    | 'unsafe'   | 'ushort'   | 'using'      | 'virtual'   | 'void'
    | 'volatile' | 'while'
    ;
```

```
// Source: §6.4.4 Keywords
contextual_keyword
    : 'add'    | 'alias'      | 'ascending' | 'async'     | 'await'
    | 'by'     | 'descending' | 'dynamic'   | 'equals'    | 'from'
    | 'get'    | 'global'     | 'group'     | 'into'      | 'join'
    | 'let'    | 'nameof'     | 'on'        | 'orderby'   | 'partial'
    | 'remove' | 'select'     | 'set'       | 'unmanaged' | 'value'
    | 'var'    | 'when'       | 'where'     | 'yield'
    ;

// Source: §6.4.5.1 General
literal
    : boolean_literal
    | Integer_Literal
    | Real_Literal
    | Character_Literal
    | String_Literal
    | null_literal
    ;

// Source: §6.4.5.2 Boolean literals
boolean_literal
    : TRUE
    | FALSE
    ;

// Source: §6.4.5.3 Integer literals
Integer_Literal
    : Decimal_Integer_Literal
    | Hexadecimal_Integer_Literal
    | Binary_Integer_Literal
    ;

fragment Decimal_Integer_Literal
    : Decimal_Digit Decorated_Decimal_Digit* Integer_Type_Suffix?
    ;

fragment Decorated_Decimal_Digit
    : '_'* Decimal_Digit
    ;

fragment Decimal_Digit
    : '0'..'9'
    ;

fragment Integer_Type_Suffix
    : 'U' | 'u' | 'L' | 'l' |
      'UL' | 'Ul' | 'uL' | 'ul' | 'LU' | 'Lu' | 'lU' | 'lu'
    ;

fragment Hexadecimal_Integer_Literal
    : ('0x' | '0X') Decorated_Hex_Digit+ Integer_Type_Suffix?
    ;

fragment Decorated_Hex_Digit
```

```
    : '_'* Hex_Digit
    ;

fragment Hex_Digit
    : '0'..'9' | 'A'..'F' | 'a'..'f'
    ;

fragment Binary_Integer_Literal
    : ('0b' | '0B') Decorated_Binary_Digit+ Integer_Type_Suffix?
    ;

fragment Decorated_Binary_Digit
    : '_'* Binary_Digit
    ;

fragment Binary_Digit
    : '0' | '1'
    ;

// Source: §6.4.5.4 Real literals
Real_Literal
    : Decimal_Digit Decorated_Decimal_Digit* '.'
      Decimal_Digit Decorated_Decimal_Digit* Exponent_Part?
Real_Type_Suffix?
    | '.' Decimal_Digit Decorated_Decimal_Digit* Exponent_Part?
Real_Type_Suffix?
    | Decimal_Digit Decorated_Decimal_Digit* Exponent_Part Real_Type_Suffix?
    | Decimal_Digit Decorated_Decimal_Digit* Real_Type_Suffix
    ;

fragment Exponent_Part
    : ('e' | 'E') Sign? Decimal_Digit Decorated_Decimal_Digit*
    ;

fragment Sign
    : '+' | '-'
    ;

fragment Real_Type_Suffix
    : 'F' | 'f' | 'D' | 'd' | 'M' | 'm'
    ;

// Source: §6.4.5.5 Character literals
Character_Literal
    : '\'' Character '\''
    ;

fragment Character
    : Single_Character
    | Simple_Escape_Sequence
    | Hexadecimal_Escape_Sequence
    | Unicode_Escape_Sequence
    ;

fragment Single_Character
```

```
    // anything but ', \, and New_Line_Character
    : ~['\\\u000D\u000A\u0085\u2028\u2029]
    ;

fragment Simple_Escape_Sequence
    : '\\'' | '\\"' | '\\\\' | '\\0' | '\\a' | '\\b' |
      '\\f' | '\\n' | '\\r' | '\\t' | '\\v'
    ;

fragment Hexadecimal_Escape_Sequence
    : '\\x' Hex_Digit Hex_Digit? Hex_Digit? Hex_Digit?
    ;

// Source: §6.4.5.6 String literals
String_Literal
    : Regular_String_Literal
    | Verbatim_String_Literal
    ;

fragment Regular_String_Literal
    : '"' Regular_String_Literal_Character* '"'
    ;

fragment Regular_String_Literal_Character
    : Single_Regular_String_Literal_Character
    | Simple_Escape_Sequence
    | Hexadecimal_Escape_Sequence
    | Unicode_Escape_Sequence
    ;

fragment Single_Regular_String_Literal_Character
    // anything but ", \, and New_Line_Character
    : ~["\\\u000D\u000A\u0085\u2028\u2029]
    ;

fragment Verbatim_String_Literal
    : '@"' Verbatim_String_Literal_Character* '"'
    ;

fragment Verbatim_String_Literal_Character
    : Single_Verbatim_String_Literal_Character
    | Quote_Escape_Sequence
    ;

fragment Single_Verbatim_String_Literal_Character
    : ~["]     // anything but quotation mark (U+0022)
    ;

fragment Quote_Escape_Sequence
    : '""'
    ;

// Source: §6.4.5.7 The null literal
null_literal
    : NULL
```

```
    ;

// Source: §6.4.6 Operators and punctuators
operator_or_punctuator
    : '{'  | '}'  | '['  | ']'  | '('  | ')'  | '.'  | ','  | ':'  | ';'
    | '+'  | '-'  | ASTERISK   | SLASH | '%'  | '&'  | '|'  | '^'  | '!' |
'~'
    | '='  | '<'  | '>'  | '?'  | '??'  | '::' | '++' | '--' | '&&' | '||'
    | '->' | '==' | '!=' | '<=' | '>='  | '+=' | '-=' | '*=' | '/=' | '%='
    | '&=' | '|=' | '^=' | '<<' | '<<=' | '=>'
    ;

right_shift
    : '>'  '>'
    ;

right_shift_assignment
    : '>' '>='
    ;

// Source: §6.5.1 General
PP_Directive
    : PP_Start PP_Kind PP_New_Line
    ;

fragment PP_Kind
    : PP_Declaration
    | PP_Conditional
    | PP_Line
    | PP_Diagnostic
    | PP_Region
    | PP_Pragma
    ;

// Only recognised at the beginning of a line
fragment PP_Start
    // See note below.
    : { getCharPositionInLine() == 0 }? PP_Whitespace? '#' PP_Whitespace?
    ;

fragment PP_Whitespace
    : ( [\p{Zs}]  // any character with Unicode class Zs
      | '\u0009'  // horizontal tab
      | '\u000B'  // vertical tab
      | '\u000C'  // form feed
      )+
    ;

fragment PP_New_Line
    : PP_Whitespace? Single_Line_Comment? New_Line
    ;

// Source: §6.5.2 Conditional compilation symbols
fragment PP_Conditional_Symbol
    // Must not be equal to tokens TRUE or FALSE. See note below.
```

```
    : Basic_Identifier
    ;

// Source: §6.5.3 Pre-processing expressions
fragment PP_Expression
    : PP_Whitespace? PP_Or_Expression PP_Whitespace?
    ;

fragment PP_Or_Expression
    : PP_And_Expression (PP_Whitespace? '||' PP_Whitespace?
PP_And_Expression)*
    ;

fragment PP_And_Expression
    : PP_Equality_Expression (PP_Whitespace? '&&' PP_Whitespace?
      PP_Equality_Expression)*
    ;

fragment PP_Equality_Expression
    : PP_Unary_Expression (PP_Whitespace? ('==' | '!=') PP_Whitespace?
      PP_Unary_Expression)*
    ;

fragment PP_Unary_Expression
    : PP_Primary_Expression
    | '!' PP_Whitespace? PP_Unary_Expression
    ;

fragment PP_Primary_Expression
    : TRUE
    | FALSE
    | PP_Conditional_Symbol
    | '(' PP_Whitespace? PP_Expression PP_Whitespace? ')'
    ;

// Source: §6.5.4 Definition directives
fragment PP_Declaration
    : 'define' PP_Whitespace PP_Conditional_Symbol
    | 'undef' PP_Whitespace PP_Conditional_Symbol
    ;

// Source: §6.5.5 Conditional compilation directives
fragment PP_Conditional
    : PP_If_Section
    | PP_Elif_Section
    | PP_Else_Section
    | PP_Endif
    ;

fragment PP_If_Section
    : 'if' PP_Whitespace PP_Expression
    ;

fragment PP_Elif_Section
    : 'elif' PP_Whitespace PP_Expression
```

```
    ;

fragment PP_Else_Section
    : 'else'
    ;

fragment PP_Endif
    : 'endif'
    ;

// Source: §6.5.6 Diagnostic directives
fragment PP_Diagnostic
    : 'error' PP_Message?
    | 'warning' PP_Message?
    ;

fragment PP_Message
    : PP_Whitespace Input_Character*
    ;

// Source: §6.5.7 Region directives
fragment PP_Region
    : PP_Start_Region
    | PP_End_Region
    ;

fragment PP_Start_Region
    : 'region' PP_Message?
    ;

fragment PP_End_Region
    : 'endregion' PP_Message?
    ;

// Source: §6.5.8 Line directives
fragment PP_Line
    : 'line' PP_Whitespace PP_Line_Indicator
    ;

fragment PP_Line_Indicator
    : Decimal_Digit+ PP_Whitespace PP_Compilation_Unit_Name
    | Decimal_Digit+
    | DEFAULT
    | 'hidden'
    ;

fragment PP_Compilation_Unit_Name
    : '"' PP_Compilation_Unit_Name_Character+ '"'
    ;

fragment PP_Compilation_Unit_Name_Character
    // Any Input_Character except "
    : ~('\u000D' | '\u000A'   | '\u0085' | '\u2028' | '\u2029' | '#')
    ;
```

```
// Source: §6.5.9 Pragma directives
fragment PP_Pragma
    : 'pragma' PP_Pragma_Text?
    ;

fragment PP_Pragma_Text
    : PP_Whitespace Input_Character*
    ;
```

A.3 Syntactic grammar

ANTLR

```
// Source: §7.8.1 General
namespace_name
    : namespace_or_type_name
    ;

type_name
    : namespace_or_type_name
    ;

namespace_or_type_name
    : identifier type_argument_list?
    | namespace_or_type_name '.' identifier type_argument_list?
    | qualified_alias_member
    ;

// Source: §8.1 General
type
    : reference_type
    | value_type
    | type_parameter
    | pointer_type     // unsafe code support
    ;

// Source: §8.2.1 General
reference_type
    : class_type
    | interface_type
    | array_type
    | delegate_type
    | 'dynamic'
    ;

class_type
    : type_name
    | 'object'
    | 'string'
    ;
```

```
interface_type
    : type_name
    ;

array_type
    : non_array_type rank_specifier+
    ;

non_array_type
    : value_type
    | class_type
    | interface_type
    | delegate_type
    | 'dynamic'
    | type_parameter
    | pointer_type      // unsafe code support
    ;

rank_specifier
    : '[' ','* ']'
    ;

delegate_type
    : type_name
    ;

// Source: §8.3.1 General
value_type
    : non_nullable_value_type
    | nullable_value_type
    ;

non_nullable_value_type
    : struct_type
    | enum_type
    ;

struct_type
    : type_name
    | simple_type
    | tuple_type
    ;

simple_type
    : numeric_type
    | 'bool'
    ;

numeric_type
    : integral_type
    | floating_point_type
    | 'decimal'
    ;

integral_type
```

```
    : 'sbyte'
    | 'byte'
    | 'short'
    | 'ushort'
    | 'int'
    | 'uint'
    | 'long'
    | 'ulong'
    | 'char'
    ;

floating_point_type
    : 'float'
    | 'double'
    ;

tuple_type
    : '(' tuple_type_element (',' tuple_type_element)+ ')'
    ;

tuple_type_element
    : type identifier?
    ;

enum_type
    : type_name
    ;

nullable_value_type
    : non_nullable_value_type '?'
    ;

// Source: §8.4.2 Type arguments
type_argument_list
    : '<' type_arguments '>'
    ;

type_arguments
    : type_argument (',' type_argument)*
    ;

type_argument
    : type
    ;

// Source: §8.5 Type parameters
type_parameter
    : identifier
    ;

// Source: §8.8 Unmanaged types
unmanaged_type
    : value_type
    | pointer_type     // unsafe code support
    ;
```

```
// Source: §9.5 Variable references
variable_reference
    : expression
    ;

// Source: §11.2.1 General
pattern
    : declaration_pattern
    | constant_pattern
    | var_pattern
    ;

// Source: §11.2.2 Declaration pattern
declaration_pattern
    : type simple_designation
    ;
simple_designation
    : single_variable_designation
    ;
single_variable_designation
    : identifier
    ;

// Source: §11.2.3 Constant pattern
constant_pattern
    : constant_expression
    ;

// Source: §11.2.4 Var pattern
var_pattern
    : 'var' designation
    ;
designation
    : simple_designation
    ;

// Source: §12.6.2.1 General
argument_list
    : argument (',' argument)*
    ;

argument
    : argument_name? argument_value
    ;

argument_name
    : identifier ':'
    ;

argument_value
    : expression
    | 'in' variable_reference
    | 'ref' variable_reference
    | 'out' variable_reference
```

```
    ;

// Source: §12.8.1 General
primary_expression
    : primary_no_array_creation_expression
    | array_creation_expression
    ;

primary_no_array_creation_expression
    : literal
    | interpolated_string_expression
    | simple_name
    | parenthesized_expression
    | tuple_expression
    | member_access
    | null_conditional_member_access
    | invocation_expression
    | element_access
    | null_conditional_element_access
    | this_access
    | base_access
    | post_increment_expression
    | post_decrement_expression
    | object_creation_expression
    | delegate_creation_expression
    | anonymous_object_creation_expression
    | typeof_expression
    | sizeof_expression
    | checked_expression
    | unchecked_expression
    | default_value_expression
    | nameof_expression
    | anonymous_method_expression
    | pointer_member_access     // unsafe code support
    | pointer_element_access    // unsafe code support
    | stackalloc_expression
    ;

// Source: §12.8.3 Interpolated string expressions
interpolated_string_expression
    : interpolated_regular_string_expression
    | interpolated_verbatim_string_expression
    ;

// interpolated regular string expressions

interpolated_regular_string_expression
    : Interpolated_Regular_String_Start Interpolated_Regular_String_Mid?
      ('{' regular_interpolation '}' Interpolated_Regular_String_Mid?)*
      Interpolated_Regular_String_End
    ;

regular_interpolation
    : expression (',' interpolation_minimum_width)?
      Regular_Interpolation_Format?
```

```
    ;

interpolation_minimum_width
    : constant_expression
    ;

Interpolated_Regular_String_Start
    : '$"'
    ;

// the following three lexical rules are context sensitive, see details
below

Interpolated_Regular_String_Mid
    : Interpolated_Regular_String_Element+
    ;

Regular_Interpolation_Format
    : ':' Interpolated_Regular_String_Element+
    ;

Interpolated_Regular_String_End
    : '"'
    ;

fragment Interpolated_Regular_String_Element
    : Interpolated_Regular_String_Character
    | Simple_Escape_Sequence
    | Hexadecimal_Escape_Sequence
    | Unicode_Escape_Sequence
    | Open_Brace_Escape_Sequence
    | Close_Brace_Escape_Sequence
    ;

fragment Interpolated_Regular_String_Character
    // Any character except " (U+0022), \\ (U+005C),
    // { (U+007B), } (U+007D), and New_Line_Character.
    : ~["\\{}\u000D\u000A\u0085\u2028\u2029]
    ;

// interpolated verbatim string expressions

interpolated_verbatim_string_expression
    : Interpolated_Verbatim_String_Start Interpolated_Verbatim_String_Mid?
      ('{' verbatim_interpolation '}' Interpolated_Verbatim_String_Mid?)*
      Interpolated_Verbatim_String_End
    ;

verbatim_interpolation
    : expression (',' interpolation_minimum_width)?
      Verbatim_Interpolation_Format?
    ;

Interpolated_Verbatim_String_Start
    : '$@"'
```

```
    ;

// the following three lexical rules are context sensitive, see details
below

Interpolated_Verbatim_String_Mid
    : Interpolated_Verbatim_String_Element+
    ;

Verbatim_Interpolation_Format
    : ':' Interpolated_Verbatim_String_Element+
    ;

Interpolated_Verbatim_String_End
    : '"'
    ;

fragment Interpolated_Verbatim_String_Element
    : Interpolated_Verbatim_String_Character
    | Quote_Escape_Sequence
    | Open_Brace_Escape_Sequence
    | Close_Brace_Escape_Sequence
    ;

fragment Interpolated_Verbatim_String_Character
    : ~["{}]    // Any character except " (U+0022), { (U+007B) and }
(U+007D)
    ;

// lexical fragments used by both regular and verbatim interpolated strings

fragment Open_Brace_Escape_Sequence
    : '{{'
    ;

fragment Close_Brace_Escape_Sequence
    : '}}'
    ;

// Source: §12.8.4 Simple names
simple_name
    : identifier type_argument_list?
    ;

// Source: §12.8.5 Parenthesized expressions
parenthesized_expression
    : '(' expression ')'
    ;

// Source: §12.8.6 Tuple expressions
tuple_expression
    : '(' tuple_element (',' tuple_element)+ ')'
    | deconstruction_expression
    ;
```

```
tuple_element
    : (identifier ':')? expression
    ;

deconstruction_expression
    : 'var' deconstruction_tuple
    ;

deconstruction_tuple
    : '(' deconstruction_element (',' deconstruction_element)+ ')'
    ;

deconstruction_element
    : deconstruction_tuple
    | identifier
    ;

// Source: §12.8.7.1 General
member_access
    : primary_expression '.' identifier type_argument_list?
    | predefined_type '.' identifier type_argument_list?
    | qualified_alias_member '.' identifier type_argument_list?
    ;

predefined_type
    : 'bool' | 'byte' | 'char' | 'decimal' | 'double' | 'float' | 'int'
    | 'long' | 'object' | 'sbyte' | 'short' | 'string' | 'uint' | 'ulong'
    | 'ushort'
    ;

// Source: §12.8.8 Null Conditional Member Access
null_conditional_member_access
    : primary_expression '?' '.' identifier type_argument_list?
      dependent_access*
    ;

dependent_access
    : '.' identifier type_argument_list?    // member access
    | '[' argument_list ']'                 // element access
    | '(' argument_list? ')'                // invocation
    ;

null_conditional_projection_initializer
    : primary_expression '?' '.' identifier type_argument_list?
    ;

// Source: §12.8.9.1 General
invocation_expression
    : primary_expression '(' argument_list? ')'
    ;

// Source: §12.8.10 Null Conditional Invocation Expression
null_conditional_invocation_expression
    : null_conditional_member_access '(' argument_list? ')'
    | null_conditional_element_access '(' argument_list? ')'
```

```
    ;

// Source: §12.8.11.1 General
element_access
    : primary_no_array_creation_expression '[' argument_list ']'
    ;

// Source: §12.8.12 Null Conditional Element Access
null_conditional_element_access
    : primary_no_array_creation_expression '?' '[' argument_list ']'
      dependent_access*
    ;

// Source: §12.8.13 This access
this_access
    : 'this'
    ;

// Source: §12.8.14 Base access
base_access
    : 'base' '.' identifier type_argument_list?
    | 'base' '[' argument_list ']'
    ;

// Source: §12.8.15 Postfix increment and decrement operators
post_increment_expression
    : primary_expression '++'
    ;

post_decrement_expression
    : primary_expression '--'
    ;

// Source: §12.8.16.2 Object creation expressions
object_creation_expression
    : 'new' type '(' argument_list? ')' object_or_collection_initializer?
    | 'new' type object_or_collection_initializer
    ;

object_or_collection_initializer
    : object_initializer
    | collection_initializer
    ;

// Source: §12.8.16.3 Object initializers
object_initializer
    : '{' member_initializer_list? '}'
    | '{' member_initializer_list ',' '}'
    ;

member_initializer_list
    : member_initializer (',' member_initializer)*
    ;

member_initializer
```

```
    : initializer_target '=' initializer_value
    ;

initializer_target
    : identifier
    | '[' argument_list ']'
    ;

initializer_value
    : expression
    | object_or_collection_initializer
    ;

// Source: §12.8.16.4 Collection initializers
collection_initializer
    : '{' element_initializer_list '}'
    | '{' element_initializer_list ',' '}'
    ;

element_initializer_list
    : element_initializer (',' element_initializer)*
    ;

element_initializer
    : non_assignment_expression
    | '{' expression_list '}'
    ;

expression_list
    : expression
    | expression_list ',' expression
    ;

// Source: §12.8.16.5 Array creation expressions
array_creation_expression
    : 'new' non_array_type '[' expression_list ']' rank_specifier*
      array_initializer?
    | 'new' array_type array_initializer
    | 'new' rank_specifier array_initializer
    ;

// Source: §12.8.16.6 Delegate creation expressions
delegate_creation_expression
    : 'new' delegate_type '(' expression ')'
    ;

// Source: §12.8.16.7 Anonymous object creation expressions
anonymous_object_creation_expression
    : 'new' anonymous_object_initializer
    ;

anonymous_object_initializer
    : '{' member_declarator_list? '}'
    | '{' member_declarator_list ',' '}'
    ;
```

```
member_declarator_list
    : member_declarator (',' member_declarator)*
    ;

member_declarator
    : simple_name
    | member_access
    | null_conditional_projection_initializer
    | base_access
    | identifier '=' expression
    ;

// Source: §12.8.17 The typeof operator
typeof_expression
    : 'typeof' '(' type ')'
    | 'typeof' '(' unbound_type_name ')'
    | 'typeof' '(' 'void' ')'
    ;

unbound_type_name
    : identifier generic_dimension_specifier?
    | identifier '::' identifier generic_dimension_specifier?
    | unbound_type_name '.' identifier generic_dimension_specifier?
    ;

generic_dimension_specifier
    : '<' comma* '>'
    ;

comma
    : ','
    ;


// Source: §12.8.18 The sizeof operator
sizeof_expression
    : 'sizeof' '(' unmanaged_type ')'
    ;

// Source: §12.8.19 The checked and unchecked operators
checked_expression
    : 'checked' '(' expression ')'
    ;

unchecked_expression
    : 'unchecked' '(' expression ')'
    ;

// Source: §12.8.20 Default value expressions
default_value_expression
    : explictly_typed_default
    | default_literal
    ;
```

```
explictly_typed_default
    : 'default' '(' type ')'
    ;

default_literal
    : 'default'
    ;

// Source: §12.8.21 Stack allocation
stackalloc_expression
    : 'stackalloc' unmanaged_type '[' expression ']'
    | 'stackalloc' unmanaged_type? '[' constant_expression? ']'
stackalloc_initializer
    ;

stackalloc_initializer
     : '{' stackalloc_initializer_element_list '}'
     ;

stackalloc_initializer_element_list
     : stackalloc_element_initializer (',' stackalloc_element_initializer)*
','?
     ;

stackalloc_element_initializer
    : expression
    ;

// Source: §12.8.22 Nameof expressions
nameof_expression
    : 'nameof' '(' named_entity ')'
    ;

named_entity
    : named_entity_target ('.' identifier type_argument_list?)*
    ;

named_entity_target
    : simple_name
    | 'this'
    | 'base'
    | predefined_type
    | qualified_alias_member
    ;

// Source: §12.9.1 General
unary_expression
    : primary_expression
    | '+' unary_expression
    | '-' unary_expression
    | '!' unary_expression
    | '~' unary_expression
    | pre_increment_expression
    | pre_decrement_expression
    | cast_expression
```

```
    | await_expression
    | pointer_indirection_expression    // unsafe code support
    | addressof_expression              // unsafe code support
    ;

// Source: §12.9.6 Prefix increment and decrement operators
pre_increment_expression
    : '++' unary_expression
    ;

pre_decrement_expression
    : '--' unary_expression
    ;

// Source: §12.9.7 Cast expressions
cast_expression
    : '(' type ')' unary_expression
    ;

// Source: §12.9.8.1 General
await_expression
    : 'await' unary_expression
    ;

// Source: §12.10.1 General
multiplicative_expression
    : unary_expression
    | multiplicative_expression '*' unary_expression
    | multiplicative_expression '/' unary_expression
    | multiplicative_expression '%' unary_expression
    ;

additive_expression
    : multiplicative_expression
    | additive_expression '+' multiplicative_expression
    | additive_expression '-' multiplicative_expression
    ;

// Source: §12.11 Shift operators
shift_expression
    : additive_expression
    | shift_expression '<<' additive_expression
    | shift_expression right_shift additive_expression
    ;

// Source: §12.12.1 General
relational_expression
    : shift_expression
    | relational_expression '<' shift_expression
    | relational_expression '>' shift_expression
    | relational_expression '<=' shift_expression
    | relational_expression '>=' shift_expression
    | relational_expression 'is' type
    | relational_expression 'is' pattern
    | relational_expression 'as' type
```

```
    ;

equality_expression
    : relational_expression
    | equality_expression '==' relational_expression
    | equality_expression '!=' relational_expression
    ;

// Source: §12.13.1 General
and_expression
    : equality_expression
    | and_expression '&' equality_expression
    ;

exclusive_or_expression
    : and_expression
    | exclusive_or_expression '^' and_expression
    ;

inclusive_or_expression
    : exclusive_or_expression
    | inclusive_or_expression '|' exclusive_or_expression
    ;

// Source: §12.14.1 General
conditional_and_expression
    : inclusive_or_expression
    | conditional_and_expression '&&' inclusive_or_expression
    ;

conditional_or_expression
    : conditional_and_expression
    | conditional_or_expression '||' conditional_and_expression
    ;

// Source: §12.15 The null coalescing operator
null_coalescing_expression
    : conditional_or_expression
    | conditional_or_expression '??' null_coalescing_expression
    | throw_expression
    ;

// Source: §12.16 The throw expression operator
throw_expression
    : 'throw' null_coalescing_expression
    ;

// Source: §12.17 Declaration expressions
declaration_expression
    : local_variable_type identifier
    ;

// Source: §12.18 Conditional operator
conditional_expression
    : null_coalescing_expression
```

```
    | null_coalescing_expression '?' expression ':' expression
    | null_coalescing_expression '?' 'ref' variable_reference ':' 'ref'
variable_reference
    ;

// Source: §12.19.1 General
lambda_expression
    : 'async'? anonymous_function_signature '=>' anonymous_function_body
    ;

anonymous_method_expression
    : 'async'? 'delegate' explicit_anonymous_function_signature? block
    ;

anonymous_function_signature
    : explicit_anonymous_function_signature
    | implicit_anonymous_function_signature
    ;

explicit_anonymous_function_signature
    : '(' explicit_anonymous_function_parameter_list? ')'
    ;

explicit_anonymous_function_parameter_list
    : explicit_anonymous_function_parameter
      (',' explicit_anonymous_function_parameter)*
    ;

explicit_anonymous_function_parameter
    : anonymous_function_parameter_modifier? type identifier
    ;

anonymous_function_parameter_modifier
    : 'ref'
    | 'out'
    | 'in'
    ;

implicit_anonymous_function_signature
    : '(' implicit_anonymous_function_parameter_list? ')'
    | implicit_anonymous_function_parameter
    ;

implicit_anonymous_function_parameter_list
    : implicit_anonymous_function_parameter
      (',' implicit_anonymous_function_parameter)*
    ;

implicit_anonymous_function_parameter
    : identifier
    ;

anonymous_function_body
    : null_conditional_invocation_expression
    | expression
```

```
    | 'ref' variable_reference
    | block
    ;

// Source: §12.20.1 General
query_expression
    : from_clause query_body
    ;

from_clause
    : 'from' type? identifier 'in' expression
    ;

query_body
    : query_body_clauses? select_or_group_clause query_continuation?
    ;

query_body_clauses
    : query_body_clause
    | query_body_clauses query_body_clause
    ;

query_body_clause
    : from_clause
    | let_clause
    | where_clause
    | join_clause
    | join_into_clause
    | orderby_clause
    ;

let_clause
    : 'let' identifier '=' expression
    ;

where_clause
    : 'where' boolean_expression
    ;

join_clause
    : 'join' type? identifier 'in' expression 'on' expression
      'equals' expression
    ;

join_into_clause
    : 'join' type? identifier 'in' expression 'on' expression
      'equals' expression 'into' identifier
    ;

orderby_clause
    : 'orderby' orderings
    ;

orderings
    : ordering (',' ordering)*
```

```
    ;

ordering
    : expression ordering_direction?
    ;

ordering_direction
    : 'ascending'
    | 'descending'
    ;

select_or_group_clause
    : select_clause
    | group_clause
    ;

select_clause
    : 'select' expression
    ;

group_clause
    : 'group' expression 'by' expression
    ;

query_continuation
    : 'into' identifier query_body
    ;

// Source: §12.21.1 General
assignment
    : unary_expression assignment_operator expression
    ;

assignment_operator
    : '=' 'ref'? | '+=' | '-=' | '*=' | '/=' | '%=' | '&=' | '|=' | '^=' |
'<<='
    | right_shift_assignment
    ;

// Source: §12.22 Expression
expression
    : non_assignment_expression
    | assignment
    ;

non_assignment_expression
    : declaration_expression
    | conditional_expression
    | lambda_expression
    | query_expression
    ;

// Source: §12.23 Constant expressions
constant_expression
    : expression
```

```
    ;

// Source: §12.24 Boolean expressions
boolean_expression
    : expression
    ;

// Source: §13.1 General
statement
    : labeled_statement
    | declaration_statement
    | embedded_statement
    ;

embedded_statement
    : block
    | empty_statement
    | expression_statement
    | selection_statement
    | iteration_statement
    | jump_statement
    | try_statement
    | checked_statement
    | unchecked_statement
    | lock_statement
    | using_statement
    | yield_statement
    | unsafe_statement   // unsafe code support
    | fixed_statement    // unsafe code support
    ;

// Source: §13.3.1 General
block
    : '{' statement_list? '}'
    ;

// Source: §13.3.2 Statement lists
statement_list
    : statement+
    ;

// Source: §13.4 The empty statement
empty_statement
    : ';'
    ;

// Source: §13.5 Labeled statements
labeled_statement
    : identifier ':' statement
    ;

// Source: §13.6.1 General
declaration_statement
    : local_variable_declaration ';'
    | local_constant_declaration ';'
```

```
    | local_function_declaration
    ;

// Source: §13.6.2 Local variable declarations
local_variable_declaration
    : ref_kind? local_variable_type local_variable_declarators
    ;

local_variable_type
    : type
    | 'var'
    ;

local_variable_declarators
    : local_variable_declarator
    | local_variable_declarators ',' local_variable_declarator
    ;

local_variable_declarator
    : identifier
    | identifier '=' local_variable_initializer
    ;

local_variable_initializer
    : expression
    | 'ref' variable_reference
    | array_initializer
    ;

// Source: §13.6.3 Local constant declarations
local_constant_declaration
    : 'const' type constant_declarators
    ;

constant_declarators
    : constant_declarator (',' constant_declarator)*
    ;

constant_declarator
    : identifier '=' constant_expression
    ;

// Source: §13.6.4 Local function declarations
local_function_declaration
    : local_function_modifier* return_type local_function_header
local_function_body
    | ref_kind ref_return_type local_function_header ref_local_function_body
    ;

local_function_header
    : identifier '(' formal_parameter_list? ')'
    | identifier type_parameter_list '(' formal_parameter_list? ')'
type_parameter_constraints_clause*
    ;
```

```
local_function_modifier
    : 'async'
    | unsafe_modifier   // unsafe code support
    ;

local_function_body
    : block
    | '=>' null_conditional_invocation_expression ';'
    | '=>' expression ';'
    ;

ref_local_function_body
    : block
    | '=>' 'ref' variable_reference ';'
    ;

// Source: §13.7 Expression statements
expression_statement
    : statement_expression ';'
    ;

statement_expression
    : null_conditional_invocation_expression
    | invocation_expression
    | object_creation_expression
    | assignment
    | post_increment_expression
    | post_decrement_expression
    | pre_increment_expression
    | pre_decrement_expression
    | await_expression
    ;

// Source: §13.8.1 General
selection_statement
    : if_statement
    | switch_statement
    ;

// Source: §13.8.2 The if statement
if_statement
    : 'if' '(' boolean_expression ')' embedded_statement
    | 'if' '(' boolean_expression ')' embedded_statement
      'else' embedded_statement
    ;

// Source: §13.8.3 The switch statement
switch_statement
    : 'switch' '(' expression ')' switch_block
    ;

switch_block
    : '{' switch_section* '}'
    ;
```

```
switch_section
    : switch_label+ statement_list
    ;

switch_label
    : 'case' pattern case_guard?  ':'
    | 'default' ':'
    ;

case_guard
    : 'when' expression
    ;

// Source: §13.9.1 General
iteration_statement
    : while_statement
    | do_statement
    | for_statement
    | foreach_statement
    ;

// Source: §13.9.2 The while statement
while_statement
    : 'while' '(' boolean_expression ')' embedded_statement
    ;

// Source: §13.9.3 The do statement
do_statement
    : 'do' embedded_statement 'while' '(' boolean_expression ')' ';'
    ;

// Source: §13.9.4 The for statement
for_statement
    : 'for' '(' for_initializer? ';' for_condition? ';' for_iterator? ')'
      embedded_statement
    ;

for_initializer
    : local_variable_declaration
    | statement_expression_list
    ;

for_condition
    : boolean_expression
    ;

for_iterator
    : statement_expression_list
    ;

statement_expression_list
    : statement_expression (',' statement_expression)*
    ;

// Source: §13.9.5 The foreach statement
```

```
foreach_statement
    : 'foreach' '(' ref_kind? local_variable_type identifier 'in'
      expression ')' embedded_statement
    ;

// Source: §13.10.1 General
jump_statement
    : break_statement
    | continue_statement
    | goto_statement
    | return_statement
    | throw_statement
    ;

// Source: §13.10.2 The break statement
break_statement
    : 'break' ';'
    ;

// Source: §13.10.3 The continue statement
continue_statement
    : 'continue' ';'
    ;

// Source: §13.10.4 The goto statement
goto_statement
    : 'goto' identifier ';'
    | 'goto' 'case' constant_expression ';'
    | 'goto' 'default' ';'
    ;

// Source: §13.10.5 The return statement
return_statement
    : 'return' ';'
    | 'return' expression ';'
    | 'return' 'ref' variable_reference ';'
    ;

// Source: §13.10.6 The throw statement
throw_statement
    : 'throw' expression? ';'
    ;

// Source: §13.11 The try statement
try_statement
    : 'try' block catch_clauses
    | 'try' block catch_clauses? finally_clause
    ;

catch_clauses
    : specific_catch_clause+
    | specific_catch_clause* general_catch_clause
    ;

specific_catch_clause
```

```
    : 'catch' exception_specifier exception_filter? block
    | 'catch' exception_filter block
    ;

exception_specifier
    : '(' type identifier? ')'
    ;

exception_filter
    : 'when' '(' boolean_expression ')'
    ;

general_catch_clause
    : 'catch' block
    ;

finally_clause
    : 'finally' block
    ;

// Source: §13.12 The checked and unchecked statements
checked_statement
    : 'checked' block
    ;

unchecked_statement
    : 'unchecked' block
    ;

// Source: §13.13 The lock statement
lock_statement
    : 'lock' '(' expression ')' embedded_statement
    ;

// Source: §13.14 The using statement
using_statement
    : 'using' '(' resource_acquisition ')' embedded_statement
    ;

resource_acquisition
    : local_variable_declaration
    | expression
    ;

// Source: §13.15 The yield statement
yield_statement
    : 'yield' 'return' expression ';'
    | 'yield' 'break' ';'
    ;

// Source: §14.2 Compilation units
compilation_unit
    : extern_alias_directive* using_directive* global_attributes?
      namespace_member_declaration*
    ;
```

```
// Source: §14.3 Namespace declarations
namespace_declaration
    : 'namespace' qualified_identifier namespace_body ';'?
    ;

qualified_identifier
    : identifier ('.' identifier)*
    ;

namespace_body
    : '{' extern_alias_directive* using_directive*
      namespace_member_declaration* '}'
    ;

// Source: §14.4 Extern alias directives
extern_alias_directive
    : 'extern' 'alias' identifier ';'
    ;

// Source: §14.5.1 General
using_directive
    : using_alias_directive
    | using_namespace_directive
    | using_static_directive
    ;

// Source: §14.5.2 Using alias directives
using_alias_directive
    : 'using' identifier '=' namespace_or_type_name ';'
    ;

// Source: §14.5.3 Using namespace directives
using_namespace_directive
    : 'using' namespace_name ';'
    ;

// Source: §14.5.4 Using static directives
using_static_directive
    : 'using' 'static' type_name ';'
    ;

// Source: §14.6 Namespace member declarations
namespace_member_declaration
    : namespace_declaration
    | type_declaration
    ;

// Source: §14.7 Type declarations
type_declaration
    : class_declaration
    | struct_declaration
    | interface_declaration
    | enum_declaration
    | delegate_declaration
```

```
    ;

// Source: §14.8.1 General
qualified_alias_member
    : identifier '::' identifier type_argument_list?
    ;

// Source: §15.2.1 General
class_declaration
    : attributes? class_modifier* 'partial'? 'class' identifier
        type_parameter_list? class_base? type_parameter_constraints_clause*
        class_body ';'?
    ;

// Source: §15.2.2.1 General
class_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'abstract'
    | 'sealed'
    | 'static'
    | unsafe_modifier   // unsafe code support
    ;

// Source: §15.2.3 Type parameters
type_parameter_list
    : '<' type_parameters '>'
  ;

type_parameters
    : attributes? type_parameter
    | type_parameters ',' attributes? type_parameter
    ;

// Source: §15.2.4.1 General
class_base
    : ':' class_type
    | ':' interface_type_list
    | ':' class_type ',' interface_type_list
    ;

interface_type_list
    : interface_type (',' interface_type)*
    ;

// Source: §15.2.5 Type parameter constraints
type_parameter_constraints_clauses
    : type_parameter_constraints_clause
    | type_parameter_constraints_clauses type_parameter_constraints_clause
    ;

type_parameter_constraints_clause
```

```
    : 'where' type_parameter ':' type_parameter_constraints
    ;

type_parameter_constraints
    : primary_constraint
    | secondary_constraints
    | constructor_constraint
    | primary_constraint ',' secondary_constraints
    | primary_constraint ',' constructor_constraint
    | secondary_constraints ',' constructor_constraint
    | primary_constraint ',' secondary_constraints ','
constructor_constraint
    ;

primary_constraint
    : class_type
    | 'class'
    | 'struct'
    | 'unmanaged'
    ;

secondary_constraints
    : interface_type
    | type_parameter
    | secondary_constraints ',' interface_type
    | secondary_constraints ',' type_parameter
    ;

constructor_constraint
    : 'new' '(' ')'
    ;

// Source: §15.2.6 Class body
class_body
    : '{' class_member_declaration* '}'
    ;

// Source: §15.3.1 General
class_member_declaration
    : constant_declaration
    | field_declaration
    | method_declaration
    | property_declaration
    | event_declaration
    | indexer_declaration
    | operator_declaration
    | constructor_declaration
    | finalizer_declaration
    | static_constructor_declaration
    | type_declaration
    ;

// Source: §15.4 Constants
constant_declaration
    : attributes? constant_modifier* 'const' type constant_declarators ';'
```

```
    ;

constant_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    ;

// Source: §15.5.1 General
field_declaration
    : attributes? field_modifier* type variable_declarators ';'
    ;

field_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'static'
    | 'readonly'
    | 'volatile'
    | unsafe_modifier   // unsafe code support
    ;

variable_declarators
    : variable_declarator (',' variable_declarator)*
    ;

variable_declarator
    : identifier ('=' variable_initializer)?
    ;

// Source: §15.6.1 General
method_declaration
    : attributes? method_modifiers return_type method_header method_body
    | attributes? ref_method_modifiers ref_kind ref_return_type
method_header ref_method_body
    ;

method_modifiers
    : method_modifier* 'partial'?
    ;

ref_kind
    : 'ref'
    | 'ref' 'readonly'
    ;

ref_method_modifiers
    : ref_method_modifier*
    ;
```

```
method_header
    : member_name '(' formal_parameter_list? ')'
    | member_name type_parameter_list '(' formal_parameter_list? ')'
type_parameter_constraints_clause*
    ;

method_modifier
    : ref_method_modifier
    | 'async'
    | unsafe_modifier   // unsafe code support
    ;

ref_method_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'static'
    | 'virtual'
    | 'sealed'
    | 'override'
    | 'abstract'
    | 'extern'
    ;

return_type
    : ref_return_type
    | 'void'
    ;

ref_return_type
    : type
    ;

member_name
    : identifier
    | interface_type '.' identifier
    ;

method_body
    : block
    | '=>' null_conditional_invocation_expression ';'
    | '=>' expression ';'
    | ';'
    ;

ref_method_body
    : block
    | '=>' 'ref' variable_reference ';'
    | ';'
    ;

// Source: §15.6.2.1 General
formal_parameter_list
```

```
    : fixed_parameters
    | fixed_parameters ',' parameter_array
    | parameter_array
    ;

fixed_parameters
    : fixed_parameter (',' fixed_parameter)*
    ;

fixed_parameter
    : attributes? parameter_modifier? type identifier default_argument?
    ;

default_argument
    : '=' expression
    ;

parameter_modifier
    : parameter_mode_modifier
    | 'this'
    ;

parameter_mode_modifier
    : 'ref'
    | 'out'
    | 'in'
    ;

parameter_array
    : attributes? 'params' array_type identifier
    ;

// Source: §15.7.1 General
property_declaration
    : attributes? property_modifier* type member_name property_body
    | attributes? property_modifier* ref_kind type member_name
ref_property_body
    ;

property_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'static'
    | 'virtual'
    | 'sealed'
    | 'override'
    | 'abstract'
    | 'extern'
    | unsafe_modifier   // unsafe code support
    ;

property_body
```

```
    : '{' accessor_declarations '}' property_initializer?
    | '=>' expression ';'
    ;

property_initializer
    : '=' variable_initializer ';'
    ;

ref_property_body
    : '{' ref_get_accessor_declaration '}'
    | '=>' 'ref' variable_reference ';'
    ;

// Source: §15.7.3 Accessors
accessor_declarations
    : get_accessor_declaration set_accessor_declaration?
    | set_accessor_declaration get_accessor_declaration?
    ;

get_accessor_declaration
    : attributes? accessor_modifier? 'get' accessor_body
    ;

set_accessor_declaration
    : attributes? accessor_modifier? 'set' accessor_body
    ;

accessor_modifier
    : 'protected'
    | 'internal'
    | 'private'
    | 'protected' 'internal'
    | 'internal' 'protected'
    | 'protected' 'private'
    | 'private' 'protected'
    ;

accessor_body
    : block
    | '=>' expression ';'
    | ';'
    ;

ref_get_accessor_declaration
    : attributes? accessor_modifier? 'get' ref_accessor_body
    ;

ref_accessor_body
    : block
    | '=>' 'ref' variable_reference ';'
    | ';'
    ;

// Source: §15.8.1 General
event_declaration
```

```
    : attributes? event_modifier* 'event' type variable_declarators ';'
    | attributes? event_modifier* 'event' type member_name
        '{' event_accessor_declarations '}'
    ;

event_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'static'
    | 'virtual'
    | 'sealed'
    | 'override'
    | 'abstract'
    | 'extern'
    | unsafe_modifier   // unsafe code support
    ;

event_accessor_declarations
    : add_accessor_declaration remove_accessor_declaration
    | remove_accessor_declaration add_accessor_declaration
    ;

add_accessor_declaration
    : attributes? 'add' block
    ;

remove_accessor_declaration
    : attributes? 'remove' block
    ;

// Source: §15.9.1 General
indexer_declaration
    : attributes? indexer_modifier* indexer_declarator indexer_body
    | attributes? indexer_modifier* ref_kind indexer_declarator
ref_indexer_body
    ;

indexer_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'virtual'
    | 'sealed'
    | 'override'
    | 'abstract'
    | 'extern'
    | unsafe_modifier   // unsafe code support
    ;

indexer_declarator
```

```
    : type 'this' '[' formal_parameter_list ']'
    | type interface_type '.' 'this' '[' formal_parameter_list ']'
    ;

indexer_body
    : '{' accessor_declarations '}'
    | '=>' expression ';'
    ;

ref_indexer_body
    : '{' ref_get_accessor_declaration '}'
    | '=>' 'ref' variable_reference ';'
    ;

// Source: §15.10.1 General
operator_declaration
    : attributes? operator_modifier+ operator_declarator operator_body
    ;

operator_modifier
    : 'public'
    | 'static'
    | 'extern'
    | unsafe_modifier   // unsafe code support
    ;

operator_declarator
    : unary_operator_declarator
    | binary_operator_declarator
    | conversion_operator_declarator
    ;

unary_operator_declarator
    : type 'operator' overloadable_unary_operator '(' fixed_parameter ')'
    ;

overloadable_unary_operator
    : '+' | '-' | '!' | '~' | '++' | '--' | 'true' | 'false'
    ;

binary_operator_declarator
    : type 'operator' overloadable_binary_operator
        '(' fixed_parameter ',' fixed_parameter ')'
    ;

overloadable_binary_operator
    : '+'  | '-'  | '*'  | '/'  | '%'  | '&' | '|' | '^'  | '<<'
    | right_shift | '==' | '!=' | '>' | '<' | '>=' | '<='
    ;

conversion_operator_declarator
    : 'implicit' 'operator' type '(' fixed_parameter ')'
    | 'explicit' 'operator' type '(' fixed_parameter ')'
    ;
```

```
operator_body
    : block
    | '=>' expression ';'
    | ';'
    ;

// Source: §15.11.1 General
constructor_declaration
    : attributes? constructor_modifier* constructor_declarator
constructor_body
    ;

constructor_modifier
    : 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'extern'
    | unsafe_modifier   // unsafe code support
    ;

constructor_declarator
    : identifier '(' formal_parameter_list? ')' constructor_initializer?
    ;

constructor_initializer
    : ':' 'base' '(' argument_list? ')'
    | ':' 'this' '(' argument_list? ')'
    ;

constructor_body
    : block
    | '=>' expression ';'
    | ';'
    ;

// Source: §15.12 Static constructors
static_constructor_declaration
    : attributes? static_constructor_modifiers identifier '(' ')'
        static_constructor_body
    ;

static_constructor_modifiers
    : 'static'
    | 'static' 'extern' unsafe_modifier?
    | 'static' unsafe_modifier 'extern'?
    | 'extern' 'static' unsafe_modifier?
    | 'extern' unsafe_modifier 'static'
    | unsafe_modifier 'static' 'extern'?
    | unsafe_modifier 'extern' 'static'
    ;

static_constructor_body
    : block
    | '=>' expression ';'
```

```
    | ';'
    ;

// Source: §15.13 Finalizers
finalizer_declaration
    : attributes? '~' identifier '(' ')' finalizer_body
    | attributes? 'extern' unsafe_modifier? '~' identifier '(' ')'
      finalizer_body
    | attributes? unsafe_modifier 'extern'? '~' identifier '(' ')'
      finalizer_body
    ;

finalizer_body
    : block
    | '=>' expression ';'
    | ';'
    ;

// Source: §16.2.1 General
struct_declaration
    : attributes? struct_modifier* 'ref'? 'partial'? 'struct'
      identifier type_parameter_list? struct_interfaces?
      type_parameter_constraints_clause* struct_body ';'?
    ;

// Source: §16.2.2 Struct modifiers
struct_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | 'readonly'
    | unsafe_modifier   // unsafe code support
    ;

// Source: §16.2.5 Struct interfaces
struct_interfaces
    : ':' interface_type_list
    ;

// Source: §16.2.6 Struct body
struct_body
    : '{' struct_member_declaration* '}'
    ;

// Source: §16.3 Struct members
struct_member_declaration
    : constant_declaration
    | field_declaration
    | method_declaration
    | property_declaration
    | event_declaration
    | indexer_declaration
    | operator_declaration
```

```
    | constructor_declaration
    | static_constructor_declaration
    | type_declaration
    | fixed_size_buffer_declaration   // unsafe code support
    ;

// Source: §17.7 Array initializers
array_initializer
    : '{' variable_initializer_list? '}'
    | '{' variable_initializer_list ',' '}'
    ;

variable_initializer_list
    : variable_initializer (',' variable_initializer)*
    ;

variable_initializer
    : expression
    | array_initializer
    ;

// Source: §18.2.1 General
interface_declaration
    : attributes? interface_modifier* 'partial'? 'interface'
      identifier variant_type_parameter_list? interface_base?
      type_parameter_constraints_clause* interface_body ';'?
    ;

// Source: §18.2.2 Interface modifiers
interface_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    | unsafe_modifier   // unsafe code support
    ;

// Source: §18.2.3.1 General
variant_type_parameter_list
    : '<' variant_type_parameters '>'
    ;

// Source: §18.2.3.1 General
variant_type_parameters
    : attributes? variance_annotation? type_parameter
    | variant_type_parameters ',' attributes? variance_annotation?
      type_parameter
    ;

// Source: §18.2.3.1 General
variance_annotation
    : 'in'
    | 'out'
    ;
```

```
// Source: §18.2.4 Base interfaces
interface_base
    : ':' interface_type_list
    ;

// Source: §18.3 Interface body
interface_body
    : '{' interface_member_declaration* '}'
    ;

// Source: §18.4.1 General
interface_member_declaration
    : interface_method_declaration
    | interface_property_declaration
    | interface_event_declaration
    | interface_indexer_declaration
    ;

// Source: §18.4.2 Interface methods
interface_method_declaration
    : attributes? 'new'? return_type interface_method_header
    | attributes? 'new'? ref_kind ref_return_type interface_method_header
    ;

interface_method_header
    : identifier '(' formal_parameter_list? ')' ';'
    | identifier type_parameter_list '(' formal_parameter_list? ')'
type_parameter_constraints_clause* ';'
    ;

// Source: §18.4.3 Interface properties
interface_property_declaration
    : attributes? 'new'? type identifier '{' interface_accessors '}'
    | attributes? 'new'? ref_kind type identifier '{' ref_interface_accessor
'}'
    ;

interface_accessors
    : attributes? 'get' ';'
    | attributes? 'set' ';'
    | attributes? 'get' ';' attributes? 'set' ';'
    | attributes? 'set' ';' attributes? 'get' ';'
    ;

ref_interface_accessor
    : attributes? 'get' ';'
    ;

// Source: §18.4.4 Interface events
interface_event_declaration
    : attributes? 'new'? 'event' type identifier ';'
    ;

// Source: §18.4.5 Interface indexers
```

```
interface_indexer_declaration
    : attributes? 'new'? type 'this' '[' formal_parameter_list ']'
      '{' interface_accessors '}'
    | attributes? 'new'? ref_kind type 'this' '[' formal_parameter_list ']'
      '{' ref_interface_accessor '}'
    ;

// Source: §19.2 Enum declarations
enum_declaration
    : attributes? enum_modifier* 'enum' identifier enum_base? enum_body ';'?
    ;

enum_base
    : ':' integral_type
    | ':' integral_type_name
    ;

integral_type_name
    : type_name // Shall resolve to an integral type other than char
    ;

enum_body
    : '{' enum_member_declarations? '}'
    | '{' enum_member_declarations ',' '}'
    ;

// Source: §19.3 Enum modifiers
enum_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    ;

// Source: §19.4 Enum members
enum_member_declarations
    : enum_member_declaration (',' enum_member_declaration)*
    ;

// Source: §19.4 Enum members
enum_member_declaration
    : attributes? identifier ('=' constant_expression)?
    ;

// Source: §20.2 Delegate declarations
delegate_declaration
    : attributes? delegate_modifier* 'delegate' return_type delegate_header
    | attributes? ref_delegate_modifier* 'delegate' ref_kind ref_return_type
delegate_header
    ;

delegate_header
    : identifier '(' formal_parameter_list? ')' ';'
    | identifier variant_type_parameter_list '(' formal_parameter_list? ')'
```

```
type_parameter_constraints_clause* ';'
    ;

delegate_modifier
    : ref_delegate_modifier
    | unsafe_modifier   // unsafe code support
    ;

ref_delegate_modifier
    : 'new'
    | 'public'
    | 'protected'
    | 'internal'
    | 'private'
    ;

// Source: §22.3 Attribute specification
global_attributes
    : global_attribute_section+
    ;

global_attribute_section
    : '[' global_attribute_target_specifier attribute_list ']'
    | '[' global_attribute_target_specifier attribute_list ',' ']'
    ;

global_attribute_target_specifier
    : global_attribute_target ':'
    ;

global_attribute_target
    : identifier
    ;

attributes
    : attribute_section+
    ;

attribute_section
    : '[' attribute_target_specifier? attribute_list ']'
    | '[' attribute_target_specifier? attribute_list ',' ']'
    ;

attribute_target_specifier
    : attribute_target ':'
    ;

attribute_target
    : identifier
    | keyword
    ;

attribute_list
    : attribute (',' attribute)*
    ;
```

```
attribute
    : attribute_name attribute_arguments?
    ;

attribute_name
    : type_name
    ;

attribute_arguments
    : '(' positional_argument_list? ')'
    | '(' positional_argument_list ',' named_argument_list ')'
    | '(' named_argument_list ')'
    ;

positional_argument_list
    : positional_argument (',' positional_argument)*
    ;

positional_argument
    : argument_name? attribute_argument_expression
    ;

named_argument_list
    : named_argument (','  named_argument)*
    ;

named_argument
    : identifier '=' attribute_argument_expression
    ;

attribute_argument_expression
    : expression
    ;
```

A.4 Grammar extensions for unsafe code

ANTLR

```
// Source: §23.2 Unsafe contexts
unsafe_modifier
    : 'unsafe'
    ;

unsafe_statement
    : 'unsafe' block
    ;

// Source: §23.3 Pointer types
pointer_type
    : value_type ('*')+
```

```
    | 'void' ('*')+
    ;

// Source: §23.6.2 Pointer indirection
pointer_indirection_expression
    : '*' unary_expression
    ;

// Source: §23.6.3 Pointer member access
pointer_member_access
    : primary_expression '->' identifier type_argument_list?
    ;

// Source: §23.6.4 Pointer element access
pointer_element_access
    : primary_no_array_creation_expression '[' expression ']'
    ;

// Source: §23.6.5 The address-of operator
addressof_expression
    : '&' unary_expression
    ;

// Source: §23.7 The fixed statement
fixed_statement
    : 'fixed' '(' pointer_type fixed_pointer_declarators ')'
embedded_statement
    ;

fixed_pointer_declarators
    : fixed_pointer_declarator (','  fixed_pointer_declarator)*
    ;

fixed_pointer_declarator
    : identifier '=' fixed_pointer_initializer
    ;

fixed_pointer_initializer
    : '&' variable_reference
    | expression
    ;

// Source: §23.8.2 Fixed-size buffer declarations
fixed_size_buffer_declaration
    : attributes? fixed_size_buffer_modifier* 'fixed' buffer_element_type
      fixed_size_buffer_declarators ';'
    ;

fixed_size_buffer_modifier
    : 'new'
    | 'public'
    | 'internal'
    | 'private'
    | 'unsafe'
    ;
```

```
buffer_element_type
    : type
    ;

fixed_size_buffer_declarators
    : fixed_size_buffer_declarator (',' fixed_size_buffer_declarator)*
    ;

fixed_size_buffer_declarator
    : identifier '[' constant_expression ']'
    ;
```

End of informative text.

Annex B Portability issues

Article • 07/14/2023

This clause is informative.

B.1 General

This annex collects some information about portability that appears in this specification.

B.2 Undefined behavior

The behavior is undefined in the following circumstances:

1. The behavior of the enclosing async function when an awaiter's implementation of the interface methods `INotifyCompletion.OnCompleted` and `ICriticalNotifyCompletion.UnsafeOnCompleted` does not cause the resumption delegate to be invoked at most once (§12.9.8.4).
2. Passing pointers as `ref` or `out` parameters (§23.3).
3. When dereferencing the result of converting one pointer type to another and the resulting pointer is not correctly aligned for the pointed-to type. (§23.5.1).
4. When the unary `*` operator is applied to a pointer containing an invalid value (§23.6.2).
5. When a pointer is subscripted to access an out-of-bounds element (§23.6.4).
6. Modifying objects of managed type through fixed pointers (§23.7).
7. The content of memory newly allocated by `stackalloc` (§12.8.21).
8. Attempting to allocate a negative number of items using `stackalloc` (§12.8.21).
9. Implicit dynamic conversions (§10.2.10) of `in` parameters with value arguments (§12.6.4.2).

B.3 Implementation-defined behavior

A conforming implementation is required to document its choice of behavior in each of the areas listed in this subclause. The following are implementation-defined:

1. The behavior when an identifier not in Normalization Form C is encountered (§6.4.3).
2. The maximum value allowed for `Decimal_Digit+` in `PP_Line_Indicator` (§6.5.8).
3. The interpretation of the *input_characters* in the *pp_pragma-text* of a #pragma directive (§6.5.9).

4. The values of any application parameters passed to `Main` by the host environment prior to application startup (§7.1).
5. The precise structure of the expression tree, as well as the exact process for creating it, when an anonymous function is converted to an expression-tree (§10.7.3).
6. Whether a `System.ArithmeticException` (or a subclass thereof) is thrown or the overflow goes unreported with the resulting value being that of the left operand, when in an `unchecked` context and the left operand of an integer division is the maximum negative `int` or `long` value and the right operand is `-1` (§12.10.3).
7. When a `System.ArithmeticException` (or a subclass thereof) is thrown when performing a decimal remainder operation (§12.10.4).
8. The impact of thread termination when a thread has no handler for an exception, and the thread is itself terminated (§13.10.6).
9. The impact of thread termination when no matching `catch` clause is found for an exception and the code that initially started that thread is reached. (§21.4).
10. The mappings between pointers and integers (§23.5.1).
11. The effect of applying the unary `*` operator to a `null` pointer (§23.6.2).
12. The behavior when pointer arithmetic overflows the domain of the pointer type (§23.6.6, §23.6.7).
13. The result of the `sizeof` operator for non-pre-defined value types (§23.6.9).
14. The behavior of the `fixed` statement if the array expression is `null` or if the array has zero elements (§23.7).
15. The behavior of the `fixed` statement if the string expression is `null` (§23.7).
16. The value returned when a stack allocation of size zero is made (§12.8.21).

B.4 Unspecified behavior

1. The time at which the finalizer (if any) for an object is run, once that object has become eligible for finalization (§7.9).
2. The representation of `true` (§8.3.9).
3. The value of the result when converting out-of-range values from `float` or `double` values to an integral type in an `unchecked` context (§10.3.2).
4. The exact target object and target method of the delegate produced from an *anonymous_method_expression* contains (§10.7.2).
5. The layout of arrays, except in an unsafe context (§12.8.16.5).
6. Whether there is any way to execute the *block* of an anonymous function other than through evaluation and invocation of the *lambda_expression* or *anonymous_method-expression* (§12.19.3).
7. The exact timing of static field initialization (§15.5.6.2).

8. The result of invoking `MoveNext` when an enumerator object is running (§15.14.5.2).
9. The result of accessing `Current` when an enumerator object is in the before, running, or after states (§15.14.5.3).
10. The result of invoking `Dispose` when an enumerator object is in the running state (§15.14.5.4).
11. The attributes of a type declared in multiple parts are determined by combining, in an unspecified order, the attributes of each of its parts (§22.3).
12. The order in which members are packed into a struct (§23.6.9).
13. An exception occurs during finalizer execution, and that exception is not caught (§21.4).
14. If more than one member matches, which member is the implementation of `I.M` (§18.6.5).

B.5 Other issues

1. The exact results of floating-point expression evaluation can vary from one implementation to another, because an implementation is permitted to evaluate such expressions using a greater range and/or precision than is required (§8.3.7).
2. The CLI reserves certain signatures for compatibility with other programming languages (§15.3.10).

End of informative text.

Annex C Standard library

Article • 04/07/2023

C.1 General

A conforming C# implementation shall provide a minimum set of types having specific semantics. These types and their members are listed here, in alphabetical order by namespace and type. For a formal definition of these types and their members, refer to ISO/IEC 23271:2012 *Common Language Infrastructure (CLI), Partition IV; Base Class Library (BCL), Extended Numerics Library, and Extended Array Library*, which are included by reference in this specification.

This text is informative.

The standard library is intended to be the minimum set of types and members required by a conforming C# implementation. As such, it contains only those members that are explicitly required by the C# language specification.

It is expected that a conforming C# implementation will supply a significantly more extensive library that enables useful programs to be written. For example, a conforming implementation might extend this library by

- Adding namespaces.
- Adding types.
- Adding members to non-interface types.
- Adding intervening base classes or interfaces.
- Having struct and class types implement additional interfaces.
- Adding attributes (other than the `ConditionalAttribute`) to existing types and members.

End of informative text.

C.2 Standard Library Types defined in ISO/IEC 23271

> *Note:* Some `struct` types below have the `readonly` modifier. This modifier was not available when ISO/IEC 23271 was released, but is required for conforming implementations of this specification. *end note*

C#

```
namespace System
{
    public delegate void Action();

    public class ArgumentException : SystemException
    {
        public ArgumentException();
        public ArgumentException(string message);
        public ArgumentException(string message, Exception innerException);
    }

    public class ArithmeticException : Exception
    {
        public ArithmeticException();
        public ArithmeticException(string message);
        public ArithmeticException(string message, Exception
innerException);
    }

    public abstract class Array : IList, ICollection, IEnumerable
    {
        public int Length { get; }
        public int Rank { get; }
        public int GetLength(int dimension);
    }

    public class ArrayTypeMismatchException : Exception
    {
        public ArrayTypeMismatchException();
        public ArrayTypeMismatchException(string message);
        public ArrayTypeMismatchException(string message,
            Exception innerException);
    }

    [AttributeUsageAttribute(AttributeTargets.All, Inherited = true,
        AllowMultiple = false)]
    public abstract class Attribute
    {
        protected Attribute();
    }

    public enum AttributeTargets
    {
        Assembly = 0x1,
        Module = 0x2,
        Class = 0x4,
        Struct = 0x8,
        Enum = 0x10,
        Constructor = 0x20,
        Method = 0x40,
        Property = 0x80,
        Field = 0x100,
```

```
        Event = 0x200,
        Interface = 0x400,
        Parameter = 0x800,
        Delegate = 0x1000,
        ReturnValue = 0x2000,
        GenericParameter = 0x4000,
        All = 0x7FFF
    }

    [AttributeUsageAttribute(AttributeTargets.Class, Inherited = true)]
    public sealed class AttributeUsageAttribute : Attribute
    {
        public AttributeUsageAttribute(AttributeTargets validOn);
        public bool AllowMultiple { get; set; }
        public bool Inherited { get; set; }
        public AttributeTargets ValidOn { get; }
    }

    public readonly struct Boolean { }
    public readonly struct Byte { }
    public readonly struct Char { }
    public readonly struct Decimal { }
    public abstract class Delegate { }

    public class DivideByZeroException : ArithmeticException
    {
        public DivideByZeroException();
        public DivideByZeroException(string message);
        public DivideByZeroException(string message, Exception
innerException);
    }

    public readonly struct Double { }

    public abstract class Enum : ValueType
    {
        protected Enum();
    }

    public class Exception
    {
        public Exception();
        public Exception(string message);
        public Exception(string message, Exception innerException);
        public sealed Exception InnerException { get; }
        public virtual string Message { get; }
    }

    public class GC { }

    public interface IDisposable
    {
        void Dispose();
    }
```

```
    public interface IFormattable { }

    public sealed class IndexOutOfRangeException : Exception
    {
        public IndexOutOfRangeException();
        public IndexOutOfRangeException(string message);
        public IndexOutOfRangeException(string message,
            Exception innerException);
    }

    public readonly struct Int16 { }
    public readonly struct Int32 { }
    public readonly struct Int64 { }
    public readonly struct IntPtr { }

    public class InvalidCastException : Exception
    {
        public InvalidCastException();
        public InvalidCastException(string message);
        public InvalidCastException(string message, Exception
innerException);
    }

    public class InvalidOperationException : Exception
    {
        public InvalidOperationException();
        public InvalidOperationException(string message);
        public InvalidOperationException(string message,
            Exception innerException);
    }

    public class NotSupportedException : Exception
    {
        public NotSupportedException();
        public NotSupportedException(string message);
        public NotSupportedException(string message, Exception
innerException);
    }

    public struct Nullable<T>
    {
        public bool HasValue { get; }
        public T Value { get; }
    }

    public class NullReferenceException : Exception
    {
        public NullReferenceException();
        public NullReferenceException(string message);
        public NullReferenceException(string message, Exception
innerException);
    }

    public class Object
    {
```

```
        public Object();
        ~Object();
        public virtual bool Equals(object obj);
        public virtual int GetHashCode();
        public Type GetType();
        public virtual string ToString();
    }

    [AttributeUsageAttribute(AttributeTargets.Class |
AttributeTargets.Struct |
        AttributeTargets.Enum | AttributeTargets.Interface |
        AttributeTargets.Constructor | AttributeTargets.Method |
        AttributeTargets.Property | AttributeTargets.Field |
        AttributeTargets.Event | AttributeTargets.Delegate, Inherited =
false)]
    public sealed class ObsoleteAttribute : Attribute
    {
        public ObsoleteAttribute();
        public ObsoleteAttribute(string message);
        public ObsoleteAttribute(string message, bool error);
        public bool IsError { get; }
        public string Message { get; }
    }

    public class OutOfMemoryException : Exception
    {
        public OutOfMemoryException();
        public OutOfMemoryException(string message);
        public OutOfMemoryException(string message, Exception
innerException);
    }

    public class OverflowException : ArithmeticException
    {
        public OverflowException();
        public OverflowException(string message);
        public OverflowException(string message, Exception innerException);
    }

    public readonly struct SByte { }
    public readonly struct Single { }

    public sealed class StackOverflowException : Exception
    {
        public StackOverflowException();
        public StackOverflowException(string message);
        public StackOverflowException(string message, Exception
innerException);
    }

    public sealed class String : IEnumerable<Char>, IEnumerable
    {
        public int Length { get; }
        public char this [int index] { get; }
        public static string Format(string format, params object[] args);
```

```
    }

    public abstract class Type : MemberInfo { }

    public sealed class TypeInitializationException : Exception
    {
        public TypeInitializationException(string fullTypeName,
            Exception innerException);
    }

    public readonly struct UInt16 { }
    public readonly struct UInt32 { }
    public readonly struct UInt64 { }
    public readonly struct UIntPtr { }

    public struct ValueTuple<T1>
    {
        public T1 Item1;
        public ValueTuple(T1 item1);
    }
    public struct ValueTuple<T1, T2>
    {
        public T1 Item1;
        public T2 Item2;
        public ValueTuple(T1 item1, T2 item2);
    }
    public struct ValueTuple<T1, T2, T3>
    {
        public T1 Item1;
        public T2 Item2;
        public T3 Item3;
        public ValueTuple(T1 item1, T2 item2, T3 item3);
    }
    public struct ValueTuple<T1, T2, T3, T4>
    {
        public T1 Item1;
        public T2 Item2;
        public T3 Item3;
        public T4 Item4;
        public ValueTuple(T1 item1, T2 item2, T3 item3, T4 item4);
    }
    public struct ValueTuple<T1, T2, T3, T4, T5>
    {
        public T1 Item1;
        public T2 Item2;
        public T3 Item3;
        public T4 Item4;
        public T5 Item5;
        public ValueTuple(T1 item1, T2 item2, T3 item3, T4 item4, T5 item5);
    }
    public struct ValueTuple<T1, T2, T3, T4, T5, T6>
    {
        public T1 Item1;
        public T2 Item2;
        public T3 Item3;
```

```
        public T4 Item4;
        public T5 Item5;
        public T6 Item6;
        public ValueTuple(T1 item1, T2 item2, T3 item3, T4 item4, T5 item5,
T6 item6);
    }
    public struct ValueTuple<T1, T2, T3, T4, T5, T6, T7>
    {
        public T1 Item1;
        public T2 Item2;
        public T3 Item3;
        public T4 Item4;
        public T5 Item5;
        public T6 Item6;
        public T7 Item7;
        public ValueTuple(T1 item1, T2 item2, T3 item3, T4 item4, T5 item5,
T6 item6, T7 item7);
    }
    public struct ValueTuple<T1, T2, T3, T4, T5, T6, T7, TRest>
    {
        public T1 Item1;
        public T2 Item2;
        public T3 Item3;
        public T4 Item4;
        public T5 Item5;
        public T6 Item6;
        public T7 Item7;
        public TRest Rest;
        public ValueTuple(T1 item1, T2 item2, T3 item3, T4 item4, T5 item5,
T6 item6, T7 item7, TRest rest);
    }

    public abstract class ValueType
    {
        protected ValueType();
    }
}

namespace System.Collections
{
    public interface ICollection : IEnumerable
    {
        int Count { get; }
        bool IsSynchronized { get; }
        object SyncRoot { get; }
        void CopyTo(Array array, int index);
    }

    public interface IEnumerable
    {
        IEnumerator GetEnumerator();
    }

    public interface IEnumerator
    {
```

```
        object Current { get; }
        bool MoveNext();
        void Reset();
    }

    public interface IList : ICollection, IEnumerable
    {
        bool IsFixedSize { get; }
        bool IsReadOnly { get; }
        object this [int index] { get; set; }
        int Add(object value);
        void Clear();
        bool Contains(object value);
        int IndexOf(object value);
        void Insert(int index, object value);
        void Remove(object value);
        void RemoveAt(int index);
    }
}

namespace System.Collections.Generic
{
    public interface ICollection<T> : IEnumerable<T>
    {
        int Count { get; }
        bool IsReadOnly { get; }
        void Add(T item);
        void Clear();
        bool Contains(T item);
        void CopyTo(T[] array, int arrayIndex);
        bool Remove(T item);
    }

    public interface IEnumerable<T> : IEnumerable
    {
        IEnumerator<T> GetEnumerator();
    }

    public interface IEnumerator<T> : IDisposable, IEnumerator
    {
        T Current { get; }
    }

    public interface IList<T> : ICollection<T>
    {
        T this [int index] { get; set; }
        int IndexOf(T item);
        void Insert(int index, T item);
        void RemoveAt(int index);
    }

    public interface IReadOnlyCollection<out T> : IEnumerable<T>
    {
        int Count { get; }
    }
```

```
    public interface IReadOnlyList<out T> : IReadOnlyCollection<T>
    {
        T this [int index] { get; }
    }
}

namespace System.Diagnostics
{
    [AttributeUsageAttribute(AttributeTargets.Method |
AttributeTargets.Class,
                             AllowMultiple = true)]
    public sealed class ConditionalAttribute : Attribute
    {
        public ConditionalAttribute(string conditionString);
        public string ConditionString { get; }
    }
}

namespace System.Reflection
{
    public abstract class MemberInfo
    {
        protected MemberInfo();
    }
}

namespace System.Runtime.CompilerServices
{
    public sealed class IndexerNameAttribute : Attribute
    {
        public IndexerNameAttribute(String indexerName);
    }

    public static class Unsafe
    {
        public static ref T NullRef<T>();
    }
}

namespace System.Threading
{
    public static class Monitor
    {
        public static void Enter(object obj);
        public static void Exit(object obj);
    }
}
```

C.3 Standard Library Types not defined in ISO/IEC 23271

The following types, including the members listed, must be defined in a conforming standard library. (These types might be defined in a future edition of ISO/IEC 23271.) It is expected that many of these types will have more members available than are listed.

A conforming implementation may provide `Task.GetAwaiter()` and `Task<TResult>.GetAwaiter()` as extension methods.

C#

```
namespace System
{
    public class FormattableString : IFormattable { }
}

namespace System.Linq.Expressions
{
    public sealed class Expression<TDelegate>
    {
        public TDelegate Compile();
    }
}
namespace System.Runtime.CompilerServices
{
    [AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct |
        AttributeTargets.Interface,
        Inherited = false, AllowMultiple = false)]
    public sealed class AsyncMethodBuilderAttribute : Attribute
    {
        public AsyncMethodBuilderAttribute(Type builderType) {}

        public Type BuilderType { get; }
    }

    [AttributeUsage(AttributeTargets.Parameter, Inherited = false)]
    public sealed class CallerFilePathAttribute : Attribute
    {
        public CallerFilePathAttribute() { }
    }

    [AttributeUsage(AttributeTargets.Parameter, Inherited = false)]
    public sealed class CallerLineNumberAttribute : Attribute
    {
        public CallerLineNumberAttribute() { }
    }

    [AttributeUsage(AttributeTargets.Parameter, Inherited = false)]
    public sealed class CallerMemberNameAttribute : Attribute
    {
        public CallerMemberNameAttribute() { }
    }

    public static class FormattableStringFactory
    {
```

```
        public static FormattableString Create(string format,
            params object[] arguments);
    }

    public interface ICriticalNotifyCompletion : INotifyCompletion
    {
        void UnsafeOnCompleted(Action continuation);
    }

    public interface INotifyCompletion
    {
        void OnCompleted(Action continuation);
    }

    public readonly struct TaskAwaiter : ICriticalNotifyCompletion,
INotifyCompletion
    {
        public bool IsCompleted { get; }
        public void GetResult();
    }

    public readonly struct TaskAwaiter<TResult> : ICriticalNotifyCompletion,
INotifyCompletion
    {
        public bool IsCompleted { get; }
        public TResult GetResult();
    }

    public readonly struct ValueTaskAwaiter : ICriticalNotifyCompletion,
INotifyCompletion
    {
        public bool IsCompleted { get; }
        public void GetResult();
    }

    public readonly struct ValueTaskAwaiter<TResult> :
ICriticalNotifyCompletion, INotifyCompletion
    {
        public bool IsCompleted { get; }
        public TResult GetResult();
    }

}

namespace System.Threading.Tasks
{
    public class Task
    {
        public System.Runtime.CompilerServices.TaskAwaiter GetAwaiter();
    }
    public class Task<TResult> : Task
    {
        public new System.Runtime.CompilerServices.TaskAwaiter<T>
GetAwaiter();
    }
```

```
    public readonly struct ValueTask : System.IEquatable<ValueTask>
    {
        public System.Runtime.CompilerServices.ValueTaskAwaiter
GetAwaiter();
    }
    public readonly struct ValueTask<TResult> :
System.IEquatable<ValueTask<TResult>>
    {
        public new System.Runtime.CompilerServices.ValueTaskAwaiter<TResult>
GetAwaiter();
    }
}
```

C#

```
namespace System
{
    public readonly ref struct ReadOnlySpan<T>
    {
        public int Length { get; }
        public ref readonly T this[int index] { get; }
    }
    public readonly ref struct Span<T>
    {
        public int Length { get; }
        public ref T this[int index] { get; }
        public static implicit operator ReadOnlySpan<T>(Span<T> span);
    }
}
```

C.4 Format Specifications

The meaning of the formats, as used in interpolated string expressions (§12.8.3), are defined in ISO/IEC 23271:2012. For convenience the following text is copied from the description of `System.IFormatable`.

This text is informative.

A *format* is a string that describes the appearance of an object when it is converted to a string. Either standard or custom formats can be used. A standard format takes the form *Axx*, where *A* is a single alphabetic character called the *format specifier*, and *xx* is an integer between zero and 99 inclusive, called the *precision specifier*. The format specifier controls the type of formatting applied to the value being represented as a string. The *precision specifier* controls the number of significant digits or decimal places in the string, if applicable.

> *Note*: For the list of standard format specifiers, see the table below. Note that a given data type, such as `System.Int32`, might not support one or more of the standard format specifiers. *end note*

> *Note*: When a format includes symbols that vary by culture, such as the currencysymbol included by the 'C' and 'c' formats, a formatting object supplies the actual characters used in the string representation. A method might include a parameter to pass a `System.IFormatProvider` object that supplies a formatting object, or the method might use the default formatting object, which contains the symbol definitions for the current culture. The current culture typically uses the same set of symbols used system-wide by default. In the Base Class Library, the formatting object for system-supplied numeric types is a `System.Globalization.NumberFormatInfo` instance. For `System.DateTime` instances, a `System.Globalization.DateTimeFormatInfo` is used. *end note*

The following table describes the standard format specifiers and associated formatting object members that are used with numeric data types in the Base Class Library.

Format Specifier	Description

`C`
 	
`c`
	Currency Format: Used for strings containing a monetary value. The `System.Globalization.NumberFormatInfo.CurrencySymbol`, `System.Globalization.NumberFormatInfo.CurrencyGroupSizes`, `System.Globalization.NumberFormatInfo.CurrencyGroupSeparator`, and `System.Globalization.NumberFormatInfo.CurrencyDecimalSeparator` members of a `System.Globalization.NumberFormatInfo` supply the currency symbol, size and separator for digit groupings, and decimal separator, respectively.
	
	`System.Globalization.NumberFormatInfo.CurrencyNegativePattern` and `System.Globalization.NumberFormatInfo.CurrencyPositivePattern` determine the symbols used to represent negative and positive values. For example, a negative value can be prefixed with a minus sign, or enclosed in parentheses.
	
	If the precision specifier is omitted, `System.Globalization.NumberFormatInfo.CurrencyDecimalDigits` determines the number of decimal places in the string. Results are rounded to the nearest representable value when necessary.

`D`
 	
`d`
	Decimal Format: (This format is valid only when specified with integral data types.) Used for strings containing integer values. Negative numbers are prefixed with the negative number symbol specified by the `System.Globalization.NumberFormatInfo.NegativeSign` property.

	The precision specifier determines the minimum number of digits that appear in the string. If the specified precision requires more digits than the value contains, the string is left-padded with zeros. If the precision specifier specifies fewer digits than are in the value, the precision specifier is ignored.
`E` `e`	**Scientific (Engineering) Format:** Used for strings in one of the following forms: [-]*m.ddddd*E+*xxx* [-]*m.ddddd*E-*xxx* [-]*m.ddddd*e+*xxx* [-]*m.ddddd*e-*xxx* The negative number symbol ('-') appears only if the value is negative, and is supplied by the `System.Globalization.NumberFormatInfo.NegativeSign` property. Exactly one non-zero decimal digit (*m*) precedes the decimal separator ('.'), which is supplied by the `System.Globalization.NumberFormatInfo.NumberDecimalSeparator` property. The precision specifier determines the number of decimal places (*ddddd*) in the string. If the precision specifier is omitted, six decimal places are included in the string. The exponent (*+/-xxx*) consists of either a positive or negative number symbol followed by a minimum of three digits (*xxx*). The exponent is left-padded with zeros, if necessary. The case of the format specifier ('E' or 'e') determines the case used for the exponent prefix (E or e) in the string. Results are rounded to the nearest representable value when necessary. The positive number symbol is supplied by the `System.Globalization.NumberFormatInfo.PositiveSign` property.
`F` `f`	**Fixed-Point Format:** Used for strings in the following form: [-]*m.dd...d* At least one non-zero decimal digit (*m*) precedes the decimal separator ('.'), which is supplied by the `System.Globalization.NumberFormatInfo.NumberDecimalSeparator` property. A negative number symbol sign ('-') precedes *m* only if the value is negative. This symbol is supplied by the `System.Globalization.NumberFormatInfo.NegativeSign` property. The precision specifier determines the number of decimal places (*dd...d*) in the string. If the precision specifier is omitted, `System.Globalization.NumberFormatInfo.NumberDecimalDigits` determines the number of decimal places in the string. Results are rounded to the nearest representable value when necessary.
`G`	**General Format:** The string is formatted in either fixed-point format ('F' or 'f') or scientific format ('E' or 'e').

`g`	For integral types: Values are formatted using fixed-point format if *exponent* < precision specifier, where *exponent* is the exponent of the value in scientific format. For all other values, scientific format is used. If the precision specifier is omitted, a default precision equal to the field width required to display the maximum value for the data type is used, which results in the value being formatted in fixed-point format. The default precisions for integral types are as follows: `System.Int16`, `System.UInt16` : 5 `System.Int32`, `System.UInt32` : 10 `System.Int64`, `System.UInt64` : 19 For Single, Decimal and Double types: Values are formatted using fixed-point format if *exponent* ≥ -4 and *exponent* < precision specifier, where *exponent* is the exponent of the value in scientific format. For all other values, scientific format is used. Results are rounded to the nearest representable value when necessary. If the precision specifier is omitted, the following default precisions are used: `System.Single` : 7 `System.Double` : 15 `System.Decimal` : 29 For all types: • The number of digits that appear in the result (not including the exponent) will not exceed the value of the precision specifier; values are rounded as necessary. • The decimal point and any trailing zeros after the decimal point are removed whenever possible. • The case of the format specifier ('G' or 'g') determines whether 'E' or 'e' prefixes the scientific format exponent.

`N`
 	
`n`
	Number Format: Used for strings in the following form:
	
	[-]*d,ddd,ddd.dd...d*
	
	The representation of negative values is determined by the `System.Globalization.NumberFormatInfo.NumberNegativePattern` property. If the pattern includes a negative number symbol ('-'), this symbol is supplied by the `System.Globalization.NumberFormatInfo.NegativeSign` property.
	
	At least one non-zero decimal digit (*d*) precedes the decimal separator ('.'), which is supplied by the `System.Globalization.NumberFormatInfo.NumberDecimalSeparator`

	property. Digits between the decimal point and the most significant digit in the value are grouped using the group size specified by the `System.Globalization.NumberFormatInfo.NumberGroupSizes` property. The group separator (',') is inserted between each digit group, and is supplied by the `System.Globalization.NumberFormatInfo.NumberGroupSeparator` property. The precision specifier determines the number of decimal places (*dd...d*). If the precision specifier is omitted, `System.Globalization.NumberFormatInfo.NumberDecimalDigits` determines the number of decimal places in the string. Results are rounded to the nearest representable value when necessary.

`P`
 	
`p`
	Percent Format: Used for strings containing a percentage. The `System.Globalization.NumberFormatInfo.PercentSymbol`, `System.Globalization.NumberFormatInfo.PercentGroupSizes`, `System.Globalization.NumberFormatInfo.PercentGroupSeparator`, and `System.Globalization.NumberFormatInfo.PercentDecimalSeparator` members of a `System.Globalization.NumberFormatInfo` supply the percent symbol, size and separator for digit groupings, and decimal separator, respectively.
	
	`System.Globalization.NumberFormatInfo.PercentNegativePattern` and `System.Globalization.NumberFormatInfo.PercentPositivePattern` determine the symbols used to represent negative and positive values. For example, a negative value can be prefixed with a minus sign, or enclosed in parentheses.
	
	If no precision is specified, the number of decimal places in the result is determined by `System.Globalization.NumberFormatInfo.PercentDecimalDigits`. Results are rounded to the nearest representable value when necessary.
	
	The result is scaled by 100 (.99 becomes 99%).

`R`
 	
`r`
	Round trip Format: (This format is valid only when specified with `System.Double` or `System.Single`.) Used to ensure that the precision of the string representation of a floating-point value is such that parsing the string does not result in a loss of precision when compared to the original value. If the maximum precision of the data type (7 for `System.Single`, and 15 for `System.Double`) would result in a loss of precision, the precision is increased by two decimal places. If a precision specifier is supplied with this format specifier, it is ignored. This format is otherwise identical to the fixed-point format.

`X`
 	
`x`
	Hexadecimal Format: (This format is valid only when specified with integral data types.) Used for string representations of numbers in Base 16. The precision determines the minimum number of digits in the string. If the precision specifies more digits than the number contains, the number is left-padded with zeros. The case of the format specifier ('X' or 'x') determines whether upper case or lower case letters are used in the hexadecimal representation.

If the numerical value is a `System.Single` or `System.Double` with a value of `NaN`, `PositiveInfinity`, or `NegativeInfinity`, the format specifier is ignored, and one of the

following is returned: `System.Globalization.NumberFormatInfo.NaNSymbol`, `System.Globalization.NumberFormatInfo.PositiveInfinitySymbol`, or `System.Globalization.NumberFormatInfo.NegativeInfinitySymbol`.

A custom format is any string specified as a format that is not in the form of a standard format string (Axx) described above. The following table describes the characters that are used in constructing custom formats.

Format Specifier	Description
`0` (zero)	**Zero placeholder:** If the value being formatted has a digit in the position where a '0' appears in the custom format, then that digit is copied to the output string; otherwise a zero is stored in that position in the output string. The position of the leftmost '0' before the decimal separator and the rightmost '0' after the decimal separator determine the range of digits that are always present in the output string. The number of Zero and/or Digit placeholders after the decimal separator determines the number of digits that appear after the decimal separator. Values are rounded as necessary.
`#`	**Digit placeholder:** If the value being formatted has a digit in the position where a '#' appears in the custom format, then that digit is copied to the output string; otherwise, nothing is stored in that position in the output string. Note that this specifier never stores the '0' character if it is not a significant digit, even if '0' is the only digit in the string. (It does display the '0' character in the output string if it is a significant digit.) The number of Zero and/or Digit placeholders after the decimal separator determines the number of digits that appear after the decimal separator. Values are rounded as necessary.
`.` (period)	**Decimal separator:** The left most '.' character in the format string determines the location of the decimal separator in the formatted value; any additional '.' characters are ignored. The `System.Globalization.NumberFormatInfo.NumberDecimalSeparator` property determines the symbol used as the decimal separator.
`,` (comma)	**Group separator and number scaling:** The ',' character serves two purposes. First, if the custom format contains this character between two Zero or Digit placeholders (0 or #) and to the left of the decimal separator if one is present, then the output will have group separators inserted between each group of digits to the left of the decimal separator. The `System.Globalization.NumberFormatInfo.NumberGroupSeparator` and `System.Globalization.NumberFormatInfo.NumberGroupSizes` properties determine the symbol used as the group separator and the number of digits in each group, respectively.

	If the format string contains one or more ‘,’ characters immediately to the left of the decimal separator, then the number will be scaled. The scale factor is determined by the number of group separator characters immediately to the left of the decimal separator. If there are x characters, then the value is divided by 1000^X before it is formatted. For example, the format string ‘0,,’ will divide a value by one million. Note that the presence of the ‘,’ character to indicate scaling does not insert group separators in the output string. Thus, to scale a number by 1 million and insert group separators, use a custom format similar to ‘#,##0,,’.
`%` (percent)	**Percentage placeholder:** The presence of a ‘%’ character in a custom format causes a number to be multiplied by 100 before it is formatted. The percent symbol is inserted in the output string at the location where the ‘%’ appears in the format string. The `System.Globalization.NumberFormatInfo.PercentSymbol` property determines the percent symbol.

`E0`
`E+0`
`E-0`
`e0`
`e+0`
`e-0`
	Engineering format: If any of the strings ‘E’, ‘E+’, ‘E-’, ‘e’, ‘e+’, or ‘e-’ are present in a custom format and is followed immediately by at least one ‘0’ character, then the value is formatted using scientific notation. The number of ‘0’ characters following the exponent prefix (E or e) determines the minimum number of digits in the exponent. The ‘E+’ and ‘e+’ formats indicate that a positive or negative number symbol always precedes the exponent. The ‘E’, ‘E-’, ‘e’, or ‘e-’ formats indicate that a negative number symbol precedes negative exponents; no symbol is precedes positive exponents. The positive number symbol is supplied by the `System.Globalization.NumberFormatInfo.PositiveSign` property. The negative number symbol is supplied by the `System.Globalization.NumberFormatInfo.NegativeSign` property.

`\` (backslash)	**Escape character:** In some languages, such as C#, the backslash character causes the next character in the custom format to be interpreted as an escape sequence. It is used with C language formatting sequences, such as ‘\n’ (newline). In some languages, the escape character itself is required to be preceded by an escape character when used as a literal. Otherwise, the compiler interprets the character as an escape sequence. This escape character is not required to be supported in all programming languages.
`'ABC'` `"ABC"`	**Literal string:** Characters enclosed in single or double quotes are copied to the output string literally, and do not affect formatting.
`;` (semicolon)	**Section separator:** The ‘;’ character is used to separate sections for positive, negative, and zero numbers in the format string. (This feature is described in detail below.)
Other	**All other characters:** All other characters are stored in the output string as literals in the position in which they appear.

Note that for fixed-point format strings (strings not containing an ‘E0’, ‘E+0’, ‘E-0’, ‘e0’, ‘e+0’, or ‘e-0’), numbers are rounded to as many decimal places as there are Zero or Digit placeholders to the right of the decimal separator. If the custom format does not contain a decimal separator, the number is rounded to the nearest integer. If the

number has more digits than there are Zero or Digit placeholders to the left of the decimal separator, the extra digits are copied to the output string immediately before the first Zero or Digit placeholder.

A custom format can contain up to three sections separated by section separator characters, to specify different formatting for positive, negative, and zero values. The sections are interpreted as follows:

- **One section**: The custom format applies to all values (positive, negative and zero). Negative values include a negative sign.
- **Two sections**: The first section applies to positive values and zeros, and the second section applies to negative values. If the value to be formatted is negative, but becomes zero after rounding according to the format in the second section, then the resulting zero is formatted according to the first section. Negative values do not include a negative sign to allow full control over representations of negative values. For example, a negative can be represented in parenthesis using a custom format similar to '####.####;(####.####)'.
- **Three sections**: The first section applies to positive values, the second section applies to negative values, and the third section applies to zeros. The second section can be empty (nothing appears between the semicolons), in which case the first section applies to all nonzero values, and negative values include a negative sign. If the number to be formatted is nonzero, but becomes zero after rounding according to the format in the first or second section, then the resulting zero is formatted according to the third section.

The `System.Enum` and `System.DateTime` types also support using format specifiers to format string representations of values. The meaning of a specific format specifier varies according to the kind of data (numeric, date/time, enumeration) being formatted. See `System.Enum` and `System.Globalization.DateTimeFormatInfo` for a comprehensive list of the format specifiers supported by each type.

C.5 Library Type Abbreviations

The following library types are referenced in this specification. The full names of those types, including the global namespace qualifier are listed below. Throughout this specification, these types appear as either the fully qualified name; with the global namespace qualifier omitted; or as a simple unqualified type name, with the namespace omitted as well. For example, the type `ICollection<T>`, when used in this specification, always means the type `global::System.Collections.Generic.ICollection<T>`.

- `global::System.Action`
- `global::System.ArgumentException`
- `global::System.ArithmeticException`
- `global::System.Array`
- `global::System.ArrayTypeMisMatchException`
- `global::System.Attribute`
- `global::System.AttributeTargets`
- `global::System.AttributeUsageAttribute`
- `global::System.Boolean`
- `global::System.Byte`
- `global::System.Char`
- `global::System.Collections.Generic.ICollection<T>`
- `global::System.Collections.Generic.IEnumerable<T>`
- `global::System.Collections.Generic.IEnumerator<T>`
- `global::System.Collections.Generic.IList<T>`
- `global::System.Collections.Generic.IReadonlyCollection<out T>`
- `global::System.Collections.Generic.IReadOnlyList<out T>`
- `global::System.Collections.ICollection`
- `global::System.Collections.IEnumerable`
- `global::System.Collections.IList`
- `global::System.Collections.IEnumerator`
- `global::System.Decimal`
- `global::System.Delegate`
- `global::System.Diagnostics.ConditionalAttribute`
- `global::System.DivideByZeroException`
- `global::System.Double`
- `global::System.Enum`
- `global::System.Exception`
- `global::System.GC`
- `global::System.ICollection`
- `global::System.IDisposable`
- `global::System.IEnumerable`
- `global::System.IEnumerable<out T>`
- `global::System.IList`
- `global::System.IndexOutOfRangeException`
- `global::System.Int16`
- `global::System.Int32`
- `global::System.Int64`

- `global::System.IntPtr`
- `global::System.InvalidCastException`
- `global::System.InvalidOperationException`
- `global::System.Linq.Expressions.Expression<TDelegate>`
- `global::System.MemberInfo`
- `global::System.NotSupportedException`
- `global::System.Nullable<T>`
- `global::System.NullReferenceException`
- `global::System.Object`
- `global::System.ObsoleteAttribute`
- `global::System.OutOfMemoryException`
- `global::System.OverflowException`
- `global::System.Runtime.CompilerServices.CallerFileAttribute`
- `global::System.Runtime.CompilerServices.CallerLineNumberAttribute`
- `global::System.Runtime.CompilerServices.CallerMemberNameAttribute`
- `global::System.Runtime.CompilerServices.ICriticalNotifyCompletion`
- `global::System.Runtime.CompilerServices.IndexerNameAttribute`
- `global::System.Runtime.CompilerServices.INotifyCompletion`
- `global::System.Runtime.CompilerServices.TaskAwaiter`
- `global::System.Runtime.CompilerServices.TaskAwaiter<T>`
- `global::System.SByte`
- `global::System.Single`
- `global::System.StackOverflowException`
- `global::System.String`
- `global::System.SystemException`
- `global::System.Threading.Monitor`
- `global::System.Threading.Tasks.Task`
- `global::System.Threading.Tasks.Task<TResult>`
- `global::System.Type`
- `global::System.TypeInitializationException`
- `global::System.UInt16`
- `global::System.UInt32`
- `global::System.UInt64`
- `global::System.UIntPtr`
- `global::System.ValueType`

End of informative text.

Commentaires de documentation

Article • 13/05/2021

C# fournit un mécanisme permettant aux programmeurs de documenter leur code à l'aide d'une syntaxe de commentaire spéciale qui contient du texte XML. Dans les fichiers de code source, les commentaires ayant un certain format peuvent être utilisés pour indiquer à un outil de produire du code XML à partir de ces commentaires et des éléments de code source, qu'ils précèdent. Les commentaires utilisant une telle syntaxe sont appelés ***Commentaires de documentation** _. Ils doivent précéder immédiatement un type défini par l'utilisateur (par exemple, une classe, un délégué ou une interface) ou un membre (par exemple, un champ, un événement, une propriété ou une méthode). L'outil de génération XML est appelé le *Générateur de documentation*. (Ce générateur peut être, mais il n'est pas nécessaire, le compilateur C# lui-même.) La sortie produite par le générateur de documentation porte le nom de *fichier de documentation*. Un fichier de documentation est utilisé comme entrée de la *visionneuse de documentation** *; outil destiné à produire une sorte d'affichage visuel des informations de type et de la documentation associée.

Cette spécification suggère un ensemble de balises à utiliser dans les commentaires de documentation, mais l'utilisation de ces balises n'est pas obligatoire, et d'autres balises peuvent être utilisées si vous le souhaitez, dans la mesure où les règles de XML bien formé sont suivies.

Introduction

Les commentaires ayant une forme spéciale peuvent être utilisés pour indiquer à un outil de produire du code XML à partir de ces commentaires et des éléments de code source, qu'ils précèdent. Ces commentaires sont des commentaires sur une seule ligne qui commencent par trois barres obliques (`///`) ou des commentaires délimités qui commencent par une barre oblique et deux étoiles (`/**`). Ils doivent précéder immédiatement un type défini par l'utilisateur (par exemple, une classe, un délégué ou une interface) ou un membre (par exemple, un champ, un événement, une propriété ou une méthode) qu'ils annotent. Les sections d'attributs (spécification d'attribut) sont considérées comme faisant partie des déclarations. les commentaires de documentation doivent donc précéder les attributs appliqués à un type ou à un membre.

Syntaxe :

```
antlr
```

```
single_line_doc_comment
    : '///' input_character*
    ;

delimited_doc_comment
    : '/**' delimited_comment_section* asterisk+ '/'
    ;
```

Dans un *single_line_doc_comment*, s'il y a un *espace blanc* qui suit les `///` caractères sur chacun des *single_line_doc_comments* adjacents à la *single_line_doc_comment* actuelle, ce caractère d' *espace blanc* n'est pas inclus dans la sortie XML.

Dans un document de commentaire délimité, si le premier caractère autre qu'un espace blanc sur la deuxième ligne est un astérisque et le même modèle de caractères d'espace blanc facultatifs et qu'un astérisque est répété au début de chaque ligne dans le document délimité-doc-comment, les caractères du modèle répété ne sont pas inclus dans la sortie XML. Le modèle peut inclure des caractères d'espacement après, ainsi qu'avant, le caractère astérisque.

Exemple :

C#

```
/// <summary>Class <c>Point</c> models a point in a two-dimensional
/// plane.</summary>
///
public class Point
{
    /// <summary>method <c>draw</c> renders the point.</summary>
    void draw() {...}
}
```

Le texte des commentaires de documentation doit être correctement formé conformément aux règles de XML (https://www.w3.org/TR/REC-xml) . Si le code XML est incorrect, un avertissement est généré et le fichier de documentation contient un commentaire indiquant qu'une erreur a été rencontrée.

Bien que les développeurs soient libres de créer leur propre ensemble de balises, un ensemble recommandé est défini dans les Balises recommandées. Certaines des balises recommandées ont des significations spéciales :

- La `<param>` balise est utilisée pour décrire les paramètres. Si une telle balise est utilisée, le générateur de documentation doit vérifier que le paramètre spécifié existe et que tous les paramètres sont décrits dans commentaires de

documentation. En cas d'échec de cette vérification, le générateur de documentation émet un avertissement.

- L'attribut `cref` peut être joint à n'importe quelle balise pour fournir une référence à un élément de code. Le générateur de documentation doit vérifier que cet élément de code existe. Si la vérification échoue, le générateur de documentation émet un avertissement. Lors de la recherche d'un nom décrit dans un `cref` attribut, le générateur de documentation doit respecter la visibilité des espaces de noms en fonction `using` des instructions figurant dans le code source. Pour les éléments de code génériques, la syntaxe générique normale (autrement dit, « `List<T>` ») ne peut pas être utilisée car elle génère du code XML non valide. Les accolades peuvent être utilisées à la place des crochets (autrement dit, « `List{T}` »), ou la syntaxe d'échappement XML peut être utilisée (autrement dit, « `List<T>` »).
- La `<summary>` balise est destinée à être utilisée par une visionneuse de documentation pour afficher des informations supplémentaires sur un type ou un membre.
- La `<include>` balise contient des informations provenant d'un fichier XML externe.

Notez soigneusement que le fichier de documentation ne fournit pas d'informations complètes sur le type et les membres (par exemple, il ne contient pas d'informations de type). Pour obtenir de telles informations sur un type ou un membre, le fichier de documentation doit être utilisé conjointement avec la réflexion sur le type ou le membre réel.

Étiquettes recommandées

Le générateur de documentation doit accepter et traiter les balises valides conformément aux règles XML. Les balises suivantes fournissent des fonctionnalités couramment utilisées dans la documentation utilisateur. (Bien sûr, d'autres balises sont possibles.)

Tag	Section	Objectif
`<c>`	<c>	Définir du texte dans une police de type code
`<code>`	<code>	Définir une ou plusieurs lignes de code source ou de sortie du programme
`<example>`	<example>	Indiquer un exemple
`<exception>`	<exception>	Identifie les exceptions qu'une méthode peut lever

Tag	Section	Objectif
`<include>`	<include>	Contient du code XML à partir d'un fichier externe
`<list>`	<list>	Créer une liste ou une table
`<para>`	<para>	Autoriser l'ajout de la structure au texte
`<param>`	<param>	Décrire un paramètre pour une méthode ou un constructeur
`<paramref>`	<paramref>	Identifier qu'un mot est un nom de paramètre
`<permission>`	<permission>	Documenter l'accessibilité de sécurité d'un membre
`<remarks>`	<remarks>	Décrire des informations supplémentaires sur un type
`<returns>`	<returns>	Décrire la valeur de retour d'une méthode
`<see>`	<see>	Spécifier un lien
`<seealso>`	<seealso>	Générer une entrée Voir également
`<summary>`	<summary>	Décrire un type ou un membre d'un type
`<value>`	<value>	Décrire une propriété
`<typeparam>`		Décrire un paramètre de type générique
`<typeparamref>`		Identifier qu'un mot est un nom de paramètre de type

`<c>`

Cette balise fournit un mécanisme pour indiquer qu'un fragment de texte d'une description doit être défini dans une police spéciale telle que celle utilisée pour un bloc de code. Pour les lignes de code réel, utilisez `<code>` (<code>).

Syntaxe :

XML

```xml
<c>text</c>
```

Exemple :

C#

```
/// <summary>Class <c>Point</c> models a point in a two-dimensional
/// plane.</summary>
```

```
public class Point
{
    // ...
}
```

`<code>`

Cette balise est utilisée pour définir une ou plusieurs lignes de code source ou de sortie de programme dans une police spéciale. Pour les petits fragments de code dans la narration, utilisez `<c>` (<c>).

Syntaxe :

XML

```
<code>source code or program output</code>
```

Exemple :

C#

```
/// <summary>This method changes the point's location by
///    the given x- and y-offsets.
/// <example>For example:
/// <code>
///    Point p = new Point(3,5);
///    p.Translate(-1,3);
/// </code>
/// results in <c>p</c>'s having the value (2,8).
/// </example>
/// </summary>

public void Translate(int xor, int yor) {
    X += xor;
    Y += yor;
}
```

`<example>`

Cette balise permet d'obtenir un exemple de code dans un commentaire, afin de spécifier comment une méthode ou un autre membre de bibliothèque peut être utilisé. En règle générale, cela implique également l'utilisation de la balise `<code>` (<code>).

Syntaxe :

XML

```xml
<example>description</example>
```

Exemple :

`<code>` <code> Pour obtenir un exemple, consultez ().

`<exception>`

Cette balise permet de documenter les exceptions qu'une méthode peut lever.

Syntaxe :

XML

```xml
<exception cref="member">description</exception>
```

où

- `member` nom d'un membre. Le générateur de documentation vérifie que le membre donné existe et se traduit par `member` le nom d'élément canonique dans le fichier de documentation.
- `description` est une description des circonstances dans lesquelles l'exception est levée.

Exemple :

C#

```csharp
public class DataBaseOperations
{
    /// <exception cref="MasterFileFormatCorruptException"></exception>
    /// <exception cref="MasterFileLockedOpenException"></exception>
    public static void ReadRecord(int flag) {
        if (flag == 1)
            throw new MasterFileFormatCorruptException();
        else if (flag == 2)
            throw new MasterFileLockedOpenException();
        // ...
    }
}
```

`<include>`

Cette balise permet d'inclure des informations à partir d'un document XML qui est externe au fichier de code source. Le fichier externe doit être un document XML bien formé et une expression XPath est appliquée à ce document pour spécifier le code XML de ce document à inclure. La `<include>` balise est ensuite remplacée par le XML sélectionné du document externe.

Syntaxe :

XML

```xml
<include file="filename" path="xpath" />
```

où

- `filename` nom de fichier d'un fichier XML externe. Le nom du fichier est interprété par rapport au fichier contenant la balise Include.
- `xpath` expression XPath qui sélectionne des données XML dans le fichier XML externe.

Exemple :

Si le code source contenait une déclaration comme :

C#

```
/// <include file="docs.xml" path='extradoc/class[@name="IntList"]/*' />
public class IntList { ... }
```

et le fichier externe « docs.xml » comportait le contenu suivant :

XML

```xml
<?xml version="1.0"?>
<extradoc>
  <class name="IntList">
     <summary>
        Contains a list of integers.
     </summary>
  </class>
  <class name="StringList">
     <summary>
        Contains a list of integers.
     </summary>
  </class>
</extradoc>
```

la même documentation est ensuite générée comme si le code source contenait :

C#

```
/// <summary>
///    Contains a list of integers.
/// </summary>
public class IntList { ... }
```

`<list>`

Cette balise est utilisée pour créer une liste ou une table d'éléments. Elle peut contenir un `<listheader>` bloc pour définir la ligne d'en-tête d'une table ou d'une liste de définitions. (Lors de la définition d'une table, seule une entrée pour `term` dans l'en-tête doit être fournie.)

Chaque élément de la liste est spécifié avec un `<item>` bloc. Lors de la création d'une liste de définitions, `term` et `description` doivent être spécifiés. Toutefois, pour une table, une liste à puces ou une liste numérotée, seul `description` doit être spécifié.

Syntaxe :

XML

```
<list type="bullet" | "number" | "table">
   <listheader>
      <term>term</term>
      <description>*description*</description>
   </listheader>
   <item>
      <term>term</term>
      <description>*description*</description>
   </item>
    ...
   <item>
      <term>term</term>
      <description>description</description>
   </item>
</list>
```

où

- `term` terme à définir, dont la définition se trouve dans `description` .
- `description` est un élément dans une liste à puces ou une liste numérotée, ou la définition d'un `term` .

Exemple :

C#

```
public class MyClass
{
    /// <summary>Here is an example of a bulleted list:
    /// <list type="bullet">
    /// <item>
    /// <description>Item 1.</description>
    /// </item>
    /// <item>
    /// <description>Item 2.</description>
    /// </item>
    /// </list>
    /// </summary>
    public static void Main () {
        // ...
    }
}
```

`<para>`

Cette balise est destinée à être utilisée dans d'autres balises, telles que `<summary>` (<remarks>) ou `<returns>` (<returns>), et permet l'ajout de la structure au texte.

Syntaxe :

XML

```
<para>content</para>
```

où `content` est le texte du paragraphe.

Exemple :

C#

```
/// <summary>This is the entry point of the Point class testing program.
/// <para>This program tests each method and operator, and
/// is intended to be run after any non-trivial maintenance has
/// been performed on the Point class.</para></summary>
public static void Main() {
    // ...
}
```

`<param>`

Cette balise est utilisée pour décrire un paramètre d'une méthode, d'un constructeur ou d'un indexeur.

Syntaxe :

```
XML

<param name="name">description</param>
```

où

- `name` correspond au nom du paramètre.
- `description` Description du paramètre.

Exemple :

```
C#

/// <summary>This method changes the point's location to
///    the given coordinates.</summary>
/// <param name="xor">the new x-coordinate.</param>
/// <param name="yor">the new y-coordinate.</param>
public void Move(int xor, int yor) {
    X = xor;
    Y = yor;
}
```

`<paramref>`

Cette balise est utilisée pour indiquer qu'un mot est un paramètre. Le fichier de documentation peut être traité pour mettre en forme ce paramètre de manière distincte.

Syntaxe :

```
XML

<paramref name="name"/>
```

où `name` est le nom du paramètre.

Exemple :

```
C#
```

```
/// <summary>This constructor initializes the new Point to
///    (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
/// <param name="xor">the new Point's x-coordinate.</param>
/// <param name="yor">the new Point's y-coordinate.</param>

public Point(int xor, int yor) {
    X = xor;
    Y = yor;
}
```

`<permission>`

Cette balise permet de documenter l'accessibilité de sécurité d'un membre.

Syntaxe :

XML

```
<permission cref="member">description</permission>
```

où

- `member` nom d'un membre. Le générateur de documentation vérifie que l'élément de code donné existe et traduit le *membre* en nom d'élément canonique dans le fichier de documentation.
- `description` Description de l'accès au membre.

Exemple :

C#

```
/// <permission cref="System.Security.PermissionSet">Everyone can
/// access this method.</permission>

public static void Test() {
    // ...
}
```

`<remarks>`

Cette balise est utilisée pour spécifier des informations supplémentaires sur un type. (Utilisez `<summary>` (<summary>) pour décrire le type lui-même et les membres d'un type.)

Syntaxe :

XML

```
<remarks>description</remarks>
```

où `description` est le texte de la remarque.

Exemple :

C#

```
/// <summary>Class <c>Point</c> models a point in a
/// two-dimensional plane.</summary>
/// <remarks>Uses polar coordinates</remarks>
public class Point
{
    // ...
}
```

`<returns>`

Cette balise est utilisée pour décrire la valeur de retour d'une méthode.

Syntaxe :

XML

```
<returns>description</returns>
```

où `description` est une description de la valeur de retour.

Exemple :

C#

```
/// <summary>Report a point's location as a string.</summary>
/// <returns>A string representing a point's location, in the form (x,y),
///    without any leading, trailing, or embedded whitespace.</returns>
public override string ToString() {
    return "(" + X + "," + Y + ")";
}
```

`<see>`

Cette balise permet de spécifier un lien dans le texte. Utilisez `<seealso>` (<seealso>) pour indiquer le texte qui doit apparaître dans une section Voir aussi.

Syntaxe :

XML

```
<see cref="member"/>
```

où `member` est le nom d'un membre. Le générateur de documentation vérifie que l'élément de code donné existe et change de *membre* en nom d'élément dans le fichier de documentation généré.

Exemple :

C#

```
/// <summary>This method changes the point's location to
///    the given coordinates.</summary>
/// <see cref="Translate"/>
public void Move(int xor, int yor) {
    X = xor;
    Y = yor;
}

/// <summary>This method changes the point's location by
///    the given x- and y-offsets.
/// </summary>
/// <see cref="Move"/>
public void Translate(int xor, int yor) {
    X += xor;
    Y += yor;
}
```

`<seealso>`

Cette balise permet de générer une entrée pour la section Voir aussi. Utilisez `<see>` (<see>) pour spécifier un lien à partir de texte.

Syntaxe :

XML

```
<seealso cref="member"/>
```

où `member` est le nom d'un membre. Le générateur de documentation vérifie que l'élément de code donné existe et change de *membre* en nom d'élément dans le fichier de documentation généré.

Exemple :

```
C#

/// <summary>This method determines whether two Points have the same
///    location.</summary>
/// <seealso cref="operator=="/>
/// <seealso cref="operator!="/>
public override bool Equals(object o) {
    // ...
}
```

`<summary>`

Cette balise peut être utilisée pour décrire un type ou un membre d'un type. Utilisez `<remarks>` (<remarks>) pour décrire le type lui-même.

Syntaxe :

```
XML

<summary>description</summary>
```

où `description` est un résumé du type ou du membre.

Exemple :

```
C#

/// <summary>This constructor initializes the new Point to (0,0).</summary>
public Point() : this(0,0) {
}
```

`<value>`

Cette balise permet de décrire une propriété.

Syntaxe :

```
XML
```

```
<value>property description</value>
```

où `property description` est une description de la propriété.

Exemple :

C#

```
/// <value>Property <c>X</c> represents the point's x-coordinate.</value>
public int X
{
    get { return x; }
    set { x = value; }
}
```

`<typeparam>`

Cette balise est utilisée pour décrire un paramètre de type générique pour une classe, un struct, une interface, un délégué ou une méthode.

Syntaxe :

XML

```
<typeparam name="name">description</typeparam>
```

où `name` est le nom du paramètre de type et `description` est sa description.

Exemple :

C#

```
/// <summary>A generic list class.</summary>
/// <typeparam name="T">The type stored by the list.</typeparam>
public class MyList<T> {
    ...
}
```

`<typeparamref>`

Cette balise est utilisée pour indiquer qu'un mot est un paramètre de type. Le fichier de documentation peut être traité pour mettre en forme ce paramètre de type de manière distincte.

Syntaxe :

```XML
<typeparamref name="name"/>
```

où `name` est le nom du paramètre de type.

Exemple :

```C#
/// <summary>This method fetches data and returns a list of <typeparamref
name="T"/>.</summary>
/// <param name="query">query to execute</param>
public List<T> FetchData<T>(string query) {
    ...
}
```

Traitement du fichier de documentation

Le générateur de documentation génère une chaîne d'ID pour chaque élément du code source qui est balisé avec un commentaire de documentation. Cette chaîne d'ID identifie de façon unique un élément source. Une visionneuse de documentation peut utiliser une chaîne d'ID pour identifier l'élément de métadonnées/réflexion correspondant auquel la documentation s'applique.

Le fichier de documentation n'est pas une représentation hiérarchique du code source ; au lieu de cela, il s'agit d'une liste plate avec une chaîne d'ID générée pour chaque élément.

Format de chaîne d'ID

Le générateur de documentation respecte les règles suivantes lorsqu'il génère les chaînes d'ID :

- La chaîne ne contient aucun espace blanc.
- La première partie de la chaîne identifie le type de membre documenté, par le biais d'un caractère unique suivi d'un signe deux-points. Les genres de membres suivants sont définis :

Caractère	Description

Caractère	Description
E	Événement
F	Champ
M	Méthode (y compris les constructeurs, les destructeurs et les opérateurs)
N	Espace de noms
P	Propriété (y compris les indexeurs)
T	Type (par exemple Class, Delegate, enum, interface et struct)
!	Chaîne d'erreur ; le reste de la chaîne fournit des informations sur l'erreur. Par exemple, le générateur de documentation génère des informations d'erreur pour les liens qui ne peuvent pas être résolus.

- La deuxième partie de la chaîne est le nom qualifié complet de l'élément, en commençant à la racine de l'espace de noms. Le nom de l'élément, ses types englobants et l'espace de noms sont séparés par des points. Si le nom de l'élément lui-même comporte des points, ils sont remplacés par des `#(U+0023)` caractères. (Il est supposé qu'aucun élément ne possède ce caractère dans son nom.)

- Pour les méthodes et les propriétés avec arguments, la liste d'arguments suit, placée entre parenthèses. Pour ceux sans arguments, les parenthèses sont omises. Les arguments sont séparés par des virgules. L'encodage de chaque argument est identique à une signature CLI, comme suit :
  - Les arguments sont représentés par leur nom de documentation, qui est basé sur leur nom complet, modifié comme suit :
    - Les arguments qui représentent des types génériques ont un `` ` `` caractère ajouté (Tops) suivi du nombre de paramètres de type
    - Les arguments ayant `out` le `ref` modificateur ou ont un `@` nom de type suivant. Les arguments passés par valeur ou via n' `params` ont pas de notation spéciale.
    - Les arguments qui sont des tableaux sont représentés par `[lowerbound:size, ... , lowerbound:size]` le fait que le nombre de virgules est le rang moins un, et les limites inférieures et la taille de chaque dimension, si elles sont connues, sont représentées au format décimal. Si une limite inférieure ou une taille n'est pas spécifiée, elle est omise. Si la limite inférieure et la taille d'une dimension particulière sont omises, le `:` est également omis. Les tableaux en escalier sont représentés par un `[]` niveau par niveau.
    - Les arguments qui ont des types pointeur autres que void sont représentés à l'aide d'un `*` qui suit le nom de type. Un pointeur void est représenté à l'aide

d'un nom de type `System.Void` .

- Les arguments qui font référence aux paramètres de type générique définis sur les types sont encodés à l'aide du `` ` `` caractère (battement) suivi de l'index de base zéro du paramètre de type.
- Les arguments qui utilisent des paramètres de type générique définis dans les méthodes utilisent un double-cycle à `` `` `` la place du `` ` `` utilisé pour les types.
- Les arguments qui font référence aux types génériques construits sont encodés à l'aide du type générique, suivi de `{` , puis d'une liste séparée par des virgules d'arguments de type, suivie de `}` .

Exemples de chaînes d'ID

Les exemples suivants montrent chacun un fragment de code C#, ainsi que la chaîne d'ID produite à partir de chaque élément source, susceptible d'avoir un commentaire de documentation :

- Les types sont représentés à l'aide de leur nom qualifié complet et sont complétés par des informations génériques :

```
C#
```

```
enum Color { Red, Blue, Green }

namespace Acme
{
    interface IProcess {...}

    struct ValueType {...}

    class Widget: IProcess
    {
        public class NestedClass {...}
        public interface IMenuItem {...}
        public delegate void Del(int i);
        public enum Direction { North, South, East, West }
    }

    class MyList<T>
    {
        class Helper<U,V> {...}
    }
}

"T:Color"
"T:Acme.IProcess"
"T:Acme.ValueType"
"T:Acme.Widget"
```

```
"T:Acme.Widget.NestedClass"
"T:Acme.Widget.IMenuItem"
"T:Acme.Widget.Del"
"T:Acme.Widget.Direction"
"T:Acme.MyList`1"
"T:Acme.MyList`1.Helper`2"
```

- Les champs sont représentés par leur nom complet :

C#

```
namespace Acme
{
    struct ValueType
    {
        private int total;
    }

    class Widget: IProcess
    {
        public class NestedClass
        {
            private int value;
        }

        private string message;
        private static Color defaultColor;
        private const double PI = 3.14159;
        protected readonly double monthlyAverage;
        private long[] array1;
        private Widget[,] array2;
        private unsafe int *pCount;
        private unsafe float **ppValues;
    }
}

"F:Acme.ValueType.total"
"F:Acme.Widget.NestedClass.value"
"F:Acme.Widget.message"
"F:Acme.Widget.defaultColor"
"F:Acme.Widget.PI"
"F:Acme.Widget.monthlyAverage"
"F:Acme.Widget.array1"
"F:Acme.Widget.array2"
"F:Acme.Widget.pCount"
"F:Acme.Widget.ppValues"
```

- Des constructeurs.

C#

```
namespace Acme
{
    class Widget: IProcess
    {
        static Widget() {...}
        public Widget() {...}
        public Widget(string s) {...}
    }
}

"M:Acme.Widget.#cctor"
"M:Acme.Widget.#ctor"
"M:Acme.Widget.#ctor(System.String)"
```

- Destructeurs.

C#

```
namespace Acme
{
    class Widget: IProcess
    {
        ~Widget() {...}
    }
}

"M:Acme.Widget.Finalize"
```

- Leurs.

C#

```
namespace Acme
{
    struct ValueType
    {
        public void M(int i) {...}
    }

    class Widget: IProcess
    {
        public class NestedClass
        {
            public void M(int i) {...}
        }

        public static void M0() {...}
        public void M1(char c, out float f, ref ValueType v) {...}
        public void M2(short[] x1, int[,] x2, long[][] x3) {...}
        public void M3(long[][] x3, Widget[][,,] x4) {...}
        public unsafe void M4(char *pc, Color **pf) {...}
```

```
        public unsafe void M5(void *pv, double *[][,] pd) {...}
        public void M6(int i, params object[] args) {...}
    }

    class MyList<T>
    {
        public void Test(T t) { }
    }

    class UseList
    {
        public void Process(MyList<int> list) { }
        public MyList<T> GetValues<T>(T inputValue) { return null; }
    }
}

"M:Acme.ValueType.M(System.Int32)"
"M:Acme.Widget.NestedClass.M(System.Int32)"
"M:Acme.Widget.M0"
"M:Acme.Widget.M1(System.Char,System.Single@,Acme.ValueType@)"
"M:Acme.Widget.M2(System.Int16[],System.Int32[0:,0:],System.Int64[][])"
"M:Acme.Widget.M3(System.Int64[][],Acme.Widget[0:,0:,0:][])"
"M:Acme.Widget.M4(System.Char*,Color**)"
"M:Acme.Widget.M5(System.Void*,System.Double*[0:,0:][])"
"M:Acme.Widget.M6(System.Int32,System.Object[])"
"M:Acme.MyList`1.Test(`0)"
"M:Acme.UseList.Process(Acme.MyList{System.Int32})"
"M:Acme.UseList.GetValues``(``0)"
```

- Propriétés et indexeurs.

C#

```
namespace Acme
{
    class Widget: IProcess
    {
        public int Width { get {...} set {...} }
        public int this[int i] { get {...} set {...} }
        public int this[string s, int i] { get {...} set {...} }
    }
}

"P:Acme.Widget.Width"
"P:Acme.Widget.Item(System.Int32)"
"P:Acme.Widget.Item(System.String,System.Int32)"
```

- Événements.

C#

```
namespace Acme
{
    class Widget: IProcess
    {
        public event Del AnEvent;
    }
}

"E:Acme.Widget.AnEvent"
```

- Les opérateurs unaires.

```
C#

namespace Acme
{
    class Widget: IProcess
    {
        public static Widget operator+(Widget x) {...}
    }
}

"M:Acme.Widget.op_UnaryPlus(Acme.Widget)"
```

L'ensemble complet des noms de fonction d'opérateur unaire utilisés est le suivant : `op_UnaryPlus` , `op_UnaryNegation` , `op_LogicalNot` , `op_OnesComplement` , `op_Increment` , `op_Decrement` , `op_True` et `op_False` .

- Opérateurs binaires.

```
C#

namespace Acme
{
    class Widget: IProcess
    {
        public static Widget operator+(Widget x1, Widget x2) {...}
    }
}

"M:Acme.Widget.op_Addition(Acme.Widget,Acme.Widget)"
```

Le jeu complet de noms de fonctions d'opérateur binaires utilisé est le suivant : `op_Addition` , `op_Subtraction` ,,,, `op_Multiply` `op_Division` `op_Modulus` `op_BitwiseAnd` , `op_BitwiseOr` , `op_ExclusiveOr` , `op_LeftShift` , `op_RightShift` , `op_Equality` , `op_Inequality` , `op_LessThan` , `op_LessThanOrEqual` , `op_GreaterThan` et `op_GreaterThanOrEqual` .

- Les opérateurs de conversion ont un « `~` » de fin suivi du type de retour.

```
C#
```

```
namespace Acme
{
    class Widget: IProcess
    {
        public static explicit operator int(Widget x) {...}
        public static implicit operator long(Widget x) {...}
    }
}

"M:Acme.Widget.op_Explicit(Acme.Widget)~System.Int32"
"M:Acme.Widget.op_Implicit(Acme.Widget)~System.Int64"
```

Exemple

Code source C#

L'exemple suivant montre le code source d'une `Point` classe :

```
C#
```

```
namespace Graphics
{

/// <summary>Class <c>Point</c> models a point in a two-dimensional plane.
/// </summary>
public class Point
{

    /// <summary>Instance variable <c>x</c> represents the point's
    ///    x-coordinate.</summary>
    private int x;

    /// <summary>Instance variable <c>y</c> represents the point's
    ///    y-coordinate.</summary>
    private int y;

    /// <value>Property <c>X</c> represents the point's x-coordinate.
</value>
    public int X
    {
        get { return x; }
        set { x = value; }
    }

    /// <value>Property <c>Y</c> represents the point's y-coordinate.
```

```
/// </value>
    public int Y
    {
        get { return y; }
        set { y = value; }
    }

    /// <summary>This constructor initializes the new Point to
    ///    (0,0).</summary>
    public Point() : this(0,0) {}

    /// <summary>This constructor initializes the new Point to
    ///    (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
    /// <param><c>xor</c> is the new Point's x-coordinate.</param>
    /// <param><c>yor</c> is the new Point's y-coordinate.</param>
    public Point(int xor, int yor) {
        X = xor;
        Y = yor;
    }

    /// <summary>This method changes the point's location to
    ///    the given coordinates.</summary>
    /// <param><c>xor</c> is the new x-coordinate.</param>
    /// <param><c>yor</c> is the new y-coordinate.</param>
    /// <see cref="Translate"/>
    public void Move(int xor, int yor) {
        X = xor;
        Y = yor;
    }

    /// <summary>This method changes the point's location by
    ///    the given x- and y-offsets.
    /// <example>For example:
    /// <code>
    ///    Point p = new Point(3,5);
    ///    p.Translate(-1,3);
    /// </code>
    /// results in <c>p</c>'s having the value (2,8).
    /// </example>
    /// </summary>
    /// <param><c>xor</c> is the relative x-offset.</param>
    /// <param><c>yor</c> is the relative y-offset.</param>
    /// <see cref="Move"/>
    public void Translate(int xor, int yor) {
        X += xor;
        Y += yor;
    }

    /// <summary>This method determines whether two Points have the same
    ///    location.</summary>
    /// <param><c>o</c> is the object to be compared to the current object.
    /// </param>
    /// <returns>True if the Points have the same location and they have
    ///    the exact same type; otherwise, false.</returns>
    /// <seealso cref="operator=="/>
```

```
    /// <seealso cref="operator!="/>
    public override bool Equals(object o) {
        if (o == null) {
            return false;
        }

        if (this == o) {
            return true;
        }

        if (GetType() == o.GetType()) {
            Point p = (Point)o;
            return (X == p.X) && (Y == p.Y);
        }
        return false;
    }

    /// <summary>Report a point's location as a string.</summary>
    /// <returns>A string representing a point's location, in the form
(x,y),
    ///    without any leading, training, or embedded whitespace.</returns>
    public override string ToString() {
        return "(" + X + "," + Y + ")";
    }

    /// <summary>This operator determines whether two Points have the same
    ///    location.</summary>
    /// <param><c>p1</c> is the first Point to be compared.</param>
    /// <param><c>p2</c> is the second Point to be compared.</param>
    /// <returns>True if the Points have the same location and they have
    ///    the exact same type; otherwise, false.</returns>
    /// <seealso cref="Equals"/>
    /// <seealso cref="operator!="/>
    public static bool operator==(Point p1, Point p2) {
        if ((object)p1 == null || (object)p2 == null) {
            return false;
        }

        if (p1.GetType() == p2.GetType()) {
            return (p1.X == p2.X) && (p1.Y == p2.Y);
        }

        return false;
    }

    /// <summary>This operator determines whether two Points have the same
    ///    location.</summary>
    /// <param><c>p1</c> is the first Point to be compared.</param>
    /// <param><c>p2</c> is the second Point to be compared.</param>
    /// <returns>True if the Points do not have the same location and the
    ///    exact same type; otherwise, false.</returns>
    /// <seealso cref="Equals"/>
    /// <seealso cref="operator=="/>
    public static bool operator!=(Point p1, Point p2) {
        return !(p1 == p2);
```

```
    }

    /// <summary>This is the entry point of the Point class testing
    /// program.
    /// <para>This program tests each method and operator, and
    /// is intended to be run after any non-trivial maintenance has
    /// been performed on the Point class.</para></summary>
    public static void Main() {
        // class test code goes here
    }
}
}
```

XML résultant

Voici la sortie produite par un générateur de documentation en fonction du code source de la classe `Point` , comme indiqué ci-dessus :

XML

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>Point</name>
    </assembly>
    <members>
        <member name="T:Graphics.Point">
            <summary>Class <c>Point</c> models a point in a two-dimensional
            plane.
            </summary>
        </member>

        <member name="F:Graphics.Point.x">
            <summary>Instance variable <c>x</c> represents the point's
            x-coordinate.</summary>
        </member>

        <member name="F:Graphics.Point.y">
            <summary>Instance variable <c>y</c> represents the point's
            y-coordinate.</summary>
        </member>

        <member name="M:Graphics.Point.#ctor">
            <summary>This constructor initializes the new Point to
        (0,0).</summary>
        </member>

        <member name="M:Graphics.Point.#ctor(System.Int32,System.Int32)">
            <summary>This constructor initializes the new Point to
            (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
            <param><c>xor</c> is the new Point's x-coordinate.</param>
```

```
            <param><c>yor</c> is the new Point's y-coordinate.</param>
        </member>

        <member name="M:Graphics.Point.Move(System.Int32,System.Int32)">
            <summary>This method changes the point's location to
            the given coordinates.</summary>
            <param><c>xor</c> is the new x-coordinate.</param>
            <param><c>yor</c> is the new y-coordinate.</param>
            <see
cref="M:Graphics.Point.Translate(System.Int32,System.Int32)"/>
        </member>

        <member
            name="M:Graphics.Point.Translate(System.Int32,System.Int32)">
            <summary>This method changes the point's location by
            the given x- and y-offsets.
            <example>For example:
            <code>
            Point p = new Point(3,5);
            p.Translate(-1,3);
            </code>
            results in <c>p</c>'s having the value (2,8).
            </example>
            </summary>
            <param><c>xor</c> is the relative x-offset.</param>
            <param><c>yor</c> is the relative y-offset.</param>
            <see cref="M:Graphics.Point.Move(System.Int32,System.Int32)"/>
        </member>

        <member name="M:Graphics.Point.Equals(System.Object)">
            <summary>This method determines whether two Points have the same
            location.</summary>
            <param><c>o</c> is the object to be compared to the current
            object.
            </param>
            <returns>True if the Points have the same location and they have
            the exact same type; otherwise, false.</returns>
            <seealso
      cref="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)"/>
            <seealso
      cref="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)"/>
        </member>

        <member name="M:Graphics.Point.ToString">
            <summary>Report a point's location as a string.</summary>
            <returns>A string representing a point's location, in the form
            (x,y),
            without any leading, training, or embedded whitespace.</returns>
        </member>

        <member
       name="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)">
            <summary>This operator determines whether two Points have the
            same
            location.</summary>
```

```
            <param><c>p1</c> is the first Point to be compared.</param>
            <param><c>p2</c> is the second Point to be compared.</param>
            <returns>True if the Points have the same location and they have
            the exact same type; otherwise, false.</returns>
            <seealso cref="M:Graphics.Point.Equals(System.Object)"/>
            <seealso
      cref="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)"/>
        </member>

        <member
       name="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)">
            <summary>This operator determines whether two Points have the
            same
            location.</summary>
            <param><c>p1</c> is the first Point to be compared.</param>
            <param><c>p2</c> is the second Point to be compared.</param>
            <returns>True if the Points do not have the same location and
            the
            exact same type; otherwise, false.</returns>
            <seealso cref="M:Graphics.Point.Equals(System.Object)"/>
            <seealso
       cref="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)"/>
        </member>

        <member name="M:Graphics.Point.Main">
            <summary>This is the entry point of the Point class testing
            program.
            <para>This program tests each method and operator, and
            is intended to be run after any non-trivial maintenance has
            been performed on the Point class.</para></summary>
        </member>

        <member name="P:Graphics.Point.X">
            <value>Property <c>X</c> represents the point's
            x-coordinate.</value>
        </member>

        <member name="P:Graphics.Point.Y">
            <value>Property <c>Y</c> represents the point's
            y-coordinate.</value>
        </member>
    </members>
</doc>
```

Annex E Bibliography

Article • 01/13/2022

> **This annex is informative.**
>
> ANSI X3.274-1996, *Programming Language REXX*. (This document is useful in understanding floating-point decimal arithmetic rules.)
>
> ISO/IEC 9075-1, *Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework)*
>
> ISO/IEC 9899, *Programming languages — C*.
>
> ISO/IEC 14882 *Programming languages — C++*
>
> ISO 80000-1, Quantities and units — Part 1: General. (This document defines "banker's rounding.")
>
> **End of informative text.**

Types référence Nullable en C

Article • 13/05/2021

L'objectif de cette fonctionnalité est de :

- Permet aux développeurs d'exprimer si une variable, un paramètre ou un résultat d'un type référence est destiné à être null ou non.
- Fournissez des avertissements lorsque ces variables, paramètres et résultats ne sont pas utilisés conformément à cet objectif.

Expression de l'intention

La langue contient déjà la `T?` syntaxe pour les types valeur. Il est facile d'étendre cette syntaxe aux types référence.

Il est supposé que l'intention d'un type de référence sans ornement `T` est qu'il ne soit pas null.

Vérification des références Nullable

Une analyse de fluide effectue le suivi des variables de référence Nullable. Lorsque l'analyse estime qu'elles ne sont pas nulles (par exemple, après une vérification ou une affectation), leur valeur est considérée comme une référence non null.

Une référence Nullable peut également être traitée explicitement comme non null avec l'opérateur postfix `x!` (l'opérateur « damnit »), lorsque l'analyse du workflow ne peut pas établir une situation non null que le développeur connaît.

Dans le cas contraire, un avertissement est fourni si une référence Nullable est déréférencée ou est convertie en un type non null.

Un avertissement est fourni lors de la conversion de `S[]` vers `T?[]` et de à `S?[]` `T[]` .

Un avertissement est fourni lors de la conversion de `C<S>` en `C<T?>` sauf si le paramètre de type est covariant (`out`), et lors de la conversion de `C<S?>` en `C<T>` sauf si le paramètre de type est contravariant (`in`).

Un avertissement est fourni `C<T?>` si le paramètre de type a des contraintes non null.

Vérification des références non null

Un avertissement est fourni si un littéral null est assigné à une variable non null ou passé comme paramètre non null.

Un avertissement est également fourni si un constructeur n'initialise pas explicitement les champs de référence non null.

Nous ne pouvons pas suivre correctement l'initialisation de tous les éléments d'un tableau de références non null. Toutefois, nous pouvons émettre un avertissement si aucun élément d'un tableau nouvellement créé n'est assigné avant que le tableau soit lu ou passé. Cela peut gérer le cas courant sans trop bruyant.

Nous devons décider si `default(T)` génère un avertissement ou est simplement traité comme étant du type `T?` .

Représentation des métadonnées

Les ornements de possibilité de valeur NULL doivent être représentés dans les métadonnées en tant qu'attributs. Cela signifie que les compilateurs de niveau inférieur les ignorent.

Nous devons décider si seules les annotations Nullable sont incluses, ou il existe également une indication précisant si la valeur non null était « on » dans l'assembly.

Génériques

Si un paramètre de type `T` a des contraintes n'acceptant pas les valeurs NULL, il est traité comme n'acceptant pas les valeurs NULL dans sa portée.

Si un paramètre de type est sans contrainte ou n'a que des contraintes Nullable, la situation est un peu plus complexe : cela signifie que l'argument de type correspondant peut avoir la *valeur* null ou n'accepte pas les valeurs NULL. La seule chose à faire dans cette situation est de traiter le paramètre de type comme Nullable et non Nullable, en donnant *des* avertissements quand l'un ou l'autre est violé.

Il est important de considérer si les contraintes de référence Nullable explicites doivent être autorisées. Notez, toutefois, que nous ne pouvons pas éviter que les types de référence Nullable soient *implicitement* des contraintes dans certains cas (contraintes héritées).

La `class` contrainte n'a pas la valeur null. Nous pouvons déterminer si `class?` doit être une contrainte Nullable valide indiquant « type de référence Nullable ».

Inférence de type

Dans l'inférence de type, si un type contributeur est un type référence Nullable, le type résultant doit être Nullable. En d'autres termes, la valeur null est propagée.

Nous devons déterminer si le `null` littéral en tant qu'expression participante doit contribuer à la valeur null. Ce n'est pas aujourd'hui le cas : pour les types de valeur, il génère une erreur, tandis que pour les types de référence, la valeur null est convertie avec succès en type brut.

C#

```
string? n = "world";
var x = b ? "Hello" : n; // string?
var y = b ? "Hello" : null; // string? or error
var z = b ? 7 : null; // Error today, could be int?
```

Aide à la protection null

En tant que fonctionnalité, les types de référence Nullable permettent aux développeurs d'exprimer leur intention et de fournir des avertissements par le biais de l'analyse du fluide si cette intention est contradictoire. Il existe une question courante quant à la nécessité ou non de comprotections null.

Exemple de protection null

C#

```
public void DoWork(Worker worker)
{
    // Guard against worker being null
    if (worker is null)
    {
        throw new ArgumentNullException(nameof(worker));
    }

    // Otherwise use worker argument
}
```

Dans l'exemple précédent, la `DoWork` fonction accepte un `Worker` et protège potentiellement `null` . Si l' `worker` argument est `null` , la `DoWork` fonction le sera `throw` . Avec les types de référence Nullable, le code de l'exemple précédent fait que le `Worker` paramètre ne l'est *pas* `null` . Si la `DoWork` fonction était une API publique, telle qu'un

package NuGet ou une bibliothèque partagée, vous devez conserver les protections null en place. En tant qu'API publique, la seule garantie qu'un appelant n' `null` est pas en mesure de se protéger.

Intention expresse

Une utilisation plus attrayante de l'exemple précédent consiste à exprimer que le `Worker` paramètre peut être `null` , rendant ainsi le garde-null plus approprié. Si vous supprimez le garde null dans l'exemple suivant, le compilateur vous avertit que vous pouvez déréférencer la valeur null. Quoi qu'il en soit, les deux protections NULL sont toujours valides.

C#

```
public void DoWork(Worker? worker)
{
    // Guard against worker being null
    if (worker is null)
    {
        throw new ArgumentNullException(nameof(worker));
    }

    // Otherwise use worker argument
}
```

Pour les API non publiques, telles que le code source entièrement sous contrôle par un développeur ou une équipe de développement, les types de référence Nullable peuvent permettre la suppression sécurisée des protections NULL lorsque les développeurs peuvent garantir qu'il n'est pas nécessaire. La fonctionnalité peut être utile avec les avertissements, mais elle ne peut pas garantir qu'au moment de l'exécution du code d'exécution peut entraîner un `NullReferenceException` .

Changements cassants

Les avertissements non NULL sont une modification avec rupture évidente sur le code existant et doivent être accompagnés d'un mécanisme d'abonnement.

Moins évidemment, les avertissements des types Nullable (comme décrit ci-dessus) sont une modification avec rupture sur le code existant dans certains scénarios où la possibilité de valeur null est implicite :

- Les paramètres de type sans contrainte sont traités comme des valeurs NULL implicites, donc leur assignation à `object` ou l'accès à, par exemple, `ToString`

générent des avertissements.
- Si l'inférence de type déduit la valeur null des `null` expressions, le code existant génère parfois des valeurs NULL plutôt que des types non Nullable, ce qui peut entraîner de nouveaux avertissements.

Les avertissements Nullable doivent également être facultatifs

Enfin, l'ajout d'annotations à une API existante sera une modification avec rupture pour les utilisateurs qui ont opté pour les avertissements lorsqu'ils effectuent la mise à niveau de la bibliothèque. Cela vous permet également d'accepter ou de refuser l'inscription. « Je veux les correctifs de bogues, mais je ne suis pas prêt à gérer leurs nouvelles annotations »

En résumé, vous devez être en mesure d'accepter/de refuser :

- Avertissements Nullable
- Avertissements non null
- Avertissements d'annotations dans d'autres fichiers

La granularité de l'abonnement suggère un modèle de type analyseur, où les quantités de code peuvent s'inscrire et sortir avec des pragmas et des niveaux de gravité peuvent être choisis par l'utilisateur. En outre, les options par bibliothèque (« ignorer les annotations de JSON.NET jusqu'à ce que je suis prêt à gérer le passage ») peuvent être exprimables dans le code en tant qu'attributs.

La conception de l'expérience d'abonnement/transition est cruciale pour la réussite et l'utilité de cette fonctionnalité. Nous devons nous assurer que :

- Les utilisateurs peuvent adopter une vérification de la possibilité de valeur NULL comme ils le souhaitent
- Les auteurs de bibliothèques peuvent ajouter des annotations de possibilité de valeur null sans craindre de perturber les clients
- En dépit de ceux-ci, il n'y a pas de sens de « cauchemar de la configuration »

Ajustements

Nous pourrions envisager de ne pas utiliser les `?` annotations sur les variables locales, mais simplement d'observer s'ils sont utilisés conformément à ce qui leur est assigné. Je ne préfère pas cela. Je pense que nous devrions permettre aux utilisateurs d'exprimer leur intention de manière uniforme.

Nous pourrions envisager un raccourci `T! x` sur les paramètres, qui génère automatiquement une vérification de la valeur null au moment de l'exécution.

Certains modèles sur des types génériques, tels que `FirstOrDefault` ou `TryGet` , ont un comportement légèrement étrange avec les arguments de type non Nullable, car ils génèrent explicitement des valeurs par défaut dans certaines situations. Nous pourrions essayer de nuancer le système de type pour s'adapter à ces améliorations. Par exemple, nous pourrions autoriser `?` sur les paramètres de type sans contrainte, même si l'argument de type peut déjà être Nullable. Je n'ai pas de doute qu'il en mérite, et cela a pour conséquence l'étrange liée à l'interaction avec les types *valeur* Nullable.

Types valeur Nullable

Nous pourrions envisager d'adopter également une partie de la sémantique ci-dessus pour les types valeur Nullable.

Nous avons déjà mentionné l'inférence de type, où nous pourrions déduire `int?` `(7, null)` , au lieu de donner simplement une erreur.

Une autre possibilité consiste à appliquer l'analyse du workflow à des types valeur Nullable. Lorsqu'ils sont considérés comme non null, nous pourrions réellement autoriser l'utilisation de comme type non Nullable de certaines façons (par exemple, l'accès aux membres). Nous devons simplement veiller à ce que les choses que vous puissiez *déjà* faire sur un type de valeur Nullable soient préférées, pour des raisons de compatibilité de retour.

Critères spéciaux récursifs

Article • 13/05/2021

Résumé

Les extensions de critères spéciaux pour C# permettent de nombreux avantages des types de données algébriques et des critères spéciaux des langages fonctionnels, mais d'une façon qui s'intègre parfaitement au langage sous-jacent. Les éléments de cette approche sont inspirés par les fonctionnalités associées dans les langages de programmation F # et Scala.

Conception détaillée

Expression is

L' `is` opérateur est étendu pour tester une expression par rapport à un *modèle*.

antlr

```
relational_expression
    : is_pattern_expression
    ;
is_pattern_expression
    : relational_expression 'is' pattern
    ;
```

Cette forme de *relational_expression* s'ajoute aux formulaires existants dans la spécification C#. Il s'agit d'une erreur de compilation si le *relational_expression* à gauche du `is` jeton ne désigne pas une valeur ou n'a pas de type.

Chaque *identificateur* du modèle introduit une nouvelle variable locale qui est *définitivement assignée* une fois que l' `is` opérateur a la `true` valeur (c'est-à-dire *assignée de manière définitive quand la valeur est true*).

> Remarque : il existe techniquement une ambiguïté entre le *type* d'un `is-expression` et *constant_pattern*, qui peut être une analyse valide d'un identificateur qualifié. Nous essayons de la lier en tant que type pour la compatibilité avec les versions précédentes du langage. en cas d'échec, nous allons le résoudre, car nous faisons une expression dans d'autres contextes, à la première chose trouvée (qui doit être

> une constante ou un type). Cette ambiguïté est uniquement présente sur le côté droit d'une `is` expression.

Modèles

Les modèles sont utilisés dans l'opérateur *is_pattern* , dans une *switch_Statement* et dans une *switch_expression* pour exprimer la forme des données par rapport à laquelle les données entrantes (que nous appelons la valeur d'entrée) doivent être comparées. Les modèles peuvent être récursifs afin que certaines parties des données puissent être comparées à des sous-modèles.

antlr

```
pattern
    : declaration_pattern
    | constant_pattern
    | var_pattern
    | positional_pattern
    | property_pattern
    | discard_pattern
    ;
declaration_pattern
    : type simple_designation
    ;
constant_pattern
    : constant_expression
    ;
var_pattern
    : 'var' designation
    ;
positional_pattern
    : type? '(' subpatterns? ')' property_subpattern? simple_designation?
    ;
subpatterns
    : subpattern
    | subpattern ',' subpatterns
    ;
subpattern
    : pattern
    | identifier ':' pattern
    ;
property_subpattern
    : '{' '}'
    | '{' subpatterns ','? '}'
    ;
property_pattern
    : type? property_subpattern simple_designation?
    ;
simple_designation
    : single_variable_designation
    | discard_designation
```

```
    ;
discard_pattern
    : '_'
    ;
```

Modèle de déclaration

antlr

```
declaration_pattern
    : type simple_designation
    ;
```

Le *declaration_pattern* teste qu'une expression est d'un type donné et la convertit en ce type si le test a la valeur. Cela peut introduire une variable locale du type donné nommé par l'identificateur donné, si la désignation est un *single_variable_designation*. Cette variable locale est *assignée définitivement* lorsque le résultat de l'opération de correspondance de modèle est `true` .

La sémantique d'exécution de cette expression est qu'elle teste le type de Runtime de l'opérande de *relational_expression* gauche par rapport au *type* dans le modèle. S'il s'agit d'un type de Runtime (ou d'un sous-type) et non `null` , le résultat de `is operator` est `true` .

Certaines combinaisons de type statique du côté gauche et du type donné sont considérées comme incompatibles et entraînent une erreur au moment de la compilation. Une valeur de type statique `E` est dite compatible avec les *modèles* avec un type `T` s'il existe une conversion d'identité, une conversion de référence implicite, une conversion boxing, une conversion de référence explicite ou une conversion unboxing de `E` en `T` , ou si l'un de ces types est un type ouvert. Il s'agit d'une erreur de compilation si une entrée de type `E` n'est pas *compatible* avec le modèle avec le *type* dans un modèle de type auquel elle est associée.

Le modèle de type est utile pour effectuer des tests de type au moment de l'exécution de types référence et remplace l'idiome

C#

```
var v = expr as Type;
if (v != null) { // code using v
```

Avec un peu plus concis

C#

```
if (expr is Type v) { // code using v
```

Il s'agit d'une erreur si le *type* est un type valeur Nullable.

Le modèle de type peut être utilisé pour tester des valeurs de types Nullable : une valeur de type `Nullable<T>` (ou un boxed `T`) correspond à un modèle de type `T2 id` si la valeur n'est pas null et que le type de `T2` est `T` , ou un type de base ou une interface de `T` . Par exemple, dans le fragment de code

C#

```
int? x = 3;
if (x is int v) { // code using v
```

La condition de l' `if` instruction est `true` au moment de l'exécution et la variable `v` contient la valeur `3` du type `int` dans le bloc. Après le bloc, la variable `v` est dans la portée, mais elle n'est pas définitivement assignée.

Modèle de constante

antlr

```
constant_pattern
    : constant_expression
    ;
```

Un modèle de constante teste la valeur d'une expression par rapport à une valeur constante. La constante peut être n'importe quelle expression constante, telle qu'un littéral, le nom d'une variable déclarée `const` ou une constante d'énumération. Lorsque la valeur d'entrée n'est pas un type ouvert, l'expression constante est implicitement convertie dans le type de l'expression correspondante ; Si le type de la valeur d'entrée n'est pas *compatible* avec le type de l'expression constante, l'opération de correspondance de modèle est une erreur.

Le modèle *c* est considéré comme correspondant à la valeur d'entrée convertie *e* si `object.Equals(c, e)` retourne `true` .

Nous pensons `e is null` qu'il s'agit de la méthode la plus courante pour tester `null` dans du code récemment écrit, car il ne peut pas appeler un défini par l'utilisateur `operator==` .

Modèle var

```antlr
var_pattern
    : 'var' designation
    ;
designation
    : simple_designation
    | tuple_designation
    ;
simple_designation
    : single_variable_designation
    | discard_designation
    ;
single_variable_designation
    : identifier
    ;
discard_designation
    : _
    ;
tuple_designation
    : '(' designations? ')'
    ;
designations
    : designation
    | designations ',' designation
    ;
```

Si la *désignation* est un *simple_designation*, une expression *e* correspond au modèle. En d'autres termes, une correspondance avec un *modèle var* est toujours réussie avec un *simple_designation*. Si le *simple_designation* est un *single_variable_designation*, la valeur de *e* est liée à une variable locale nouvellement introduite. Le type de la variable locale est le type statique de *e*.

Si la *désignation* est un *tuple_designation*, le modèle est équivalent à un *positional_pattern* de la `(var` *désignation* de formulaire,... `)` où les *désignations* sont celles qui se trouvent dans le *tuple_designation*. Par exemple, le modèle `var (x, (y, z))` est équivalent à `(var x, (var y, var z))` .

Il s'agit d'une erreur si le nom est `var` lié à un type.

Supprimer le modèle

```antlr
discard_pattern
    : '_'
```

```
    ;
```

Une expression *e* correspond toujours au modèle `_` . En d'autres termes, chaque expression correspond au modèle d'abandon.

Un modèle d'abandon ne peut pas être utilisé comme modèle d'un *is_pattern_expression*.

Modèle positionnel

Un modèle positionnel vérifie que la valeur d'entrée n'est pas `null` , appelle une `Deconstruct` méthode appropriée et effectue d'autres critères spéciaux sur les valeurs résultantes. Il prend également en charge une syntaxe de modèle de type Tuple (sans le type fourni) lorsque le type de la valeur d'entrée est le même que le type qui contient `Deconstruct` , ou si le type de la valeur d'entrée est un type de tuple, ou si le type de la valeur d'entrée est `object` ou `ITuple` et que le type d'exécution de l'expression implémente `ITuple` .

antlr

```
positional_pattern
    : type? '(' subpatterns? ')' property_subpattern? simple_designation?
    ;
subpatterns
    : subpattern
    | subpattern ',' subpatterns
    ;
subpattern
    : pattern
    | identifier ':' pattern
    ;
```

Si le *type* est omis, nous le prenons comme type statique de la valeur d'entrée.

Pour une correspondance entre une valeur d'entrée et le *type* de modèle `(` *subpattern_list* `)` , une méthode est sélectionnée en recherchant dans *type* les déclarations accessibles de `Deconstruct` et en en sélectionnant une parmi celles-ci en utilisant les mêmes règles que pour la déclaration de déconstruction.

Il s'agit d'une erreur si un *positional_pattern* omet le type, a un seul sous- *modèle* sans *identificateur*, n'a pas de *property_subpattern* et n'a pas de *simple_designation*. Ce ambiguïté entre une *constant_pattern* entre parenthèses et un *positional_pattern*.

Afin d'extraire les valeurs pour les faire correspondre aux modèles de la liste,

- Si le *type* a été omis et que le type de la valeur d'entrée est un type tuple, le nombre de sous-modèles doit être le même que la cardinalité du tuple. Chaque élément de tuple est mis en correspondance avec le sous- *modèle* correspondant, et la correspondance est établie si tous ces éléments sont correctement exécutés. Si un sous- *modèle* a un *identificateur*, cela doit nommer un élément de tuple à la position correspondante dans le type de Tuple.
- Sinon, si un approprié `Deconstruct` existe en tant que membre de *type*, il s'agit d'une erreur de compilation si le type de la valeur d'entrée n'est pas compatible avec le *modèle* . Au moment de l'exécution, la valeur d'entrée est testée par rapport au *type*. En cas d'échec, la correspondance du modèle positionnel échoue. Si elle réussit, la valeur d'entrée est convertie vers ce type et `Deconstruct` est appelée avec les nouvelles variables générées par le compilateur pour recevoir les `out` paramètres. Chaque valeur reçue est mise en correspondance avec le sous-*modèle* correspondant, et la correspondance est établie si toutes ces valeurs sont exécutées correctement. Si un sous- *modèle* a un *identificateur*, alors il doit nommer un paramètre à la position correspondante de `Deconstruct` .
- Sinon, si le *type* a été omis, si la valeur d'entrée est de type `object` ou `ITuple` ou un type pouvant être converti en `ITuple` par une conversion de référence implicite, et si aucun *identificateur* n'apparaît parmi les sous-modèles, nous effectuons une correspondance avec `ITuple` .
- Dans le cas contraire, le modèle est une erreur au moment de la compilation.

L'ordre dans lequel les sous-modèles sont mis en correspondance au moment de l'exécution n'est pas spécifié, et une erreur de correspondance peut ne pas tenter de faire correspondre tous les sous-modèles.

Exemple

Cet exemple utilise un grand nombre des fonctionnalités décrites dans cette spécification.

```
c#
```

```
    var newState = (GetState(), action, hasKey) switch {
        (DoorState.Closed, Action.Open, _) => DoorState.Opened,
        (DoorState.Opened, Action.Close, _) => DoorState.Closed,
        (DoorState.Closed, Action.Lock, true) => DoorState.Locked,
        (DoorState.Locked, Action.Unlock, true) => DoorState.Closed,
        (var state, _, _) => state };
```

Modèle de propriété

Un modèle de propriété vérifie que la valeur d'entrée n'est pas `null` et fait correspondre de manière récursive les valeurs extraites par l'utilisation de propriétés ou de champs accessibles.

antlr

```
property_pattern
    : type? property_subpattern simple_designation?
    ;
property_subpattern
    : '{' '}'
    | '{' subpatterns ','? '}'
    ;
```

Il s'agit d'une erreur si un sous- *modèle* d'un *property_pattern* ne contient pas d' *identificateur* (il doit être de la deuxième forme, qui a un *identificateur*). Une virgule de fin après le dernier sous-modèle est facultative.

Notez qu'un modèle de vérification null est issu d'un modèle de propriété trivial. Pour vérifier si la chaîne `s` n'est pas null, vous pouvez écrire l'une des formes suivantes :

C#

```
if (s is object o) ... // o is of type object
if (s is string x) ... // x is of type string
if (s is {} x) ... // x is of type string
if (s is {}) ...
```

En raison d'une correspondance entre une expression *e* et le *type* de modèle `{` *property_pattern_list* `}` , il s'agit d'une erreur de compilation si l'expression *e* n'est pas *compatible* avec le type *T* désigné par le type. Si le type est absent, nous le prenons comme type statique de *e*. Si l' *identificateur* est présent, il déclare une variable de modèle *de type type*. Chacun des identificateurs figurant sur le côté gauche de son *property_pattern_list* doit désigner une propriété ou un champ accessible en lecture accessible de *T*. Si le *simple_designation* du *property_pattern* est présent, il définit une variable de modèle de type *T*.

Au moment de l'exécution, l'expression est testée par rapport à *T*. En cas d'échec, la correspondance du modèle de propriété échoue et le résultat est `false` . Si elle est réussie, chaque *property_subpattern* champ ou propriété est lu et sa valeur correspond à son modèle correspondant. Le résultat de la correspondance entière est `false` uniquement si le résultat de l'un de ces résultats est `false` . L'ordre dans lequel les sous-modèles sont mis en correspondance n'est pas spécifié, et une correspondance qui a échoué peut ne pas correspondre à tous les sous-modèles au moment de l'exécution.

Si la correspondance est établie et que la *simple_designation* du *property_pattern* est un *single_variable_designation*, elle définit une variable de type *T* à laquelle est assignée la valeur correspondante.

> Remarque : le modèle de propriété peut être utilisé pour effectuer une correspondance de modèle avec des types anonymes.

Exemple

C#

```
if (o is string { Length: 5 } s)
```

Expression de switch

Une *switch_expression* est ajoutée à la sémantique de type prise en charge `switch` pour un contexte d'expression.

La syntaxe du langage C# est augmentée avec les productions syntaxiques suivantes :

antlr

```
multiplicative_expression
    : switch_expression
    | multiplicative_expression '*' switch_expression
    | multiplicative_expression '/' switch_expression
    | multiplicative_expression '%' switch_expression
    ;
switch_expression
    : range_expression 'switch' '{' '}'
    | range_expression 'switch' '{' switch_expression_arms ','? '}'
    ;
switch_expression_arms
    : switch_expression_arm
    | switch_expression_arms ',' switch_expression_arm
    ;
switch_expression_arm
    : pattern case_guard? '=>' expression
    ;
case_guard
    : 'when' null_coalescing_expression
    ;
```

Le *switch_expression* n'est pas autorisé en tant que *expression_statement*.

> Nous allons nous pencher sur la prochaine révision.

Le type de l' *switch_expression* est le *meilleur type commun* des expressions apparaissant à droite des `=>` jetons des *switch_expression_arm* s si un tel type existe et l'expression de chaque ARM de l'expression de commutateur peut être convertie implicitement en ce type. En outre, nous ajoutons une nouvelle *conversion d'expression de commutateur*, qui est une conversion implicite prédéfinie d'une expression de commutateur à chaque type `T` pour lequel il existe une conversion implicite de l'expression de chaque ARM vers `T` .

Il s'agit d'une erreur si le modèle de *switch_expression_arm* ne peut pas affecter le résultat, car un modèle précédent et une protection correspondent toujours.

Une expression de commutateur est dite *exhaustive* si une partie de l'expression de commutateur gère chaque valeur de son entrée. Le compilateur génère un avertissement si une expression de commutateur n'est pas *exhaustive*.

Lors de l'exécution, le résultat de l' *switch_expression* est la valeur de l' *expression* de la première *switch_expression_arm* pour laquelle l'expression située à gauche de la *switch_expression* correspond au modèle de l' *switch_expression_arm*, et pour laquelle la *case_guard* du *switch_expression_arm*, le cas échéant, est évaluée à `true` . S'il n'y a pas de *switch_expression_arm* de ce type, la *switch_expression* lève une instance de l'exception `System.Runtime.CompilerServices.SwitchExpressionException` .

Parenthèses facultatif lors du basculement sur un littéral de Tuple

Pour activer un littéral de tuple à l'aide de la *switch_Statement*, vous devez écrire ce qui semble être redondant parenthèses

```
C#

switch ((a, b))
{
```

Pour autoriser

```
C#

switch (a, b)
{
```

les parenthèses de l'instruction switch sont facultatives lorsque l'expression qui est basculée est un littéral de Tuple.

Ordre d'évaluation dans la mise en correspondance des modèles

Donner à la flexibilité du compilateur la possibilité de réorganiser les opérations exécutées pendant la mise en correspondance des modèles peut permettre de bénéficier d'une certaine flexibilité pour améliorer l'efficacité de la mise en correspondance des modèles. L'exigence (non appliquée) serait que les propriétés accessibles dans un modèle et les méthodes de déconstruction doivent être « pures » (effet secondaire gratuit, idempotent, etc.). Cela ne signifie pas que nous ajouterions la pureté comme un concept de langage, seulement que nous autoriserions la flexibilité du compilateur à réorganiser les opérations.

Résolution 2018-04-04 LDM: confirmée : le compilateur est autorisé à réorganiser les appels à `Deconstruct` , aux accès aux propriétés et aux appels de méthodes dans `ITuple` , et peut supposer que les valeurs retournées sont identiques à partir de plusieurs appels. Le compilateur ne doit pas appeler de fonctions qui ne peuvent pas affecter le résultat, et nous allons faire attention avant d'apporter des modifications à l'ordre d'évaluation généré par le compilateur à l'avenir.

Optimisations possibles

La compilation des critères spéciaux peut tirer parti des parties communes des modèles. Par exemple, si le test de type de niveau supérieur de deux modèles successifs dans un *switch_Statement* est du même type, le code généré peut ignorer le test de type pour le deuxième modèle.

Quand certains des modèles sont des entiers ou des chaînes, le compilateur peut générer le même type de code qu'il génère pour une instruction switch dans les versions antérieures du langage.

Pour plus d'informations sur ces types d'optimisations, consultez [Scott et Ramsey (2000)] .

Méthodes d'interface par défaut

Article • 13/05/2021

- [x] proposé
- [] Prototype : en cours
- [] Implémentation : aucune
- [] Spécification : en cours, en dessous

Résumé

Ajoutez la prise en charge des *méthodes d'extension virtuelles* (méthodes) dans les interfaces avec des implémentations concrètes. Une classe ou un struct qui implémente une telle interface doit avoir une implémentation unique *la plus spécifique* pour la méthode d'interface, implémentée par la classe ou le struct, ou héritée de ses classes de base ou interfaces. Les méthodes d'extension virtuelles permettent à un auteur d'API d'ajouter des méthodes à une interface dans les futures versions sans altérer la compatibilité source ou binaire avec les implémentations existantes de cette interface.

Celles-ci sont similaires aux « méthodes par défaut » de Java.

(En fonction de la technique d'implémentation probable) cette fonctionnalité requiert la prise en charge correspondante dans l'interface CLI/CLR. Les programmes qui tirent parti de cette fonctionnalité ne peuvent pas s'exécuter sur des versions antérieures de la plateforme.

Motivation

Les motivations principales de cette fonctionnalité sont les

- Les méthodes d'interface par défaut permettent à un auteur d'API d'ajouter des méthodes à une interface dans les versions futures sans altérer la compatibilité source ou binaire avec les implémentations existantes de cette interface.
- Cette fonctionnalité permet à C# d'interagir avec les API ciblant Android (Java) et iOS (SWIFT), qui prennent en charge des fonctionnalités similaires.
- Il s'avère que l'ajout d'implémentations d'interface par défaut fournit les éléments de la fonctionnalité de langage « traits » (https://en.wikipedia.org/wiki/Trait_(computer_programming)). Les caractéristiques ont prouvé qu'il s'agit d'une technique de programmation puissante (http://scg.unibe.ch/archive/papers/Scha03aTraits.pdf).

Conception détaillée

La syntaxe d'une interface est étendue pour permettre

- déclarations de membre qui déclarent des constantes, des opérateurs, des constructeurs statiques et des types imbriqués ;
- *corps* d'une méthode ou d'un indexeur, d'une propriété ou d'un accesseur d'événement (autrement dit, une implémentation « par défaut »);
- déclarations de membre qui déclarent des champs, des méthodes, des propriétés, des indexeurs et des événements statiques ;
- déclarations de membre à l'aide de la syntaxe d'implémentation d'interface explicite ; les
- Modificateurs d'accès explicites (l'accès par défaut est `public`).

Les membres avec des corps permettent à l'interface de fournir une implémentation « par défaut » pour la méthode dans les classes et les structs qui ne fournissent pas d'implémentation de substitution.

Les interfaces ne peuvent pas contenir l'état de l'instance. Alors que les champs statiques sont désormais autorisés, les champs d'instance ne sont pas autorisés dans les interfaces. Les propriétés auto des instances ne sont pas prises en charge dans les interfaces, car elles déclarent implicitement un champ masqué.

Les méthodes statiques et privées autorisent la refactorisation et l'organisation de code utiles utilisées pour implémenter l'API publique de l'interface.

Une substitution de méthode dans une interface doit utiliser la syntaxe d'implémentation d'interface explicite.

La déclaration d'un type de classe, d'un type struct ou d'un type enum dans la portée d'un paramètre de type qui a été déclaré avec un *variance_annotation* est une erreur. Par exemple, la déclaration de `C` ci-dessous est une erreur.

C#

```
interface IOuter<out T>
{
    class C { } // error: class declaration within the scope of variant type
parameter 'T'
}
```

Méthodes concrètes dans les interfaces

La forme la plus simple de cette fonctionnalité est la possibilité de déclarer une *méthode concrète* dans une interface, qui est une méthode avec un corps.

```
C#
```

```
interface IA
{
    void M() { WriteLine("IA.M"); }
}
```

Une classe qui implémente cette interface n'A pas besoin d'implémenter sa méthode concrète.

```
C#
```

```
class C : IA { } // OK

IA i = new C();
i.M(); // prints "IA.M"
```

La substitution finale de `IA.M` dans la classe `C` est la méthode concrète `M` déclarée dans `IA` . Notez qu'une classe n'hérite pas des membres de ses interfaces ; Cela n'est pas modifié par cette fonctionnalité :

```
C#
```

```
new C().M(); // error: class 'C' does not contain a member 'M'
```

Dans un membre d'instance d'une interface, `this` a le type de l'interface englobante.

Modificateurs dans les interfaces

La syntaxe d'une interface est assouplie pour autoriser les modificateurs sur ses membres. Les éléments suivants sont autorisés : `private` , `protected` , `internal` , `public` , `virtual` , `abstract` ,,, `sealed` `static` `extern` et `partial` .

> ***TODO***: Vérifiez les autres modificateurs existants.

Un membre d'interface dont la déclaration comprend un corps est un `virtual` membre, sauf si le `sealed` `private` modificateur ou est utilisé. Le `virtual` modificateur peut être utilisé sur un membre de fonction qui serait sinon implicitement `virtual` . De même, bien que `abstract` soit la valeur par défaut sur les membres d'interface sans corps, ce

modificateur peut être fourni explicitement. Un membre non virtuel peut être déclaré à l'aide du `sealed` mot clé.

Si un `private` `sealed` membre de fonction ou d'une interface n'a pas de corps, il s'agit d'une erreur. Une `private` fonction membre ne peut pas avoir le modificateur `sealed` .

Les modificateurs d'accès peuvent être utilisés sur les membres d'interface de tous les types de membres qui sont autorisés. Le niveau d'accès `public` est la valeur par défaut, mais elle peut être fournie de manière explicite.

> ***Problème d'ouverture :*** Nous devons spécifier la signification précise des modificateurs d'accès tels que `protected` et `internal` , ainsi que les déclarations qui ne se substituent pas à elles (dans une interface dérivée) ou qui les implémentent (dans une classe qui implémente l'interface).

Les interfaces peuvent déclarer des `static` membres, y compris des types imbriqués, des méthodes, des indexeurs, des propriétés, des événements et des constructeurs statiques. Le niveau d'accès par défaut pour tous les membres d'interface est `public` .

Les interfaces ne peuvent pas déclarer des constructeurs, des destructeurs ou des champs d'instance.

> *__Problème fermé :__ _ les déclarations d'opérateur _ doivent-elles être autorisées dans une interface ? Les opérateurs de conversion ne sont probablement pas, mais qu'en est-il des autres ? *Décision*: les opérateurs sont autorisés *except * pour les opérateurs de conversion, d'égalité et d'inégalité.Decision_*: Operators are permitted _except* for conversion, equality, and inequality operators.

> *__Problème fermé :__ _ doit `new` être autorisé sur les déclarations de membres d'interface qui masquent les membres des interfaces de base ? _ *Décision* * : Oui.

> *__Problème fermé :__ _ nous n'autorisons pas actuellement `partial` une interface ou ses membres. Cela nécessiterait une proposition distincte. _ *Décision* * : Oui. https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#permit-partial-in-interface

Substitutions dans les interfaces

Les déclarations override (c'est-à-dire celles contenant le `override` modificateur) permettent au programmeur de fournir une implémentation la plus spécifique d'un

membre virtuel dans une interface dans laquelle le compilateur ou le runtime n'en trouveraient pas un. Il permet également de transformer un membre abstrait d'une super interface en membre par défaut dans une interface dérivée. Une déclaration override est autorisée à substituer *explicitement* une méthode d'interface de base particulière en qualifiant la déclaration avec le nom de l'interface (aucun modificateur d'accès n'est autorisé dans ce cas). Les substitutions implicites ne sont pas autorisées.

C#

```
interface IA
{
    void M() { WriteLine("IA.M"); }
}
interface IB : IA
{
    override void IA.M() { WriteLine("IB.M"); } // explicitly named
}
interface IC : IA
{
    override void M() { WriteLine("IC.M"); } // implicitly named
}
```

Les déclarations override dans les interfaces ne peuvent pas être déclarées `sealed` .

Les `virtual` fonctions membres publiques dans une interface peuvent être substituées explicitement dans une interface dérivée (en qualifiant le nom dans la déclaration override avec le type d'interface qui a initialement déclaré la méthode et en omettant un modificateur d'accès).

`virtual` les fonctions membres dans une interface peuvent uniquement être substituées explicitement (pas implicitement) dans les interfaces dérivées, et les membres qui ne `public` peuvent pas être implémentés explicitement dans une classe ou une structure (pas implicitement). Dans les deux cas, le membre substitué ou implémenté doit être *accessible* là où il est substitué.

Réabstraction

Une méthode virtuelle (concrète) déclarée dans une interface peut être substituée pour être abstraite dans une interface dérivée

C#

```
interface IA
{
    void M() { WriteLine("IA.M"); }
}
```

```
interface IB : IA
{
    abstract void IA.M();
}
class C : IB { } // error: class 'C' does not implement 'IA.M'.
```

Le `abstract` modificateur n'est pas requis dans la déclaration de `IB.M` (il s'agit de la valeur par défaut dans les interfaces), mais il est probablement conseillé d'être explicite dans une déclaration override.

Cela est utile dans les interfaces dérivées où l'implémentation par défaut d'une méthode est inappropriée et une implémentation plus appropriée doit être fournie par l'implémentation de classes.

> ***Problème d'ouverture :*** La réabstraction doit-elle être autorisée ?

Règle de remplacement la plus spécifique

Nous avons besoin que chaque interface et classe aient une *substitution la plus spécifique* pour chaque membre virtuel parmi les substitutions apparaissant dans le type ou ses interfaces directes et indirectes. La *substitution la plus spécifique* est un remplacement unique qui est plus spécifique que tous les autres remplacements. S'il n'y a pas de substitution, le membre lui-même est considéré comme la substitution la plus spécifique.

Une substitution `M1` est considérée comme *plus spécifique* qu'une autre substitution `M2` si `M1` est déclarée sur le type `T1` , `M2` est déclarée sur le type `T2` et soit

1. `T1` contient `T2` parmi ses interfaces directes ou indirectes, ou
2. `T2` est un type interface `T1` , mais n'est pas un type interface.

Par exemple :

C#

```
interface IA
{
    void M() { WriteLine("IA.M"); }
}
interface IB : IA
{
    void IA.M() { WriteLine("IB.M"); }
}
interface IC : IA
{
    void IA.M() { WriteLine("IC.M"); }
```

```
}
interface ID : IB, IC { } // error: no most specific override for 'IA.M'
abstract class C : IB, IC { } // error: no most specific override for 'IA.M'
abstract class D : IA, IB, IC // ok
{
    public abstract void M();
}
```

La règle de remplacement la plus spécifique garantit qu'un conflit (c.-à-d. une ambiguïté résultant de l'héritage de diamant) est résolu explicitement par le programmeur au point où le conflit se produit.

Étant donné que nous prenons en charge les substitutions abstraites explicites dans les interfaces, nous pourrions le faire également dans les classes

C#

```
abstract class E : IA, IB, IC // ok
{
    abstract void IA.M();
}
```

> ***Problème d'ouverture***: devons-nous prendre en charge les remplacements abstraits d'interface explicites dans les classes ?

En outre, il s'agit d'une erreur si dans une déclaration de classe la substitution la plus spécifique d'une méthode d'interface est une substitution abstraite qui a été déclarée dans une interface. Il s'agit d'une règle existante retraitée à l'aide de la nouvelle terminologie.

C#

```
interface IF
{
    void M();
}
abstract class F : IF { } // error: 'F' does not implement 'IF.M'
```

Il est possible pour une propriété virtuelle déclarée dans une interface d'avoir une substitution la plus spécifique pour son `get` accesseur dans une interface et une substitution la plus spécifique pour son `set` accesseur dans une interface différente. Cela est considéré comme une violation de la règle de *remplacement la plus spécifique* .

Méthodes `static` et `private`

Étant donné que les interfaces peuvent désormais contenir du code exécutable, il est utile d'extraire le code commun en méthodes privées et statiques. Nous les autorisons maintenant dans les interfaces.

> *__Problème fermé _ : devons-nous prendre en charge les méthodes privées ?
> *Devons-nous prendre en charge les méthodes statiques ? Décision : Oui**

> ***Problème d'ouverture***: devons-nous autoriser les méthodes d'interface à être ou à un `protected` `internal` autre accès ? Dans ce cas, quelle est la sémantique ? Sont-elles par `virtual` défaut ? Dans ce cas, existe-t-il un moyen de les rendre non virtuels ?

> ***Problème d'ouverture***: si nous prenons en charge les méthodes statiques, devons-nous prendre en charge les opérateurs (statiques) ?

Appels de l'interface de base

Le code d'un type qui dérive d'une interface avec une méthode par défaut peut appeler explicitement l'implémentation de « base » de cette interface.

C#

```
interface I0
{
   void M() { Console.WriteLine("I0"); }
}
interface I1 : I0
{
   override void M() { Console.WriteLine("I1"); }
}
interface I2 : I0
{
   override void M() { Console.WriteLine("I2"); }
}
interface I3 : I1, I2
{
   // an explicit override that invoke's a base interface's default method
   void I0.M() { I2.base.M(); }
}
```

Une méthode d'instance (non statique) est autorisée à appeler l'implémentation d'une méthode d'instance accessible dans une interface de base directe de manière non

virtuelle en la nommant à l'aide de la syntaxe `base(Type).M` . Cela est utile quand une substitution qui doit être fournie en raison de l'héritage Diamond est résolue par la délégation à une implémentation de base particulière.

C#

```
interface IA
{
    void M() { WriteLine("IA.M"); }
}
interface IB : IA
{
    override void IA.M() { WriteLine("IB.M"); }
}
interface IC : IA
{
    override void IA.M() { WriteLine("IC.M"); }
}

class D : IA, IB, IC
{
    void IA.M() { base(IB).M(); }
}
```

Quand un `virtual` `abstract` membre ou est accessible à l'aide de la syntaxe `base(Type).M` , il est requis et `Type` contient une *substitution unique la plus spécifique* pour `M` .

Liaison de clauses de base

Les interfaces contiennent désormais des types. Ces types peuvent être utilisés dans la clause de base comme interfaces de base. Lors de la liaison d'une clause de base, nous pouvons être amenés à connaître l'ensemble des interfaces de base pour lier ces types (par exemple, pour effectuer une recherche dans ceux-ci et pour résoudre l'accès protégé). La signification de la clause de base d'une interface est donc définie de manière circulaire. Pour rompre le cycle, nous ajoutons une nouvelle règle de langage correspondant à une règle similaire déjà en place pour les classes.

Tout en déterminant la signification de la *interface_base* d'une interface, les interfaces de base sont temporairement supposées être vides. Intuitivement, cela garantit que la signification d'une clause de base ne peut pas dépendre de lui-même de manière récursive.

Nous avons utilisé les règles suivantes :

«Quand une classe B dérive d'une classe A, il s'agit d'une erreur de compilation qui dépend de B. Une classe **dépend directement de** sa classe de base directe (le cas échéant) et **dépend directement de** la ~~**classe**~~ dans laquelle elle est immédiatement imbriquée (le cas échéant). À partir de cette définition, l'ensemble complet des ~~**classes**~~ dont dépend une classe est la fermeture réflexive et transitive de la relation **directement dépend** de la relation.

Il s'agit d'une erreur au moment de la compilation pour qu'une interface hérite directement ou indirectement de lui-même. Les **interfaces de base** d'une interface sont les interfaces de base explicites et leurs interfaces de base. En d'autres termes, l'ensemble des interfaces de base est la fermeture transitive complète des interfaces de base explicites, leurs interfaces de base explicites, et ainsi de suite.

Nous les ajustons comme suit :

Quand une classe B dérive d'une classe A, il s'agit d'une erreur au moment de la compilation pour qu'un dépende de B. Une classe **dépend directement de** sa classe de base directe (le cas échéant) et **dépend directement** du ***type*** dans lequel elle est immédiatement imbriquée (le cas échéant).

Quand une interface IB étend une interface IA, il s'agit d'une erreur de compilation pour IA qui dépend de IB. Une interface **dépend directement de** ses interfaces de base directes (le cas échéant) et **dépend directement** du type dans lequel elle est immédiatement imbriquée (le cas échéant).

Étant donné ces définitions, le jeu complet de **types** dont dépend un type est la fermeture réflexive et transitive de la relation **directement dépend de** la relation.

Effet sur les programmes existants

Les règles présentées ici sont destinées à n'avoir aucun effet sur la signification des programmes existants.

Exemple 1 :

C#

```
interface IA
{
    void M();
}
class C: IA // Error: IA.M has no concrete most specific override in C
{
    public static void M() { } // method unrelated to 'IA.M' because static
}
```

Exemple 2 :

```C#
interface IA
{
    void M();
}
class Base: IA
{
    void IA.M() { }
}
class Derived: Base, IA // OK, all interface members have a concrete most specific override
{
    private void M() { } // method unrelated to 'IA.M' because private
}
```

Les mêmes règles donnent des résultats similaires à la situation analogue impliquant des méthodes d'interface par défaut :

```C#
interface IA
{
    void M() { }
}
class Derived: IA // OK, all interface members have a concrete most specific override
{
    private void M() { } // method unrelated to 'IA.M' because private
}
```

> *__Closed issue__ _ : confirmez qu'il s'agit d'une conséquence prévue de la spécification. *Décision : Oui**

Résolution de méthode d'exécution

> ***Problème fermé :*** La spécification doit décrire l'algorithme de résolution de la méthode d'exécution en face des méthodes par défaut de l'interface. Nous devons nous assurer que la sémantique est cohérente avec la sémantique de langue, par exemple les méthodes déclarées qui font et ne substituent pas ou n'implémentent pas une `internal` méthode.

API de prise en charge CLR

Pour que les compilateurs détectent lorsqu'ils sont en cours de compilation pour un Runtime qui prend en charge cette fonctionnalité, les bibliothèques de ces runtimes sont modifiées pour annoncer ce fait via l'API décrite dans https://github.com/dotnet/corefx/issues/17116 . Nous ajoutons

```
C#
```

```
namespace System.Runtime.CompilerServices
{
    public static class RuntimeFeature
    {
        // Presence of the field indicates runtime support
        public const string DefaultInterfaceImplementation =
nameof(DefaultInterfaceImplementation);
    }
}
```

> *__Ouvrir le problème__ _ : est-ce que le meilleur nom pour la fonctionnalité _CLR * ? La fonctionnalité CLR fait bien plus que cela (par exemple, assouplit les contraintes de protection, prend en charge les remplacements dans les interfaces, etc.). Par exemple, elle doit être appelée « méthodes concrètes dans les interfaces » ou « caractéristiques » ?

Autres zones à spécifier

- [] Il serait utile de cataloguer les genres d'effets de compatibilité source et binaire dus à l'ajout de méthodes d'interface par défaut et de remplacements à des interfaces existantes.

Inconvénients

Cette proposition requiert une mise à jour coordonnée de la spécification du CLR (pour prendre en charge les méthodes concrètes dans les interfaces et la résolution de méthode). Par conséquent, il est relativement « coûteux » et il peut être intéressant d'utiliser d'autres fonctionnalités que nous prévoyons d'exiger également des modifications du CLR.

Autres solutions

Aucun.

Questions non résolues

- Les questions ouvertes sont dénommées tout au long de la proposition, ci-dessus.
- Pour obtenir https://github.com/dotnet/csharplang/issues/406 la liste des questions ouvertes, consultez également.
- La spécification détaillée doit décrire le mécanisme de résolution utilisé lors de l'exécution pour sélectionner la méthode précise à appeler.
- L'interaction des métadonnées produites par les nouveaux compilateurs et consommées par les compilateurs plus anciens doit être traitée en détail. Par exemple, nous devons nous assurer que la représentation de métadonnées que nous utilisons n'entraîne pas l'ajout d'une implémentation par défaut dans une interface pour arrêter une classe existante qui implémente cette interface lorsqu'elle est compilée par un compilateur plus ancien. Cela peut affecter la représentation des métadonnées que nous pouvons utiliser.
- La conception doit prendre en compte l'interopérabilité avec d'autres langages et des compilateurs existants pour d'autres langages.

Questions résolues

Remplacement abstrait

La spécification préliminaire précédente contenait la possibilité de « reabstract » une méthode héritée :

C#

```
interface IA
{
    void M();
}
interface IB : IA
{
    override void M() { }
}
interface IC : IB
{
    override void M(); // make it abstract again
}
```

Mes notes pour 2017-03-20 ont montré que nous avons décidé de ne pas les autoriser. Toutefois, il existe au moins deux cas d'utilisation :

1. Les API Java, avec lesquelles certains utilisateurs de cette fonctionnalité espèrent interagir, dépendent de cette fonctionnalité.
2. La programmation avec les *caractéristiques* tire parti de cette. La réabstraction est l'un des éléments de la fonctionnalité de langage « traits » (https://en.wikipedia.org/wiki/Trait_(computer_programming)) . Les éléments suivants sont autorisés avec les classes :

C#

```
public abstract class Base
{
    public abstract void M();
}
public abstract class A : Base
{
    public override void M() { }
}
public abstract class B : A
{
    public override abstract void M(); // reabstract Base.M
}
```

Malheureusement, ce code ne peut pas être refactorisé comme un ensemble d'interfaces (traits) à moins que cela ne soit autorisé. Le *principe Jared de cupidité* doit être autorisé.

> ***Problème fermé :*** La réabstraction doit-elle être autorisée ? Oui Mes notes sont incorrectes. Les Notes LDM indiquent que la réabstraction est autorisée dans une interface. Pas dans une classe.

Modificateur virtuel et modificateur sealed

À partir de Aleksey Tsingauz :

> Nous avons décidé d'autoriser les modificateurs explicitement indiqués sur les membres d'interface, à moins qu'il y ait une raison d'en interdire certains. Cela pose une question intéressante concernant le modificateur virtuel. Doit-il être requis sur les membres avec l'implémentation par défaut ?
>
> Nous pouvons affirmer que :
>
> - s'il n'existe aucune implémentation et que ni Virtual, ni sealed ne sont spécifiés, nous supposons que le membre est abstrait.
> - s'il existe une implémentation et que ni abstract, ni sealed ne sont spécifiés, nous supposons que le membre est virtuel.

> - un modificateur sealed est requis pour rendre une méthode non virtuelle ou abstraite.
>
> Vous pouvez également indiquer que le modificateur virtuel est requis pour un membre virtuel. Par exemple, s'il existe un membre dont l'implémentation n'est pas explicitement marquée avec le modificateur Virtual, il n'est ni virtuel ni abstrait. Cette approche peut offrir une meilleure expérience quand une méthode est déplacée d'une classe vers une interface :
>
> - une méthode abstraite reste abstraite.
> - une méthode virtuelle reste virtuelle.
> - une méthode sans modificateur ne reste ni virtuelle, ni abstraite.
> - un modificateur sealed ne peut pas être appliqué à une méthode qui n'est pas une substitution.
>
> Qu'en pensez-vous ?

> ***Problème fermé :*** Une méthode concrète (avec implémentation) doit-elle être implicitement `virtual` ? Oui

Décisions : Créé dans le LDM 2017-04-05 :

1. les non virtuelles doivent être exprimées explicitement par `sealed` ou `private` .
2. `sealed` est le mot clé pour rendre les membres d'instance d'interface avec des corps non virtuels
3. Nous souhaitons autoriser tous les modificateurs dans les interfaces
4. L'accessibilité par défaut pour les membres d'interface est publique, y compris les types imbriqués
5. les fonctions membres privées dans les interfaces sont implicitement sealed et ne `sealed` sont pas autorisées sur elles.
6. Les classes privées (dans les interfaces) sont autorisées et peuvent être sealed, ce qui signifie qu'elles sont scellées dans le sens de la classe de Sealed.
7. L'absence d'une bonne proposition, partielle n'est toujours pas autorisée sur les interfaces ou leurs membres.

Compatibilité binaire 1

Quand une bibliothèque fournit une implémentation par défaut

```
C#
```

```
interface I1
{
    void M() { Impl1 }
}
interface I2 : I1
{
}
class C : I2
{
}
```

Nous savons que l'implémentation de `I1.M` dans `C` est `I1.M` . Que se passe-vous si l'assembly contenant `I2` est modifié comme suit et recompilé

C#

```
interface I2 : I1
{
    override void M() { Impl2 }
}
```

mais `C` n'est pas recompilé. Que se passe-t-il lorsque le programme est exécuté ? Un appel de `(C as I1).M()`

1. S'exécute `I1.M`
2. S'exécute `I2.M`
3. Lève un certain type d'erreur d'exécution

Décision : Rendu 2017-04-11 : exécutions `I2.M` , ce qui correspond à la substitution la plus spécifique non ambiguë au moment de l'exécution.

Accesseurs d'événement (fermé)

> ***Problème fermé :*** Un événement peut-il être remplacé par « par morceaux » ?

Prenons le cas suivant :

C#

```
public interface I1
{
    event T e1;
}
public interface I2 : I1
{
```

```
    override event T
    {
        add { }
        // error: "remove" accessor missing
    }
}
```

Cette implémentation « partielle » de l'événement n'est pas autorisée, car, comme dans une classe, la syntaxe d'une déclaration d'événement n'autorise qu'un seul accesseur ; les deux (ou aucune) doivent être fournis. Vous pourriez accomplir la même chose en autorisant l'accesseur remove abstrait dans la syntaxe à être implicitement abstrait par l'absence d'un corps :

C#

```
public interface I1
{
    event T e1;
}
public interface I2 : I1
{
    override event T
    {
        add { }
        remove; // implicitly abstract
    }
}
```

Notez qu' *il s'agit d'une nouvelle syntaxe (proposée)*. Dans la grammaire actuelle, les accesseurs d'événement ont un corps obligatoire.

> ***Problème fermé :*** Un accesseur d'événement peut-il être (implicitement) abstrait par l'omission d'un corps, de la même façon que les méthodes des accesseurs d'interfaces et de propriétés sont abstraites (implicitement) par l'omission d'un corps ?

Décision : (2017-04-18) non, les déclarations d'événements requièrent des accesseurs concrets (ou aucun).

Réabstraction dans une classe (fermée)

Problème fermé : Nous devons confirmer que cela est autorisé (sinon, l'ajout d'une implémentation par défaut serait une modification avec rupture) :

C#

```
interface I1
{
    void M() { }
}
abstract class C : I1
{
    public abstract void M(); // implement I1.M with an abstract method in C
}
```

Décision : (2017-04-18) Oui, l'ajout d'un corps à une déclaration de membre d'interface ne doit pas rompre la valeur C.

Remplacement sealed (fermé)

La question précédente suppose implicitement que le `sealed` modificateur peut être appliqué à un `override` dans une interface. Cela contredit la spécification brouillon. Voulez-vous autoriser le scellement d'un remplacement ? Les effets de la compatibilité source et binaire de la fermeture doivent être pris en compte.

> ***Problème fermé :*** Faut-il autoriser le scellement d'un remplacement ?

Décision : (2017-04-18) n'est pas autorisé `sealed` sur les remplacements dans les interfaces. La seule utilisation de `sealed` sur les membres d'interface consiste à les rendre non virtuelles dans leur déclaration initiale.

Héritage et classes Diamond (fermé)

Le brouillon de la proposition préfère les remplacements de classe aux remplacements d'interface dans les scénarios d'héritage Diamond :

> Nous avons besoin que chaque interface et classe aient une *substitution la plus spécifique* pour chaque méthode d'interface parmi les substitutions apparaissant dans le type ou ses interfaces directes et indirectes. La *substitution la plus spécifique* est un remplacement unique qui est plus spécifique que tous les autres remplacements. S'il n'y a pas de substitution, la méthode elle-même est considérée comme la substitution la plus spécifique.
>
> Une substitution `M1` est considérée comme *plus spécifique* qu'une autre substitution `M2` si `M1` est déclarée sur le type `T1` , `M2` est déclarée sur le type `T2` et soit
>
> 1. `T1` contient `T2` parmi ses interfaces directes ou indirectes, ou
> 2. `T2` est un type interface `T1` , mais n'est pas un type interface.

Le scénario est le

```C#
interface IA
{
    void M();
}
interface IB : IA
{
    override void M() { WriteLine("IB"); }
}
class Base : IA
{
    void IA.M() { WriteLine("Base"); }
}
class Derived : Base, IB // allowed?
{
    static void Main()
    {
        Ia a = new Derived();
        a.M();           // what does it do?
    }
}
```

Nous devons confirmer ce comportement (ou décider autrement)

> *__Problème fermé :__ _ confirmez la spécification du brouillon, ci-dessus, pour _most remplacement spécifique * tel qu'il s'applique aux classes et interfaces mixtes (une classe est prioritaire sur une interface). Consultez https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-19.md#diamonds-with-classes .

Méthodes d'interface et structs (fermé)

Il existe des interactions fâcheuses entre les méthodes et structs de l'interface par défaut.

```C#
interface IA
{
    public void M() { }
}
struct S : IA
{
}
```

Notez que les membres d'interface ne sont pas hérités :

C#

```
var s = default(S);
s.M(); // error: 'S' does not contain a member 'M'
```

Par conséquent, le client doit Box le struct pour appeler des méthodes d'interface

C#

```
IA s = default(S); // an S, boxed
s.M(); // ok
```

De cette façon, le boxing est à l'encontre des principaux avantages d'un `struct` type. En outre, toutes les méthodes de mutation n'auront aucun effet apparent, car elles fonctionnent sur une *copie boxed* du struct :

C#

```
interface IB
{
    public void Increment() { P += 1; }
    public int P { get; set; }
}
struct T : IB
{
    public int P { get; set; } // auto-property
}

T t = default(T);
Console.WriteLine(t.P); // prints 0
(t as IB).Increment();
Console.WriteLine(t.P); // prints 0
```

> ***Problème fermé :*** Que pouvons-nous faire pour cela :
>
> 1. Interdit `struct` à un d'hériter d'une implémentation par défaut. Toutes les méthodes d'interface seraient traitées comme abstraites dans un `struct` . Ensuite, nous pouvons prendre du temps ultérieurement pour décider comment améliorer son fonctionnement.
> 2. Trouvez un type de stratégie de génération de code qui évite la conversion boxing. À l'intérieur d'une méthode telle que `IB.Increment` , le type de `this` serait peut-être similaire à un paramètre de type avec une limite de `IB` . En conjonction avec cela, pour éviter le boxing dans l'appelant, les méthodes non

abstraites sont héritées des interfaces. Cela peut augmenter considérablement le fonctionnement du compilateur et de l'implémentation du CLR.
3. Ne vous inquiétez pas et ne vous contentez pas de gênant.
4. Autres idées ?

Décision : Ne vous inquiétez pas et ne vous contentez pas de gênant. Consultez https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-19.md#structs-and-default-implementations .

Appels de l'interface de base (fermé)

Le brouillon de spécifications suggère une syntaxe pour les appels de l'interface de base inspirée par Java : `Interface.base.M()` . Nous devons sélectionner une syntaxe, au moins pour le prototype initial. Mon favori est `base<Interface>.M()` .

> ***Problème fermé :*** Quelle est la syntaxe d'un appel de membre de base ?

Décision : La syntaxe est `base(Interface).M()` . Consultez https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-19.md#base-invocation . L'interface ainsi nommée doit être une interface de base, mais elle n'a pas besoin d'être une interface de base directe.

> ***Problème d'ouverture :*** Les appels d'interface de base doivent-ils être autorisés dans les membres de classe ?

Décision: Oui. https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-19.md#base-invocation

Substitution de membres d'interface non publics (fermés)

Dans une interface, les membres non publics des interfaces de base sont substitués à l'aide du `override` modificateur. S'il s'agit d'une substitution « explicite » qui nomme l'interface contenant le membre, le modificateur d'accès est omis.

> ***Problème fermé :*** S'il s'agit d'une substitution « implicite » qui ne nomme pas l'interface, le modificateur d'accès doit-il correspondre ?

Décision : Seuls les membres publics peuvent être substitués implicitement, et l'accès doit correspondre. Consultez

https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-18.md#dim-implementing-a-non-public-interface-member-not-in-list .

> ***Problème d'ouverture :*** Le modificateur d'accès est-il obligatoire, facultatif ou omis sur une substitution explicite telle que `override void IB.M() {}` ?

> ***Problème d'ouverture :*** Est `override` requis, facultatif ou omis sur une substitution explicite telle que `void IB.M() {}` ?

Comment l'un implémente-t-il un membre d'interface non public dans une classe ? Peut-être devez-vous le faire explicitement ?

C#

```
interface IA
{
    internal void MI();
    protected void MP();
}
class C : IA
{
    // are these implementations?
    internal void MI() {}
    protected void MP() {}
}
```

> ***Problème fermé :*** Comment l'un implémente-t-il un membre d'interface non public dans une classe ?

Décision : Vous pouvez uniquement implémenter des membres d'interface non publics explicitement. Consultez https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-04-18.md#dim-implementing-a-non-public-interface-member-not-in-list .

Decision: aucun `override` mot clé n'est autorisé sur les membres d'interface. https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#does-an-override-in-an-interface-introduce-a-new-member

Compatibilité binaire 2 (fermée)

Considérez le code suivant dans lequel chaque type se trouve dans un assembly distinct.

C#

```
interface I1
{
    void M() { Impl1 }
}
interface I2 : I1
{
    override void M() { Impl2 }
}
interface I3 : I1
{
}
class C : I2, I3
{
}
```

Nous savons que l'implémentation de `I1.M` dans `C` est `I2.M` . Que se passe-vous si l'assembly contenant `I3` est modifié comme suit et recompilé

C#

```
interface I3 : I1
{
    override void M() { Impl3 }
}
```

mais `C` n'est pas recompilé. Que se passe-t-il lorsque le programme est exécuté ? Un appel de `(C as I1).M()`

1. S'exécute `I1.M`
2. S'exécute `I2.M`
3. S'exécute `I3.M`
4. 2 ou 3, déterministe
5. Lève un type d'exception d'exécution

Décision: levez une exception (5). Consultez https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#issues-in-default-interface-methods .

Autoriser `partial` dans l'interface ? légendes

Étant donné que les interfaces peuvent être utilisées de manière analogue à la façon dont les classes abstraites sont utilisées, il peut être utile de les déclarer `partial` . Cela serait particulièrement utile pour les générateurs.

> ***Proposition :*** Supprimez la restriction de langue pour laquelle les interfaces et les membres d'interfaces ne peuvent pas être déclarés `partial` .

Décision: Oui. Consultez https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#permit-partial-in-interface .

`Main` dans une interface ? légendes

> ***Problème d'ouverture :*** Une `static Main` méthode dans une interface peut-elle être le point d'entrée du programme ?

Décision: Oui. Consultez https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#main-in-an-interface .

Confirmer l'intention de prendre en charge les méthodes publiques non virtuelles (fermées)

Pouvons-nous confirmer (ou inverser) notre décision d'autoriser les méthodes publiques non virtuelles dans une interface ?

```
C#

interface IA
{
    public sealed void M() { }
}
```

> ***Problème semi-fermé :*** (2017-04-18) nous pensons qu'il va être utile, mais y revenir. Il s'agit d'un bloc de déclenchement de modèle mental.

Décision: Oui. https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#confirm-that-we-support-public-non-virtual-methods .

Un `override` dans une interface présente-t-il un nouveau membre ? légendes

Il existe plusieurs façons d'observer si une déclaration de remplacement introduit un nouveau membre ou non.

C#

```
interface IA
{
    void M(int x) { }
}
interface IB : IA
{
    override void M(int y) { }
}
interface IC : IB
{
    static void M2()
    {
        M(y: 3); // permitted?
    }
    override void IB.M(int z) { } // permitted? What does it override?
}
```

> ***Problème d'ouverture :*** Une déclaration override dans une interface présente-t-elle un nouveau membre ? légendes

Dans une classe, une méthode de substitution est « visible » dans certains sens. Par exemple, les noms de ses paramètres ont la priorité sur les noms des paramètres dans la méthode substituée. Il peut être possible de dupliquer ce comportement dans les interfaces, car il y a toujours une substitution la plus spécifique. Mais voulez-vous dupliquer ce comportement ?

Il est également possible de « remplacer » une méthode override ? Discutable

Decision: aucun `override` mot clé n'est autorisé sur les membres d'interface. https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#does-an-override-in-an-interface-introduce-a-new-member .

Propriétés avec un accesseur private (fermé)

Nous disons que les membres privés ne sont pas virtuels et que la combinaison de Virtual et Private est interdite. Mais qu'en est-il d'une propriété avec un accesseur private ?

C#

```
interface IA
{
    public virtual int P
    {
```

```
        get => 3;
        private set => { }
    }
}
```

Est-ce autorisé ? L'accesseur est-il `set` présent `virtual` ou non ? Peut-il être substitué là où il est accessible ? Les éléments suivants implémentent-ils implicitement uniquement l' `get` accesseur ?

C#

```
class C : IA
{
    public int P
    {
        get => 4;
        set { }
    }
}
```

Est probablement une erreur due à l'IA. P. Set n'est pas virtuel et aussi parce qu'il n'est pas accessible ?

C#

```
class C : IA
{
    int IA.P
    {
        get => 4;
        set { }
    }
}
```

Décision: le premier exemple semble valide, tandis que le dernier ne le fait pas. Cela est résolu de façon similaire à la façon dont il fonctionne déjà en C#. https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#properties-with-a-private-accessor

Appels de l'interface de base, arrondi 2 (fermé)

Notre « résolution » précédente à la manière de gérer les appels de base n'offre pas en fait suffisamment d'expressivité. Il s'avère que en C# et le CLR, contrairement à Java, vous devez spécifier l'interface contenant la déclaration de méthode et l'emplacement de l'implémentation que vous souhaitez appeler.

Je propose la syntaxe suivante pour les appels de base dans les interfaces. Je n'aime pas, mais il illustre ce que toute syntaxe doit pouvoir exprimer :

C#

```
interface I1 { void M(); }
interface I2 { void M(); }
interface I3 : I1, I2 { void I1.M() { } void I2.M() { } }
interface I4 : I1, I2 { void I1.M() { } void I2.M() { } }
interface I5 : I3, I4
{
    void I1.M()
    {
        base<I3>(I1).M(); // calls I3's implementation of I1.M
        base<I4>(I1).M(); // calls I4's implementation of I1.M
    }
    void I2.M()
    {
        base<I3>(I2).M(); // calls I3's implementation of I2.M
        base<I4>(I2).M(); // calls I4's implementation of I2.M
    }
}
```

S'il n'y a pas d'ambiguïté, vous pouvez l'écrire plus simplement

C#

```
interface I1 { void M(); }
interface I3 : I1 { void I1.M() { } }
interface I4 : I1 { void I1.M() { } }
interface I5 : I3, I4
{
    void I1.M()
    {
        base<I3>.M(); // calls I3's implementation of I1.M
        base<I4>.M(); // calls I4's implementation of I1.M
    }
}
```

ou

C#

```
interface I1 { void M(); }
interface I2 { void M(); }
interface I3 : I1, I2 { void I1.M() { } void I2.M() { } }
interface I5 : I3
{
    void I1.M()
    {
        base(I1).M(); // calls I3's implementation of I1.M
```

```
    }
    void I2.M()
    {
        base(I2).M(); // calls I3's implementation of I2.M
    }
}
```

ou

C#

```
interface I1 { void M(); }
interface I3 : I1 { void I1.M() { } }
interface I5 : I3
{
    void I1.M()
    {
        base.M(); // calls I3's implementation of I1.M
    }
}
```

Décision: décidée `base(N.I1<T>).M(s)` , en concédant que si nous avons une liaison d'appel, un problème peut survenir ici plus tard. https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-11-14.md#default-interface-implementations

Avertissement indiquant que la structure n'implémente pas la méthode par défaut ? légendes

@vancem déclare que nous devrions sérieusement envisager de générer un avertissement si une déclaration de type de valeur ne peut pas substituer une méthode d'interface, même si elle hérite d'une implémentation de cette méthode à partir d'une interface. Dans la mesure où elle provoque des appels de conversion boxing et de sous-mines.

Décision: cela semble plus adapté à un analyseur. Il semble également que cet avertissement soit bruyant, car il se déclenche même si la méthode d'interface par défaut n'est jamais appelée et qu'il n'y aura jamais de boxing. https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#warning-for-struct-not-implementing-default-method

Constructeurs statiques d'interface (fermés)

Quand les constructeurs statiques d'interface s'exécutent-ils ? La version actuelle de l'interface CLI suggère qu'elle se produit lors de l'accès à la première méthode ou au champ statique. S'il n'y en a pas, il n'est peut-être jamais exécuté ?

[2018-10-09 l'équipe CLR propose « va mettre en miroir ce que nous faisons pour les points de contrôle de l'accès à chaque méthode d'instance) »]

Décision: les constructeurs statiques sont également exécutés à l'entrée dans des méthodes d'instance, si le constructeur statique n'en était pas `beforefieldinit` , auquel cas les constructeurs statiques sont exécutés avant l'accès au premier champ statique. https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-10-17.md#when-are-interface-static-constructors-run

Réunions de conception

Notes de réunion LDM 2017-03-08 Notes de réunion LDM 2017-03-21 2017-03-23 Réunion « comportement du CLR pour les méthodes d'interface par défaut » Notes de réunion LDM 2017-04-05 Notes de réunion LDM 2017-04-11 Notes de réunion LDM 2017-04-18 Notes de réunion LDM 2017-04-19 Notes de réunion LDM 2017-05-17 Notes de réunion LDM 2017-05-31 Notes de réunion LDM 2017-06-14 Notes de réunion LDM 2018-10-17 notes de réunion LDM 2018-11-14

Flux asynchrones

Article • 13/05/2021

- [x] proposé
- [x] prototype
- [] Implémentation
- [] Spécification

Résumé

C# prend en charge les méthodes d'itérateur et les méthodes Async, mais aucune prise en charge d'une méthode qui est à la fois un itérateur et une méthode Async. Nous devons résoudre ce cas en autorisant l'utilisation de for `await` dans une nouvelle forme d' `async` iterator, qui retourne un `IAsyncEnumerable<T>` ou `IAsyncEnumerator<T>` plutôt qu'un `IEnumerable<T>` ou `IEnumerator<T>` , avec `IAsyncEnumerable<T>` un consommable dans un nouveau `await foreach` . Une `IAsyncDisposable` interface est également utilisée pour activer le nettoyage asynchrone.

Discussion connexe

- https://github.com/dotnet/roslyn/issues/261
- https://github.com/dotnet/roslyn/issues/114

Conception détaillée

Interfaces

IAsyncDisposable

Il y a eu un grand débat sur `IAsyncDisposable` (par exemple https://github.com/dotnet/roslyn/issues/114) , s'il s'agit d'une bonne idée. Toutefois, il s'agit d'un concept obligatoire à ajouter pour la prise en charge des itérateurs Async. Étant donné que les `finally` blocs peuvent contenir des `await` s et que les `finally` blocs doivent être exécutés dans le cadre de la suppression d'itérateurs, nous avons besoin d'une suppression asynchrone. Il est également généralement utile que tout nettoyage des ressources peut prendre un certain temps, par exemple, pour fermer des

fichiers (en exigeant des vidages), annuler l'inscription des rappels et fournir un moyen de savoir quand la désinscription est terminée, etc.

L'interface suivante est ajoutée aux bibliothèques .NET de base (par exemple, System. private. CoreLib/System. Runtime) :

```C#
namespace System
{
    public interface IAsyncDisposable
    {
        ValueTask DisposeAsync();
    }
}
```

Comme avec `Dispose` , l'appel de `DisposeAsync` plusieurs fois est acceptable, et les appels suivants de la première doivent être traités en tant que NOPS, ce qui a pour conséquence de retourner une tâche ayant abouti de manière synchrone (`DisposeAsync` ne doit pas être thread-safe, mais n'a pas besoin de prendre en charge l'appel simultané). En outre, les types peuvent implémenter `IDisposable` `IAsyncDisposable` à la fois et, et si c'est le cas, il est tout à fait acceptable d'appeler, puis, `Dispose` `DisposeAsync` ou vice versa, mais seul le premier doit être significatif et les appels suivants doivent être un NOP. Par conséquent, si un type implémente à la fois, les consommateurs sont encouragés à appeler une fois et une seule fois la méthode la plus pertinente en fonction du contexte, `Dispose` dans des contextes synchrones et dans des contextes `DisposeAsync` asynchrones.

(Je laisse une discussion sur la façon dont `IAsyncDisposable` interagit avec `using` pour une discussion distincte. La couverture de la manière dont elle interagit avec `foreach` est traitée plus loin dans cette proposition.)

Alternatives prises en compte :

- *`DisposeAsync` acceptation d' `CancellationToken` un*: bien qu'en théorie, il est logique de supprimer tout élément asynchrone, de le nettoyer, de fermer des éléments, de free'ing des ressources, etc., ce qui n'est généralement pas un élément qui doit être annulé ; le nettoyage reste important pour le travail annulé. La même chose `CancellationToken` qui a provoqué l'annulation du travail réel est généralement le même jeton passé à, ce qui n'a `DisposeAsync` rien, `DisposeAsync` car l'annulation du travail aurait `DisposeAsync` pour effet d'être un NOP. Si une personne souhaite éviter d'être bloquée en attente de suppression, elle peut éviter

d'attendre le résultat `ValueTask` , ou l'attendre uniquement pendant un certain laps de temps.

- *`DisposeAsync` retour d' `Task` un*: maintenant qu'un non générique `ValueTask` existe et peut être construit à partir d'un `IValueTaskSource` , le retour `ValueTask` de `DisposeAsync` permet à un objet existant d'être réutilisé comme promesse représentant l'achèvement asynchrone éventuel de `DisposeAsync` , en enregistrant une `Task` allocation dans le cas où se `DisposeAsync` termine de façon asynchrone.
- *Configuration `DisposeAsync` avec un `bool continueOnCapturedContext` (`ConfigureAwait`)*: bien qu'il puisse y avoir des problèmes liés à la façon dont un tel concept est exposé à `using` , et à `foreach` d'autres constructions de langage qui consomment ce concept, du point de vue de l'interface, il n'effectue aucune action `await` et il n'y a rien à configurer... les consommateurs du `ValueTask` peuvent l'utiliser, mais ils le souhaitent.
- héritage : étant donné qu'un seul ou l'autre doit être utilisé, il n'est pas judicieux de forcer les types à implémenter les deux. *`IAsyncDisposable` `IDisposable`*
- *`IDisposableAsync` au lieu `IAsyncDisposable` de* cela : nous avons suivi l'appellation des éléments/types « asynchrone », alors que les opérations sont « asynchrones ». ainsi, les types ont « Async » comme préfixe et les méthodes ont « Async » comme suffixe.

IAsyncEnumerable/IAsyncEnumerator

Deux interfaces sont ajoutées aux bibliothèques .NET de base :

C#

```
namespace System.Collections.Generic
{
    public interface IAsyncEnumerable<out T>
    {
        IAsyncEnumerator<T> GetAsyncEnumerator(CancellationToken
cancellationToken = default);
    }

    public interface IAsyncEnumerator<out T> : IAsyncDisposable
    {
        ValueTask<bool> MoveNextAsync();
        T Current { get; }
    }
}
```

La consommation classique (sans fonctionnalités de langage supplémentaires) ressemble à ceci :

C#

```
IAsyncEnumerator<T> enumerator = enumerable.GetAsyncEnumerator();
try
{
    while (await enumerator.MoveNextAsync())
    {
        Use(enumerator.Current);
    }
}
finally { await enumerator.DisposeAsync(); }
```

Options ignorées prises en compte :

- `Task<bool> MoveNextAsync(); T current { get; }`: L'utilisation de `Task<bool>` prend en charge l'utilisation d'un objet de tâche mis en cache pour représenter des appels synchrones et réussis `MoveNextAsync` , mais une allocation serait toujours nécessaire pour une exécution asynchrone. En retournant `ValueTask<bool>` , nous autorisons l'objet énumérateur à lui-même à implémenter `IValueTaskSource<bool>` et à être utilisé comme stockage pour le `ValueTask<bool>` retourné de `MoveNextAsync` , ce qui, à son tour, permet de réduire considérablement les surplus.
- `ValueTask<(bool, T)> MoveNextAsync();`: Ce n'est pas seulement plus difficile à consommer, mais cela signifie qu'il `T` ne peut plus être covariant.
- `ValueTask<T?> TryMoveNextAsync();`: N'est pas covariant.
- `Task<T?> TryMoveNextAsync();`: N'est pas covariant, les allocations à chaque appel, etc.
- `ITask<T?> TryMoveNextAsync();`: N'est pas covariant, les allocations à chaque appel, etc.
- `ITask<(bool,T)> TryMoveNextAsync();`: N'est pas covariant, les allocations à chaque appel, etc.
- `Task<bool> TryMoveNextAsync(out T result);`: Le `out` résultat doit être défini lorsque l'opération est retournée de manière synchrone, et non pas lorsqu'il termine de façon asynchrone la tâche potentiellement assez longtemps à l'avenir. à ce stade, il n'y a aucun moyen de communiquer le résultat.
- `IAsyncEnumerator<T>` *non implémenté* `IAsyncDisposable`: nous pourrions choisir de les séparer. Toutefois, cela complique certaines autres parties de la proposition, car le code doit être en mesure de faire face à la possibilité qu'un énumérateur ne fournisse pas de suppression, ce qui complique l'écriture des applications auxiliaires basées sur les modèles. En outre, il est courant que les énumérateurs aient besoin d'être éliminés (par exemple, tout itérateur Async C# qui a un bloc finally, la plupart des éléments qui énumèrent les données à partir d'une connexion réseau, etc.), et si ce n'est pas le cas, il est simple d'implémenter la

méthode simplement `public ValueTask DisposeAsync() => default(ValueTask);` avec une surcharge supplémentaire minimale.

- _ `IAsyncEnumerator<T> GetAsyncEnumerator()` : Aucun paramètre de jeton d'annulation.

Alternative viable :

C#

```
namespace System.Collections.Generic
{
    public interface IAsyncEnumerable<out T>
    {
        IAsyncEnumerator<T> GetAsyncEnumerator();
    }

    public interface IAsyncEnumerator<out T> : IAsyncDisposable
    {
        ValueTask<bool> WaitForNextAsync();
        T TryGetNext(out bool success);
    }
}
```

`TryGetNext` est utilisé dans une boucle interne pour consommer des éléments avec un seul appel d'interface, à condition qu'ils soient disponibles de façon synchrone. Lorsque l'élément suivant ne peut pas être récupéré de façon synchrone, il retourne false, et chaque fois qu'il retourne false, un appelant doit appeler `WaitForNextAsync` pour attendre que l'élément suivant soit disponible ou pour déterminer qu'il n'y aura jamais d'autre élément. La consommation classique (sans fonctionnalités de langage supplémentaires) ressemble à ceci :

C#

```
IAsyncEnumerator<T> enumerator = enumerable.GetAsyncEnumerator();
try
{
    while (await enumerator.WaitForNextAsync())
    {
        while (true)
        {
            int item = enumerator.TryGetNext(out bool success);
            if (!success) break;
            Use(item);
        }
    }
}
finally { await enumerator.DisposeAsync(); }
```

L’avantage est qu’il s’agit d’un double pli, d’un secondaire et d’un principal :

- *Minor : permet à un énumérateur de prendre en charge plusieurs consommateurs*. Il peut exister des scénarios dans lesquels il est utile qu’un énumérateur prenne en charge plusieurs consommateurs simultanés. Cela ne peut pas être atteint quand `MoveNextAsync` et `Current` sont séparés de telle sorte qu’une implémentation ne peut pas rendre son utilisation atomique. En revanche, cette approche fournit une méthode unique `TryGetNext` qui prend en charge le push de l’énumérateur vers l’avant et l’obtention de l’élément suivant, de sorte que l’énumérateur peut activer l’atomicité si vous le souhaitez. Toutefois, il est probable que ces scénarios peuvent également être activés en donnant à chaque consommateur son propre énumérateur à partir d’un énumérable partagé. En outre, nous ne souhaitons pas imposer que chaque énumérateur prenne en charge l’utilisation simultanée, car cela ajouterait des surcharges non triviales au cas majoritaire qui n’en a pas besoin, ce qui signifie qu’un consommateur de l’interface ne pouvait généralement pas s’appuyer sur cette méthode.
- *Majeur : performances*. L' `MoveNextAsync` / `Current` approche nécessite deux appels d’interface par opération, tandis que la `WaitForNextAsync` / `TryGetNext` plupart des itérations se terminent de manière synchrone, ce qui permet une boucle interne étroite avec `TryGetNext` , de sorte que nous n’ayons qu’un seul appel d’interface par opération. Cela peut avoir un impact mesurable dans les situations où les appels d’interface dominent le calcul.

Toutefois, il existe des inconvénients non négligeables, y compris une complexité considérablement accrue lors de l’utilisation manuelle de ces derniers, ainsi qu’une meilleure chance d’introduire des bogues lors de leur utilisation. Et bien que les avantages en matière de performances s’affichent dans les microtesteurs, nous ne pensons pas qu’ils seront impactés dans la grande majorité des utilisations réelles. S’il s’avère que c’est le cas, nous pouvons introduire un deuxième ensemble d’interfaces de manière claire.

Options ignorées prises en compte :

- `ValueTask<bool> WaitForNextAsync(); bool TryGetNext(out T result);`: les `out` paramètres ne peuvent pas être covariants. Il y a également un faible impact (un problème avec le modèle try en général) qui entraîne probablement une barrière d’écriture d’exécution pour les résultats de type référence.

Annulation

Il existe plusieurs approches possibles pour la prise en charge de l’annulation :

1. `IAsyncEnumerable<T>`/`IAsyncEnumerator<T>` n'est pas une annulation : `CancellationToken` n'apparaît nulle part. L'annulation est obtenue en regroupant logiquement dans `CancellationToken` l'énumérateur et l'énumérable de la manière appropriée, par exemple, lors de l'appel d'un itérateur, passant le `CancellationToken` en tant qu'argument à la méthode d'itérateur et en l'utilisant dans le corps de l'itérateur, comme c'est le cas avec tout autre paramètre.
2. `IAsyncEnumerator<T>.GetAsyncEnumerator(CancellationToken)`: Vous transmettez un `CancellationToken` à `GetAsyncEnumerator` , et les `MoveNextAsync` opérations suivantes le respectent toutefois.
3. `IAsyncEnumerator<T>.MoveNextAsync(CancellationToken)`: Vous transmettez un `CancellationToken` à chaque `MoveNextAsync` appel individuel.
4. 1 && 2 : incorporer des `CancellationToken` s dans votre énumérable/énumérateur et passer des `CancellationToken` s dans `GetAsyncEnumerator` .
5. 1 && 3 : incorporez des `CancellationToken` s dans votre énumérable/énumérateur et transmettez-les `CancellationToken` dans `MoveNextAsync` .

D'un point de vue purement théorique, (5) est la plus robuste, en `MoveNextAsync` acceptant un qui `CancellationToken` permet le contrôle le plus affiné sur ce qui est annulé, et (b) `CancellationToken` est tout autre type qui peut être passé en tant qu'argument à des itérateurs, incorporés dans des types arbitraires, etc.

Toutefois, il existe plusieurs problèmes avec cette approche :

- Comment une `CancellationToken` passe-t `GetAsyncEnumerator` -elle dans le corps de l'itérateur ? Nous pourrions exposer un nouveau `iterator` mot clé que vous pourriez déconnecter pour accéder au passé à `CancellationToken` `GetEnumerator` , mais un) qui est un grand nombre de machines supplémentaires, b), c'est un citoyen de premier ordre et c) le cas de 99% semble être le même code appelant un itérateur et appelant `GetAsyncEnumerator` dessus, auquel cas il peut simplement passer le `CancellationToken` comme argument dans la méthode de la méthode.
- Comment un passe-t-il `CancellationToken` pour `MoveNextAsync` accéder au corps de la méthode ? Cela est encore pire, comme s'il s'agissait d'un `iterator` objet local, sa valeur peut être modifiée dans await, ce qui signifie que tout code inscrit avec le jeton doit annuler son inscription avant await, puis se réinscrire après. il est également potentiellement coûteux de procéder à l'inscription et à l'annulation de l'inscription dans chaque `MoveNextAsync` appel, qu'il soit implémenté par le compilateur dans un itérateur ou par un développeur manuellement.
- Comment un développeur annule-t- `foreach` il une boucle ? Si c'est le cas en donnant un `CancellationToken` à un énumérable/énumérateur, puis à un (e)), nous devons prendre en charge `foreach` les énumérateurs de type « ING », ce qui leur

permet d'être des citoyens de première classe. et maintenant, vous devez commencer à réfléchir à un écosystème développé autour d'énumérateurs (par exemple, des méthodes Linq) ou b) nous devons tout d'abord incorporer `CancellationToken` dans l'énumérable tout en ayant une `WithCancellation` méthode d'extension de `IAsyncEnumerable<T>` qui stockerait le jeton fourni, puis le passera dans les énumérables encapsulés `GetAsyncEnumerator` lorsque le `GetAsyncEnumerator` sur le struct retourné est appelé (en ignorant ce jeton). Ou bien, vous pouvez simplement utiliser le `CancellationToken` dont vous disposez dans le corps de l'instruction foreach.
- Si/lorsque les interprétations des requêtes sont prises en charge, comment le a-t-il été `CancellationToken` fourni à `GetEnumerator` ou `MoveNextAsync` passé dans chaque clause ? La méthode la plus simple consisterait simplement à ce que la clause la capture, à laquelle le jeton passé à `GetAsyncEnumerator` / `MoveNextAsync` est ignoré.

Une version antérieure de ce document est recommandée (1), mais nous avons passé à (4).

Les deux principaux problèmes avec (1) :

- les producteurs d'énumérables annulables doivent implémenter une certaine utilisation et peuvent uniquement tirer parti de la prise en charge par le compilateur des itérateurs Async pour implémenter une `IAsyncEnumerator<T> GetAsyncEnumerator(CancellationToken)` méthode.
- Il est probable que de nombreux producteurs soient tentés d'ajouter simplement un `CancellationToken` paramètre à leur signature Async-Enumerable à la place, ce qui empêchera les consommateurs de transmettre le jeton d'annulation souhaité lorsqu'ils reçoivent un `IAsyncEnumerable` type.

Il existe deux scénarios de consommation principaux :

1. `await foreach (var i in GetData(token)) ...` où le consommateur appelle la méthode Async-Iterator,
2. `await foreach (var i in givenIAsyncEnumerable.WithCancellation(token)) ...` où le consommateur traite une instance donnée `IAsyncEnumerable` .

Nous constatons qu'une compromission raisonnable de la prise en charge des deux scénarios de manière pratique pour les producteurs et les consommateurs de Async-Streams consiste à utiliser un paramètre spécialement annoté dans la méthode Async-Iterator. L' `[EnumeratorCancellation]` attribut est utilisé à cet effet. Le fait de placer cet attribut sur un paramètre indique au compilateur que, si un jeton est passé à la

`GetAsyncEnumerator` méthode, ce jeton doit être utilisé à la place de la valeur transmise à l'origine pour le paramètre.

Prenez le cas de `IAsyncEnumerable<int> GetData([EnumeratorCancellation] CancellationToken token = default)`. L'implémenteur de cette méthode peut simplement utiliser le paramètre dans le corps de la méthode. Le consommateur peut utiliser les modèles de consommation ci-dessus :

1. Si vous utilisez `GetData(token)` , le jeton est enregistré dans Async-Enumerable et sera utilisé dans l'itération.
2. Si vous utilisez `givenIAsyncEnumerable.WithCancellation(token)` , le jeton passé à `GetAsyncEnumerator` remplace tout jeton enregistré dans l'énumérable Async.

foreach

`foreach` sera augmenté pour la prise en charge `IAsyncEnumerable<T>` en plus de la prise en charge existante de `IEnumerable<T>` . Et il prend en charge l'équivalent de `IAsyncEnumerable<T>` comme modèle si les membres pertinents sont exposés publiquement, en revenant à l'utilisation de l'interface directement dans le cas contraire, afin d'activer les extensions basées sur un struct qui évitent d'allouer et d'utiliser d'autres awaitables comme type de retour de `MoveNextAsync` et `DisposeAsync` .

Syntax

suivant la syntaxe : .

```
C#

foreach (var i in enumerable)
```

C# continuera à traiter `enumerable` en tant qu'énumérable synchrone, de sorte que même s'il expose les API pertinentes pour les énumérables Async (exposant le modèle ou implémentant l'interface), il prend uniquement en compte les API synchrones.

Pour forcer `foreach` à ne prendre en compte que les API asynchrones, `await` est inséré comme suit :

```
C#

await foreach (var i in enumerable)
```

Aucune syntaxe n'est fournie pour prendre en charge l'utilisation des API Async ou Sync ; le développeur doit choisir en fonction de la syntaxe utilisée.

Options ignorées prises en compte :

- *`foreach (var i in await enumerable)`*: Il s'agit déjà d'une syntaxe valide, et la modification de sa signification serait une modification avec rupture. Cela signifie que vous pouvez `await enumerable` récupérer un événement pouvant être iteré de façon synchrone à partir de celui-ci, puis effectuer une itération synchrone.
- *`foreach (var i await in enumerable)`* , *`foreach (var await i in enumerable)`* , *`foreach (await var i in enumerable)`*: Ils suggèrent que nous attendons l'élément suivant, mais il existe d'autres await impliqués dans foreach, en particulier si le Enumerable est un `IAsyncDisposable` , nous allons procéder à `await` sa suppression asynchrone. Cette expression await correspond à la portée de foreach plutôt qu'à chaque élément individuel. par conséquent, le `await` mot clé mérite d'être au `foreach` niveau. En outre, le fait de l'associer à la `foreach` nous donne un moyen de décrire le `foreach` avec un terme différent, par exemple, « await foreach ». Mais plus important encore, il est important de tenir compte de `foreach` la syntaxe en même temps que la `using` syntaxe, afin qu'ils restent cohérents les uns avec les autres et qu'il s'agisse déjà d'une `using (await ...)` syntaxe valide.
- *`foreach await (var i in enumerable)`*

Toutefois, tenez compte des éléments suivants :

- `foreach` aujourd'hui ne prend pas en charge l'itération au sein d'un énumérateur. Nous pensons qu'il est plus courant d'avoir des « `IAsyncEnumerator<T>` s », et qu'il est donc tentant de prendre en charge `await foreach` avec `IAsyncEnumerable<T>` et `IAsyncEnumerator<T>` . Toutefois, une fois que nous avons ajouté une telle prise en charge, il est question de savoir si `IAsyncEnumerator<T>` est un citoyen de première classe, et si nous devons avoir des surcharges de combinateurs qui fonctionnent sur des énumérateurs en plus des énumérables. Souhaitez-vous encourager les méthodes pour retourner des énumérateurs plutôt que des énumérables ? Nous devons continuer à aborder ce sujet. Si nous décidons de ne pas la prendre en charge, nous pourrions souhaiter introduire une méthode d'extension `public static IAsyncEnumerable<T> AsEnumerable<T>(this IAsyncEnumerator<T> enumerator);` qui permettrait à un énumérateur d'avoir toujours la valeur `foreach` . Si nous décidons de le prendre en charge, nous devrons également décider si le `await foreach` serait responsable de l'appel `DisposeAsync` sur l'énumérateur et que la réponse est probablement « non, le contrôle de la suppression doit être géré par la personne appelée `GetEnumerator` . »

Compilation basée sur des modèles

Le compilateur crée une liaison avec les API basées sur un modèle s'ils existent, en préférant ceux qui utilisent l'interface (le modèle peut être satisfait avec des méthodes d'instance ou des méthodes d'extension). La configuration requise pour le modèle est la suivante :

- L'énumérable doit exposer une `GetAsyncEnumerator` méthode qui peut être appelée sans argument et qui retourne un énumérateur qui répond au modèle approprié.
- L'énumérateur doit exposer une `MoveNextAsync` méthode qui peut être appelée sans argument et qui retourne une valeur qui peut être `await` Ed et dont `GetResult()` le retourne un `bool` .
- L'énumérateur doit également exposer la `Current` propriété dont l'accesseur Get retourne un `T` représentant le type de données énumérées.
- L'énumérateur peut éventuellement exposer une `DisposeAsync` méthode qui peut être appelée sans argument et qui retourne un objet qui peut être `await` Ed et dont le `GetResult()` retourne `void` .

Ce code :

C#

```
var enumerable = ...;
await foreach (T item in enumerable)
{
   ...
}
```

est traduit en l'équivalent de :

C#

```
var enumerable = ...;
var enumerator = enumerable.GetAsyncEnumerator();
try
{
    while (await enumerator.MoveNextAsync())
    {
       T item = enumerator.Current;
       ...
    }
}
finally
{
    await enumerator.DisposeAsync(); // omitted, along with the try/finally,
```

```
if the enumerator doesn't expose DisposeAsync
}
```

Si le type itéré n'expose pas le modèle approprié, les interfaces sont utilisées.

ConfigureAwait

Cette compilation basée sur des modèles autorisera `ConfigureAwait` à être utilisée sur toutes les await, par le biais d'une `ConfigureAwait` méthode d'extension :

C#

```
await foreach (T item in enumerable.ConfigureAwait(false))
{
   ...
}
```

Cela sera basé sur les types que nous ajouterons également à .NET, susceptibles de System.Threading.Tasks.Extensions.dll :

C#

```
// Approximate implementation, omitting arg validation and the like
namespace System.Threading.Tasks
{
    public static class AsyncEnumerableExtensions
    {
        public static ConfiguredAsyncEnumerable<T> ConfigureAwait<T>(this
IAsyncEnumerable<T> enumerable, bool continueOnCapturedContext) =>
            new ConfiguredAsyncEnumerable<T>(enumerable,
continueOnCapturedContext);

        public struct ConfiguredAsyncEnumerable<T>
        {
            private readonly IAsyncEnumerable<T> _enumerable;
            private readonly bool _continueOnCapturedContext;

            internal ConfiguredAsyncEnumerable(IAsyncEnumerable<T>
enumerable, bool continueOnCapturedContext)
            {
                _enumerable = enumerable;
                _continueOnCapturedContext = continueOnCapturedContext;
            }

            public ConfiguredAsyncEnumerator<T> GetAsyncEnumerator() =>
                new ConfiguredAsyncEnumerator<T>
(_enumerable.GetAsyncEnumerator(), _continueOnCapturedContext);

            public struct Enumerator
```

```
            {
                private readonly IAsyncEnumerator<T> _enumerator;
                private readonly bool _continueOnCapturedContext;

                internal Enumerator(IAsyncEnumerator<T> enumerator, bool
continueOnCapturedContext)
                {
                    _enumerator = enumerator;
                    _continueOnCapturedContext = continueOnCapturedContext;
                }

                public ConfiguredValueTaskAwaitable<bool> MoveNextAsync() =>

_enumerator.MoveNextAsync().ConfigureAwait(_continueOnCapturedContext);

                public T Current => _enumerator.Current;

                public ConfiguredValueTaskAwaitable DisposeAsync() =>

_enumerator.DisposeAsync().ConfigureAwait(_continueOnCapturedContext);
            }
        }
    }
}
```

Notez que cette approche ne permet pas `ConfigureAwait` d'être utilisée avec les énumérables basés sur des modèles. mais là encore, il est déjà possible que le `ConfigureAwait` soit uniquement exposé en tant qu'extension sur `Task` / `Task<T>` / `ValueTask` / `ValueTask<T>` et ne puisse pas être appliqué à des choses arbitraires, car il n'a de sens que lorsqu'il est appliqué aux tâches (il contrôle un comportement implémenté dans la prise en charge de continuation de la tâche) Toute personne qui retourne des éléments await peut fournir son propre comportement personnalisé dans de tels scénarios avancés.

(Si nous pouvons trouver un moyen de prendre en charge une solution au niveau de l'étendue ou de l'assembly `ConfigureAwait` , cela ne sera pas nécessaire.)

Itérateurs Async

Le langage/compilateur prendra en charge la production `IAsyncEnumerable<T>` `IAsyncEnumerator<T>` de et des s en plus de les consommer. Aujourd'hui, le langage prend en charge l'écriture d'un itérateur comme :

C#

```
static IEnumerable<int> MyIterator()
{
```

```
    try
    {
        for (int i = 0; i < 100; i++)
        {
            Thread.Sleep(1000);
            yield return i;
        }
    }
    finally
    {
        Thread.Sleep(200);
        Console.WriteLine("finally");
    }
}
```

mais `await` ne peut pas être utilisé dans le corps de ces itérateurs. Nous allons ajouter cette prise en charge.

Syntax

La prise en charge du langage existant pour les itérateurs déduit la nature de l'itérateur de la méthode selon qu'elle contient des `yield` s. Il en va de même pour les itérateurs Async. Ces itérateurs Async sont délimitées et différenciés des itérateurs synchrones via `async` l'ajout à la signature, et doivent alors également avoir `IAsyncEnumerable<T>` ou `IAsyncEnumerator<T>` comme type de retour. Par exemple, l'exemple ci-dessus peut être écrit comme un itérateur Async comme suit :

C#

```
static async IAsyncEnumerable<int> MyIterator()
{
    try
    {
        for (int i = 0; i < 100; i++)
        {
            await Task.Delay(1000);
            yield return i;
        }
    }
    finally
    {
        await Task.Delay(200);
        Console.WriteLine("finally");
    }
}
```

Alternatives prises en compte :

- *N'utilisez pas* `async` *dans la signature*: l'utilisation de `async` est probablement requise techniquement par le compilateur, car il l'utilise pour déterminer si `await` est valide dans ce contexte. Mais même si ce n'est pas obligatoire, nous avons établi que ne `await` peut être utilisé que dans les méthodes marquées comme `async` , et il semble important de conserver la cohérence.
- *Activation des générateurs personnalisés* `IAsyncEnumerable<T>` *pour*: c'est un aspect à venir, mais la machine est compliquée et nous ne prenons pas en charge cette opération pour les homologues synchrones.
- *Présence d'un* `iterator` *mot clé dans la signature : les* itérateurs Async utilisent `async iterator` dans la signature et `yield` peuvent uniquement être utilisés dans les `async` méthodes incluses `iterator` ; ils `iterator` sont ensuite rendus facultatifs sur les itérateurs synchrones. En fonction de votre perspective, cela présente l'avantage d'être très clair par la signature de la méthode, si elle est `yield` autorisée et si la méthode est en fait supposée retourner des instances de type `IAsyncEnumerable<T>` plutôt que le compilateur en fabrication, selon que le code utilise `yield` ou non. Mais il est différent des itérateurs synchrones, ce qui n'est pas et ne peut pas être fait pour en exiger un. En outre, certains développeurs n'aiment pas la syntaxe supplémentaire. Si nous étions en train de le concevoir à partir de zéro, nous aurions probablement effectué cette opération, mais à ce stade, il y a bien plus de valeur pour maintenir les itérateurs Async près des itérateurs de synchronisation.

LINQ

Il y a plus de ~ 200 surcharges de méthodes sur la `System.Linq.Enumerable` classe, qui fonctionnent toutes en termes de `IEnumerable<T>` ; certaines d'entre elles sont `IEnumerable<T>` générées, `IEnumerable<T>` et bien plus encore. L'ajout de la prise en charge de LINQ pour entraînerait `IAsyncEnumerable<T>` probablement la duplication de toutes ces surcharges pour celle-ci, pour une autre ~ 200. Et étant donné que `IAsyncEnumerator<T>` est susceptible d'être plus courant comme une entité autonome dans le monde asynchrone que `IEnumerator<T>` dans le monde synchrone, nous pourrions potentiellement avoir besoin d'une autre surcharges ~ 200 qui fonctionnent avec `IAsyncEnumerator<T>` . En outre, un grand nombre de surcharges traitent des prédicats (par exemple, `Where` qui prend un `Func<T, bool>`) et il peut être souhaitable d'avoir des `IAsyncEnumerable<T>` surcharges basées sur des prédicats synchrones et asynchrones (par exemple, `Func<T, ValueTask<bool>>` en plus de `Func<T, bool>`). Bien que cela ne s'applique pas à toutes les nouvelles surcharges de plus de 400, un calcul approximatif est qu'elle s'applique à la moitié, ce qui signifie une autre ~ 200 surcharges, pour un total de ~ 600 nouvelles méthodes.

Il s'agit d'un nombre d'API en quinconce, avec un potentiel encore plus possible lorsque des bibliothèques d'extensions telles que les extensions interactives (IX) sont prises en compte. Mais IX a déjà une implémentation de beaucoup d'entre elles, et il n'existe pas de bonne raison de dupliquer ce travail. Nous devons à la place aider la communauté à améliorer IX et la recommander quand les développeurs souhaitent utiliser LINQ avec `IAsyncEnumerable<T>` .

Il y a également le problème de syntaxe d'interprétation des requêtes. La nature basée sur les modèles des interprétations des requêtes leur permettrait de « travailler » avec certains opérateurs, par exemple, si IX fournit les méthodes suivantes :

```C#
public static IAsyncEnumerable<TResult> Select<TSource, TResult>(this
IAsyncEnumerable<TSource> source, Func<TSource, TResult> func);
public static IAsyncEnumerable<T> Where(this IAsyncEnumerable<T> source,
Func<T, bool> func);
```

Ce code C# fonctionnera alors « tout simplement » :

```C#
IAsyncEnumerable<int> enumerable = ...;
IAsyncEnumerable<int> result = from item in enumerable
                               where item % 2 == 0
                               select item * 2;
```

Toutefois, il n'existe aucune syntaxe d'interprétation des requêtes qui prend en charge l'utilisation `await` de dans les clauses, donc si IX a été ajouté, par exemple :

```C#
public static IAsyncEnumerable<TResult> Select<TSource, TResult>(this
IAsyncEnumerable<TSource> source, Func<TSource, ValueTask<TResult>> func);
```

Cela ne fonctionnerait alors que :

```C#
IAsyncEnumerable<string> result = from url in urls
                                  where item % 2 == 0
                                  select SomeAsyncMethod(item);

async ValueTask<int> SomeAsyncMethod(int item)
{
    await Task.Yield();
```

```
    return item * 2;
}
```

mais il n'y avait aucun moyen de l'écrire avec l' `await` inline dans la `select` clause. Dans le cadre d'un travail distinct, nous pourrions envisager d'ajouter des `async { ... }` expressions au langage. à ce stade, nous pourrions les autoriser à être utilisés dans les interprétations des requêtes et, à la place, il serait possible de les écrire comme suit :

C#

```
IAsyncEnumerable<int> result = from item in enumerable
                               where item % 2 == 0
                               select async
                               {
                                   await Task.Yield();
                                   return item * 2;
                               };
```

ou pour permettre l' `await` utilisation directe de dans les expressions, par exemple en prenant en charge `async from` . Toutefois, il est peu probable qu'une conception ici ait un impact sur le reste de l'ensemble de fonctionnalités, ou sur l'autre, et ce n'est pas une chose particulièrement importante pour investir à l'heure actuelle. par conséquent, la proposition consiste à ne rien faire d'autre ici pour le moment.

Intégration à d'autres frameworks asynchrones

L'intégration avec `IObservable<T>` et d'autres frameworks asynchrones (par exemple, les flux réactifs) est effectuée au niveau de la bibliothèque plutôt qu'au niveau de la langue. Par exemple, toutes les données d'un `IAsyncEnumerator<T>` peuvent être publiées dans un `IObserver<T>` tout simplement par `await foreach` le biais de l'énumérateur et des `OnNext` données vers l'observateur `AsObservable<T>` . une méthode d'extension est donc possible. L'utilisation d'un `IObservable<T>` dans un `await foreach` nécessite la mise en mémoire tampon des données (si un autre élément est envoyé alors que l'élément précédent est toujours en cours de traitement), mais un tel adaptateur push-pull peut facilement être implémenté pour permettre l'extraction d'un objet à `IObservable<T>` partir de avec un `IAsyncEnumerator<T>` . Suite. RX/IX fournit déjà des prototypes de telles implémentations, et les bibliothèques telles que https://github.com/dotnet/corefx/tree/master/src/System.Threading.Channels fournissent différents types de structures de données de mise en mémoire tampon. La langue n'est pas nécessaire à ce niveau.

Plages

Article • 17/06/2021

Récapitulatif

Cette fonctionnalité consiste à offrir deux nouveaux opérateurs qui permettent de construire `System.Index` des `System.Range` objets et, et à les utiliser pour indexer/découper des collections au moment de l'exécution.

Vue d'ensemble

Types et membres connus

Pour utiliser les nouvelles formes syntaxiques pour `System.Index` et `System.Range` , de nouveaux types et membres connus peuvent être nécessaires, en fonction des formes syntaxiques utilisées.

Pour utiliser l'opérateur « Hat » (`^`), les conditions suivantes sont requises :

C#

```
namespace System
{
    public readonly struct Index
    {
        public Index(int value, bool fromEnd);
    }
}
```

Pour utiliser le `System.Index` type en tant qu'argument dans un accès à un élément de tableau, le membre suivant est requis :

C#

```
int System.Index.GetOffset(int length);
```

La `..` syntaxe de `System.Range` requiert le `System.Range` type, ainsi qu'un ou plusieurs des membres suivants :

C#

```
namespace System
{
    public readonly struct Range
    {
        public Range(System.Index start, System.Index end);
        public static Range StartAt(System.Index start);
        public static Range EndAt(System.Index end);
        public static Range All { get; }
    }
}
```

La `..` syntaxe permet d'absenter l'un ou l'autre, ou aucun de ses arguments. Quel que soit le nombre d'arguments, le `Range` constructeur est toujours suffisant pour utiliser la `Range` syntaxe. Toutefois, si l'un des autres membres est présent et qu'un ou plusieurs `..` arguments manquent, le membre approprié peut être substitué.

Enfin, pour une valeur de type `System.Range` à utiliser dans une expression d'accès à un élément de tableau, le membre suivant doit être présent :

C#

```
namespace System.Runtime.CompilerServices
{
    public static class RuntimeHelpers
    {
        public static T[] GetSubArray<T>(T[] array, System.Range range);
    }
}
```

System. index

C# n'a aucun moyen d'indexer une collection à partir de la fin, mais la plupart des indexeurs utilisent la notion « from Start » ou une expression « length-i ». Nous introduisons une nouvelle expression d'index qui signifie « à partir de la fin ». La fonctionnalité introduira un nouvel opérateur « Hat » de préfixe unaire. Son opérande unique doit pouvoir être converti en `System.Int32` . Il sera abaissé dans l' `System.Index` appel de méthode de fabrique approprié.

Nous enrichissons la grammaire de *unary_expression* avec la forme de syntaxe supplémentaire suivante :

antlr

```
unary_expression
    : '^' unary_expression
```

```
    ;
```

Nous appelons l' *index à partir de* l'opérateur end. L' *index prédéfini à partir des opérateurs end* est le suivant :

```
C#

System.Index operator ^(int fromEnd);
```

Le comportement de cet opérateur est défini uniquement pour les valeurs d'entrée supérieures ou égales à zéro.

Exemples :

```
C#

var array = new int[] { 1, 2, 3, 4, 5 };
var thirdItem = array[2];    // array[2]
var lastItem = array[^1];    // array[new Index(1, fromEnd: true)]
```

System. Range

C# n'offre aucun moyen syntaxique d'accéder aux « plages » ou aux « tranches » de collections. Généralement, les utilisateurs sont obligés d'implémenter des structures complexes pour filtrer/faire fonctionner sur des secteurs de mémoire, ou recourir à des méthodes LINQ comme `list.Skip(5).Take(2)` . Avec l'ajout de `System.Span<T>` et d'autres types similaires, il est plus important d'avoir ce type d'opération pris en charge sur un niveau plus profond dans le langage/l'exécution et de faire en sorte que l'interface soit unifiée.

Le langage introduira un nouvel opérateur de plage `x..y` . Il s'agit d'un opérateur d'infixe binaire qui accepte deux expressions. L'un des opérandes peut être omis (exemples ci-dessous) et doit être convertible en `System.Index` . Il sera abaissé à l' `System.Range` appel de méthode de fabrique approprié.

Nous remplaçons les règles de grammaire C# pour *multiplicative_expression* par les éléments suivants (afin d'introduire un nouveau niveau de priorité) :

```
antlr

range_expression
    : unary_expression
    | range_expression? '..' range_expression?
    ;
```

```
multiplicative_expression
    : range_expression
    | multiplicative_expression '*' range_expression
    | multiplicative_expression '/' range_expression
    | multiplicative_expression '%' range_expression
    ;
```

Toutes les formes de l' *opérateur de plage* ont la même priorité. Ce nouveau groupe de précédence est inférieur aux *opérateurs unaires* et supérieur aux *opérateurs arithmétiques de multiplication*.

Nous appelons l' `..` opérateur *Range*. L'opérateur de plage intégré peut être compris approximativement pour correspondre à l'appel d'un opérateur intégré de ce formulaire :

C#

```
System.Range operator ..(Index start = 0, Index end = ^0);
```

Exemples :

C#

```
var array = new int[] { 1, 2, 3, 4, 5 };
var slice1 = array[2..^3];    // array[new Range(2, new Index(3, fromEnd:
true))]
var slice2 = array[..^3];     // array[Range.EndAt(new Index(3, fromEnd:
true))]
var slice3 = array[2..];      // array[Range.StartAt(2)]
var slice4 = array[..];       // array[Range.All]
```

En outre, `System.Index` doit avoir une conversion implicite de `System.Int32` , afin d'éviter la nécessité de surcharger le mélange des entiers et des index sur des signatures multidimensionnelles.

Ajout de la prise en charge des index et des plages aux types de bibliothèques existants

Prise en charge d'index implicite

Le langage fournira un membre d'indexeur d'instance avec un paramètre unique de type `Index` pour les types qui répondent aux critères suivants :

- Le type est NBVAL.
- Le type a un indexeur d'instance accessible qui accepte un seul `int` comme argument.
- Le type n'a pas d'indexeur d'instance accessible qui prend `Index` comme premier paramètre. `Index`Doit être le seul paramètre ou les paramètres restants doivent être facultatifs.

Un type est ***compté*** s'il a une propriété nommée `Length` ou `Count` avec un accesseur Get accessible et un type de retour `int` . Le langage peut utiliser cette propriété pour convertir une expression de type `Index` en un `int` au point de l'expression sans qu'il soit nécessaire d'utiliser le type `Index` . Si `Length` et sont tous deux `Count` présents, `Length` sera préféré. Par souci de simplicité, la proposition utilise le nom `Length` pour représenter `Count` ou `Length` .

Pour ces types, le langage agit comme s'il s'agissait d'un membre d'indexeur du formulaire `T this[Index index]` où `T` est le type de retour de l' `int` indexeur de base, y compris les `ref` Annotations de style. Le nouveau membre aura les mêmes `get` membres et `set` avec l'accessibilité correspondante que l' `int` indexeur.

Le nouvel indexeur sera implémenté en convertissant l'argument de type `Index` en un `int` et en émettant un appel à l' `int` indexeur de base. À des fins de discussion, utilisons l'exemple de `receiver[expr]` . La conversion de `expr` en `int` se produit comme suit :

- Lorsque l'argument se présente sous la forme `^expr2` et que le type de `expr2` est `int` , il est converti en `receiver.Length - expr2` .
- Dans le cas contraire, elle sera traduite par `expr.GetOffset(receiver.Length)` .

Cela permet aux développeurs d'utiliser la `Index` fonctionnalité sur les types existants sans nécessiter de modification. Par exemple :

C#

```
List<char> list = ...;
var value = list[^1];

// Gets translated to
var value = list[list.Count - 1];
```

Les expressions et sont débordées `receiver` `Length` de manière appropriée pour garantir que tous les effets secondaires ne sont exécutés qu'une seule fois. Par exemple :

C#

```
class Collection {
    private int[] _array = new[] { 1, 2, 3 };

    public int Length {
        get {
            Console.Write("Length ");
            return _array.Length;
        }
    }

    public int this[int index] => _array[index];
}

class SideEffect {
    Collection Get() {
        Console.Write("Get ");
        return new Collection();
    }

    void Use() {
        int i = Get()[^1];
        Console.WriteLine(i);
    }
}
```

Ce code affiche « obtient la longueur 3 ».

Prise en charge des plages implicites

Le langage fournira un membre d'indexeur d'instance avec un paramètre unique de type `Range` pour les types qui répondent aux critères suivants :

- Le type est NBVAL.
- Le type a un membre accessible nommé `Slice` qui a deux paramètres de type `int` .
- Le type n'a pas d'indexeur d'instance qui accepte un seul `Range` comme premier paramètre. `Range`Doit être le seul paramètre ou les paramètres restants doivent être facultatifs.

Pour ces types, le langage est lié comme s'il s'agissait d'un membre d'indexeur de la forme `T this[Range range]` où `T` est le type de retour de la `Slice` méthode, y compris les `ref` Annotations de style. Le nouveau membre aura également une accessibilité correspondante avec `Slice` .

Lorsque l' `Range` indexeur de base est lié à une expression nommée `receiver` , il est abaissé en convertissant l' `Range` expression en deux valeurs qui sont ensuite passées à la `Slice` méthode. À des fins de discussion, utilisons l'exemple de `receiver[expr]` .

Le premier argument de `Slice` sera obtenu en convertissant l'expression de plage typée de la manière suivante :

- Lorsque a `expr` la forme `expr1..expr2` (où `expr2` peut être omis) et a le `expr1` type `int` , il est émis en tant que `expr1` .
- Lorsque `expr` a la forme `^expr1..expr2` (où `expr2` peut être omis), il est émis en tant que `receiver.Length - expr1` .
- Lorsque `expr` a la forme `..expr2` (où `expr2` peut être omis), il est émis en tant que `0` .
- Dans le cas contraire, il sera émis sous la forme `expr.Start.GetOffset(receiver.Length)` .

Cette valeur sera réutilisée dans le calcul du deuxième `Slice` argument. Dans ce cas, il est appelé `start` . Le deuxième argument de `Slice` sera obtenu en convertissant l'expression de plage typée de la manière suivante :

- Lorsque a `expr` la forme `expr1..expr2` (où `expr1` peut être omis) et a le `expr2` type `int` , il est émis en tant que `expr2 - start` .
- Lorsque `expr` a la forme `expr1..^expr2` (où `expr1` peut être omis), il est émis en tant que `(receiver.Length - expr2) - start` .
- Lorsque `expr` a la forme `expr1..` (où `expr1` peut être omis), il est émis en tant que `receiver.Length - start` .
- Dans le cas contraire, il sera émis sous la forme `expr.End.GetOffset(receiver.Length) - start` .

Les `receiver` expressions, et sont débordées `Length` `expr` de manière appropriée pour garantir que tous les effets secondaires ne sont exécutés qu'une seule fois. Par exemple :

C#

```
class Collection {
    private int[] _array = new[] { 1, 2, 3 };

    public int Length {
        get {
            Console.Write("Length ");
            return _array.Length;
        }
    }
```

```
    public int[] Slice(int start, int length) {
        var slice = new int[length];
        Array.Copy(_array, start, slice, 0, length);
        return slice;
    }
}

class SideEffect {
    Collection Get() {
        Console.Write("Get ");
        return new Collection();
    }

    void Use() {
        var array = Get()[0..2];
        Console.WriteLine(array.Length);
    }
}
```

Ce code affiche « obtient la longueur 2 ».

Le langage respecte le cas particulier des types connus suivants :

- `string`: la méthode `Substring` sera utilisée à la place de `Slice` .
- `array`: la méthode `System.Runtime.CompilerServices.RuntimeHelpers.GetSubArray` sera utilisée à la place de `Slice` .

Autres solutions

Les nouveaux opérateurs (`^` et `..`) sont des sucres syntaxiques. La fonctionnalité peut être implémentée par des appels explicites à `System.Index` et des `System.Range` méthodes de fabrique, mais elle génère beaucoup plus de code réutilisable, et l'expérience n'est pas intuitive.

Représentation IL

Ces deux opérateurs seront abaissés aux appels d'indexeur/de méthode normaux, sans modification des couches de compilateur suivantes.

Comportement au moment de l'exécution

- Le compilateur peut optimiser des indexeurs pour des types intégrés tels que des tableaux et des chaînes, et réduire l'indexation aux méthodes existantes appropriées.

- `System.Index` lève une exception si elle est construite avec une valeur négative.
- `^0` ne lève pas, mais convertit la longueur de la collection/énumérable à laquelle il est fourni.
- `Range.All` est sémantiquement équivalent à `0..^0` , et peut être déconstruite sur ces index.

Considérations

Détecter les indexables basés sur ICollection

L'inspiration de ce comportement était des initialiseurs de collection. Utilisation de la structure d'un type pour indiquer qu'il a opté pour une fonctionnalité. Dans le cas des initialiseurs de collection, les types peuvent accepter la fonctionnalité en implémentant l'interface `IEnumerable` (non générique).

Cette proposition nécessitait initialement que les types implémentent `ICollection` pour qualifier comme indexable. Cela nécessitait un certain nombre de cas particuliers :

- `ref struct`: celles-ci ne peuvent pas implémenter `Span<T>` d'interfaces, mais sont idéales pour la prise en charge des index/plages.
- `string`: n'implémente pas `ICollection` et n'ajoute pas qui `interface` a un coût élevé.

Cela signifie que la prise en charge de la casse spéciale des types de clés est déjà nécessaire. La casse spéciale de `string` est moins intéressante, car le langage le fait dans d'autres zones (`foreach` abaissement, constantes, etc.). La casse spéciale de `ref struct` est davantage en ce qui concerne la casse particulière d'une classe entière de types. Ils sont étiquetés comme indexables s'ils ont simplement une propriété nommée `Count` avec un type de retour `int` .

Après considération, la conception a été normalisée pour indiquer que tout type qui a une propriété `Count` / `Length` avec un type de retour `int` est indexable. Qui supprime toute la casse spéciale, même pour les `string` tableaux et.

Détecter uniquement le nombre

La détection des noms de propriété `Count` ou `Length` complique la conception. Le choix d'un seul pour normaliser n'est pas suffisant, car il finit par exclure un grand nombre de types :

- Utilisation `Length` : exclut à peu près chaque collection dans System. Collections et les sous-espaces de noms. Celles-ci ont tendance à dériver de `ICollection` et, par conséquent, préfèrent dépasser la `Count` longueur.
- Utilisation `Count` : exclut `string` , `Span<T>` les tableaux et les types les plus `ref struct` basés

La plus grande complication de la détection initiale de types indexables est dépensée par sa simplification dans d'autres aspects.

Choix d'un secteur en tant que nom

Le nom `Slice` a été choisi, car il s'agit du nom standard de facto pour les opérations de style de découpage dans .net. À compter de netcoreapp 2.1, tous les types de style d'étendue utilisent le nom `Slice` pour les opérations de découpage. Avant netcoreapp 2.1, il n'y a pas vraiment d'exemples de découpage à rechercher pour obtenir un exemple. Les types tels que `List<T>` , `ArraySegment<T>` , `SortedList<T>` seraient idéaux pour le découpage, mais le concept n'existait pas lors de l'ajout de types.

Par conséquent, `Slice` en tant que seul exemple, il a été choisi comme nom.

Conversion du type de cible d'index

Une autre façon d'afficher la `Index` transformation dans une expression d'indexeur est de convertir un type cible. Au lieu de créer une liaison comme s'il s'agissait d'un membre du formulaire `return_type this[Index]` , le langage assigne à la place une conversion de cible typée en `int` .

Ce concept peut être généralisé à l'accès de tous les membres sur les types dénombrables. Chaque fois qu'une expression de type `Index` est utilisée en tant qu'argument pour un appel de membre d'instance et que le récepteur est compté, l'expression aura une conversion de type cible en `int` . Les appels de membres applicables pour cette conversion incluent les méthodes, les indexeurs, les propriétés, les méthodes d'extension, etc. Seuls les constructeurs sont exclus, car ils n'ont pas de récepteur.

La conversion de type cible sera implémentée comme suit pour toute expression qui a un type `Index` . À des fins de discussion, vous pouvez utiliser l'exemple de `receiver[expr]` :

- Quand `expr` a la forme `^expr2` et que le type de `expr2` est `int` , il est converti en `receiver.Length - expr2` .

- Dans le cas contraire, elle sera traduite par `expr.GetOffset(receiver.Length)` .

Les expressions et sont débordées `receiver` `Length` de manière appropriée pour garantir que tous les effets secondaires ne sont exécutés qu'une seule fois. Par exemple :

C#

```
class Collection {
    private int[] _array = new[] { 1, 2, 3 };

    public int Length {
        get {
            Console.Write("Length ");
            return _array.Length;
        }
    }

    public int GetAt(int index) => _array[index];
}

class SideEffect {
    Collection Get() {
        Console.Write("Get ");
        return new Collection();
    }

    void Use() {
        int i = Get().GetAt(^1);
        Console.WriteLine(i);
    }
}
```

Ce code affiche « obtient la longueur 3 ».

Cette fonctionnalité est bénéfique pour tout membre qui avait un paramètre qui représentait un index. Par exemple, `List<T>.InsertAt`. Cela présente également un risque de confusion, car le langage ne peut pas vous aider à déterminer si une expression est destinée à l'indexation. Tout ce qu'il peut faire est de convertir n'importe quelle `Index` expression en lors de l' `int` appel d'un membre sur un type compté.

Restrictions :

- Cette conversion s'applique uniquement lorsque l'expression avec le type `Index` est directement un argument du membre. Elle ne s'applique pas aux expressions imbriquées.

Décisions prises pendant l'implémentation

- Tous les membres du modèle doivent être membres d'instance
- Si une méthode de longueur est trouvée mais que son type de retour est incorrect, continuez à rechercher le nombre
- L'indexeur utilisé pour le modèle d'index doit avoir exactement un paramètre int
- La méthode Slice utilisée pour le modèle Range doit avoir exactement deux paramètres int
- Lorsque vous recherchez les membres du modèle, nous cherchons les définitions d'origine, et non les membres construits.

Réunions de conception

- 10 janvier 2018
- 18 janvier 2018
- 22 janvier, 2018
- 3 décembre 2018
- 25 mars, 2019
- 1er avril, 2019
- 15 avril 2019

« utilisation basée sur des modèles » et « utilisation des déclarations »

Article • 13/05/2021

Résumé

Le langage ajoutera deux nouvelles fonctionnalités autour de l' `using` instruction afin de simplifier la gestion des ressources : `using` doit reconnaître un modèle jetable en plus de `IDisposable` et ajouter une `using` déclaration au langage.

Motivation

L' `using` affirmation est un outil efficace pour la gestion des ressources, mais elle nécessite un peu de cérémonie. Les méthodes qui ont un certain nombre de ressources à gérer peuvent être facilement coincées dans une série d' `using` instructions. Cette charge de syntaxe est suffisante pour que la plupart des règles de style de codage aient explicitement une exception autour des accolades pour ce scénario.

La `using` déclaration supprime la plus grande partie de la cérémonie ici et obtient le langage C# en même temps que d'autres langages qui incluent des blocs de gestion des ressources. En outre, le modèle basé sur `using` des modèles permet aux développeurs de développer l'ensemble des types qui peuvent participer ici. Dans de nombreux cas, il est inutile de créer des types de wrappers qui n'existent que pour permettre l'utilisation d'une valeur dans une `using` instruction.

Ensemble, ces fonctionnalités permettent aux développeurs de simplifier et d'étendre les scénarios dans lesquels l' `using` application peut être appliquée.

Conception détaillée

déclaration using

Le langage autorisera `using` à être ajouté à une déclaration de variable locale. Une telle déclaration aura le même effet que la déclaration de la variable dans une `using` instruction au même emplacement.

```
C#
```

```
if (...)
{
   using FileStream f = new FileStream(@"C:\users\jaredpar\using.md");
   // statements
}

// Equivalent to
if (...)
{
   using (FileStream f = new FileStream(@"C:\users\jaredpar\using.md"))
   {
    // statements
   }
}
```

La durée de vie d'un `using` local s'étend jusqu'à la fin de l'étendue dans laquelle il est déclaré. Les `using` variables locales seront ensuite supprimées dans l'ordre inverse dans lequel elles sont déclarées.

C#

```
{
    using var f1 = new FileStream("...");
    using var f2 = new FileStream("..."), f3 = new FileStream("...");
    ...
    // Dispose f3
    // Dispose f2
    // Dispose f1
}
```

Il n'existe aucune restriction concernant `goto` , ou toute autre construction de workflow de contrôle en face d'une `using` déclaration. Au lieu de cela, le code agit comme pour l' `using` instruction équivalente :

C#

```
{
    using var f1 = new FileStream("...");
  target:
    using var f2 = new FileStream("...");
    if (someCondition)
    {
        // Causes f2 to be disposed but has no effect on f1
        goto target;
    }
}
```

Une variable locale déclarée dans une `using` déclaration locale est implicitement en lecture seule. Cela correspond au comportement des variables locales déclarées dans une `using` instruction.

La grammaire de la langue pour les `using` déclarations sera la suivante :

antlr

```
local-using-declaration:
  using type using-declarators

using-declarators:
  using-declarator
  using-declarators , using-declarator

using-declarator:
  identifier = expression
```

Restrictions autour de la `using` déclaration :

- Peut ne pas apparaître directement à l'intérieur d'une `case` étiquette, mais au lieu d'être dans un bloc à l'intérieur de l' `case` étiquette.
- Peut ne pas apparaître dans le cadre d'une `out` déclaration de variable.
- Doit avoir un initialiseur pour chaque déclarateur.
- Le type local doit être implicitement convertible en `IDisposable` ou rempli le `using` modèle.

utilisation basée sur des modèles

Le langage ajoutera la notion d'un modèle jetable : il s'agit d'un type qui a une `Dispose` méthode d'instance accessible. Les types qui correspondent au modèle jetable peuvent participer à une `using` instruction ou une déclaration sans qu'il soit nécessaire d'implémenter `IDisposable` .

C#

```
class Resource
{
    public void Dispose() { ... }
}

using (var r = new Resource())
{
    // statements
}
```

Cela permet aux développeurs de tirer parti de `using` plusieurs nouveaux scénarios :

- `ref struct`: Ces types ne peuvent pas implémenter d'interfaces aujourd'hui et ne peuvent donc pas participer aux `using` instructions.
- Les méthodes d'extension permettent aux développeurs d'ajouter des types dans d'autres assemblys pour participer aux `using` instructions.

Dans le cas où un type peut être implicitement converti en `IDisposable` et s'adapte également au modèle jetable, `IDisposable` est préféré. Bien que cela prenne l'approche opposée de `foreach` (modèle préféré par rapport à l'interface), elle est nécessaire à des fins de compatibilité descendante.

Les mêmes restrictions `using` qu'une instruction traditionnelle s'appliquent également ici : les variables locales déclarées dans le `using` sont en lecture seule, une `null` valeur ne provoque pas la levée d'une exception, etc. La génération de code sera différente uniquement en ce qu'il n'y aura pas de cast vers `IDisposable` avant d'appeler dispose :

C#

```
{
      Resource r = new Resource();
      try {
            // statements
      }
      finally {
            if (r != null) r.Dispose();
      }
}
```

Pour s'adapter au modèle jetable, la `Dispose` méthode doit être accessible, sans paramètre et avoir un `void` type de retour. Il n'existe aucune autre restriction. Cela signifie explicitement que les méthodes d'extension peuvent être utilisées ici.

Considérations

étiquettes case sans blocs

Un `using declaration` est illégal directement à l'intérieur d'une `case` étiquette en raison de complications autour de sa durée de vie réelle. Une solution possible consiste simplement à lui attribuer la même durée de vie que `out var` dans le même emplacement. Il a été considéré comme la complexité supplémentaire de

l’implémentation des fonctionnalités et la facilité de contournement (juste ajouter un bloc à l' `case` étiquette) n’a pas justifié la réalisation de cet itinéraire.

Expansions futures

variables locales fixes

Une `fixed` instruction possède toutes les propriétés des `using` instructions qui ont motivé la capacité à avoir des `using` variables locales. Vous devez également envisager d’étendre cette fonctionnalité aux `fixed` variables locales. Les règles de durée de vie et de classement doivent s’appliquer aussi bien pour `using` et `fixed` ici.

Fonctions locales statiques

Article • 13/05/2021

Résumé

Prend en charge les fonctions locales qui interdisent la capture de l'État à partir de la portée englobante.

Motivation

Évitez de capturer de manière involontaire un État dans le contexte englobant. Autorisez l'utilisation de fonctions locales dans les scénarios où une `static` méthode est requise.

Conception détaillée

Une fonction locale déclarée `static` ne peut pas capturer l'État à partir de la portée englobante. Par conséquent, les variables locales, les paramètres et `this` de la portée englobante ne sont pas disponibles dans une `static` fonction locale.

Une `static` fonction locale ne peut pas faire référence à des membres d'instance à partir d'une référence implicite ou explicite `this` ou `base` .

Une `static` fonction locale peut faire référence `static` à des membres de la portée englobante.

Une `static` fonction locale peut référencer `constant` des définitions de la portée englobante.

`nameof()` dans une `static` fonction locale, peut faire référence à des variables locales, des paramètres ou `this` `base` de la portée englobante.

Les règles d'accessibilité des `private` membres de la portée englobante sont les mêmes pour les `static` fonctions et non `static` locales.

Une `static` définition de fonction locale est émise comme une `static` méthode dans les métadonnées, même si elle est utilisée uniquement dans un délégué.

Une `static` fonction ou une expression lambda non locale peut capturer l'État à partir d'une `static` fonction locale englobante, mais ne peut pas capturer l'État en dehors de

la `static` fonction locale englobante.

Une `static` fonction locale ne peut pas être appelée dans une arborescence d'expressions.

Un appel à une fonction locale est émis au `call` lieu `callvirt` de, que la fonction locale soit ou non `static` .

Résolution de surcharge d'un appel dans une fonction locale non affectée par le fait que la fonction locale est `static` .

La suppression du `static` modificateur d'une fonction locale dans un programme valide ne modifie pas la signification du programme.

Réunions de conception

https://github.com/dotnet/csharplang/blob/master/meetings/2018/LDM-2018-09-10.md#static-local-functions

Attribution de fusion Null

Article • 13/05/2021

- [x] proposé
- [x] prototype : terminé
- [x] implémentation : terminée
- [x] spécification : ci-dessous

Résumé

Simplifie un modèle de codage commun dans lequel une valeur est assignée à une variable si elle est null.

Dans le cadre de cette proposition, nous relâcherons également les exigences de type sur `??` pour permettre l'utilisation d'une expression dont le type est un paramètre de type sans contrainte à utiliser sur le côté gauche.

Motivation

Il est courant de voir le code du formulaire

C#

```
if (variable == null)
{
    variable = expression;
}
```

Cette proposition ajoute un opérateur binaire non surchargeable dans le langage qui exécute cette fonction.

Il y a eu au moins huit demandes de communauté distinctes pour cette fonctionnalité.

Conception détaillée

Nous ajoutons une nouvelle forme d'opérateur d'assignation

antlr

```
assignment_operator
    : '??='
    ;
```

Qui suit les règles sémantiques existantes pour les opérateurs d'assignation composée, à l'exception du fait que nous Elide l'assignation si le côté gauche n'est pas null. Les règles de cette fonctionnalité sont les suivantes.

Donné `a ??= b` , où `A` est le type de `a` , `B` est le type de `b` , et `A0` est le type sous-jacent de `A` si `A` est un type valeur Nullable :

1. Si `A` n'existe pas ou s'il s'agit d'un type valeur qui n'autorise pas les valeurs NULL, une erreur de compilation se produit.
2. Si `B` n'est pas implicitement convertible en `A` ou `A0` (s' `A0` il existe), une erreur de compilation se produit.
3. Si `A0` existe et qu'il `B` est implicitement convertible en `A0` et `B` n'est pas dynamique, le type de `a ??= b` est `A0` . `a ??= b` est évaluée au moment de l'exécution comme suit :

   ```
   C#

   var tmp = a.GetValueOrDefault();
   if (!a.HasValue) { tmp = b; a = tmp; }
   tmp
   ```

   Sauf que `a` n'est évalué qu'une seule fois.
4. Sinon, le type de `a ??= b` est `A` . `a ??= b` est évaluée au moment de l'exécution en tant `a ?? (a = b)` que, sauf que `a` n'est évalué qu'une seule fois.

Pour l'assouplissement des exigences de type de `??` , nous mettons à jour la spécification là où elle déclare actuellement, étant donné que `a ?? b` `A` est le type de `a` :

> 1. Si un existe et n'est pas un type Nullable ou un type référence, une erreur de compilation se produit.

Nous détendons cette exigence pour :

1. Si un existe et qu'il s'agit d'un type valeur n'acceptant pas les valeurs NULL, une erreur de compilation se produit.

Cela permet à l'opérateur de fusion Null de fonctionner sur des paramètres de type sans contrainte, étant donné que le paramètre de type sans contrainte T existe, qu'il n'est pas un type Nullable et qu'il n'est pas un type référence.

Inconvénients

Comme pour toutes les fonctionnalités de langage, nous devons déterminer si la complexité supplémentaire de la langue est remboursée dans la clarté supplémentaire offerte par le corps des programmes C# qui tireraient parti de la fonctionnalité.

Autres solutions

Le programmeur peut écrire `(x = x ?? y)` , `if (x == null) x = y;` ou `x ?? (x = y)` à la main.

Questions non résolues

- [] Requiert la revue LDM
- [] Devons-nous également prendre en charge `&&=` les `||=` opérateurs et ?

Réunions de conception

Aucun.

Membres d'instance en lecture seule

Article • 13/05/2021

Problème champion : https://github.com/dotnet/csharplang/issues/1710

Récapitulatif

Fournir un moyen de spécifier des membres d'instance individuels sur un struct ne pas modifier l'État, de la même façon qu' `readonly struct` aucun membre d'instance ne modifie l'État.

Il convient de noter que `readonly instance member` ! = `pure instance member` . Un `pure` membre d'instance garantit qu'aucun État ne sera modifié. Un `readonly` membre d'instance ne garantit que l'état de l'instance ne sera pas modifié.

Tous les membres d'instance sur un `readonly struct` peuvent être considérés implicitement `readonly instance members` . `readonly instance members`Les déclarations explicites sur des structs non ReadOnly se comportent de la même manière. Par exemple, ils créent toujours des copies masquées si vous avez appelé un membre d'instance (sur l'instance actuelle ou sur un champ de l'instance) qui n'était lui-même pas en lecture seule.

Motivation

Aujourd'hui, les utilisateurs ont la possibilité de créer `readonly struct` des types que le compilateur impose que tous les champs soient ReadOnly (et, par extension, qu'aucun membre d'instance ne modifie l'État). Toutefois, il existe des scénarios où vous disposez d'une API existante qui expose des champs accessibles ou qui a une combinaison de membres mutants et non mutants. Dans ces circonstances, vous ne pouvez pas marquer le type comme `readonly` (il s'agit d'une modification avec rupture).

Cela n'a généralement pas d'impact, sauf dans le cas des `in` paramètres. Avec `in` les paramètres des structs non ReadOnly, le compilateur crée une copie du paramètre pour chaque appel de membre d'instance, car il ne peut pas garantir que l'appel ne modifie pas l'état interne. Cela peut aboutir à une multitude de copies et à de mauvaises performances globales que si vous aviez juste passé la structure directement par valeur. Pour obtenir un exemple, consultez ce code sur sharplab

Voici d'autres scénarios dans lesquels des copies masquées peuvent se produire : `static readonly fields` et `literals` . S'ils sont pris en charge à l'avenir, `blittable constants`

finissent par se retrouver dans le même bateau ; c'est qu'ils nécessitent tous actuellement une copie complète (sur l'appel de membre d'instance) si le struct n'est pas marqué `readonly` .

Conception

Permet à un utilisateur de spécifier qu'un membre d'instance est lui-même `readonly` et ne modifie pas l'état de l'instance (avec toutes les vérifications appropriées effectuées par le compilateur, bien entendu). Par exemple :

C#

```
public struct Vector2
{
    public float x;
    public float y;

    public readonly float GetLengthReadonly()
    {
        return MathF.Sqrt(LengthSquared);
    }

    public float GetLength()
    {
        return MathF.Sqrt(LengthSquared);
    }

    public readonly float GetLengthIllegal()
    {
        var tmp = MathF.Sqrt(LengthSquared);

        x = tmp;    // Compiler error, cannot write x
        y = tmp;    // Compiler error, cannot write y

        return tmp;
    }

    public readonly float LengthSquared
    {
        get
        {
            return (x * x) +
                   (y * y);
        }
    }
}

public static class MyClass
{
    public static float ExistingBehavior(in Vector2 vector)
    {
```

```
        // This code causes a hidden copy, the compiler effectively emits:
        //    var tmpVector = vector;
        //    return tmpVector.GetLength();
        //
        // This is done because the compiler doesn't know that `GetLength()`
        // won't mutate `vector`.

        return vector.GetLength();
    }

    public static float ReadonlyBehavior(in Vector2 vector)
    {
        // This code is emitted exactly as listed. There are no hidden
        // copies as the `readonly` modifier indicates that the method
        // won't mutate `vector`.

        return vector.GetLengthReadonly();
    }
}
```

ReadOnly peut être appliqué aux accesseurs de propriété pour indiquer que `this` ne sera pas muté dans l'accesseur. Les exemples suivants ont des accesseurs set en lecture seule, car ces accesseurs modifient l'état du champ de membre, mais ne modifient pas la valeur de ce champ de membre.

C#

```
public readonly int Prop1
{
    get
    {
        return this._store["Prop1"];
    }
    set
    {
        this._store["Prop1"] = value;
    }
}
```

Lorsque `readonly` est appliqué à la syntaxe de propriété, cela signifie que tous les accesseurs sont `readonly` .

C#

```
public readonly int Prop2
{
    get
    {
        return this._store["Prop2"];
    }
```

```
    set
    {
        this._store["Prop2"] = value;
    }
}
```

ReadOnly ne peut être appliqué qu'à des accesseurs qui n'effectuent pas de mutation sur le type conteneur.

C#

```
public int Prop3
{
    readonly get
    {
        return this._prop3;
    }
    set
    {
        this._prop3 = value;
    }
}
```

ReadOnly peut être appliqué à certaines propriétés implémentées automatiquement, mais il n'aura pas d'effet significatif. Le compilateur traite tous les getters implémentés automatiquement comme ReadOnly, que le `readonly` mot clé soit présent ou non.

C#

```
// Allowed
public readonly int Prop4 { get; }
public int Prop5 { readonly get; }
public int Prop6 { readonly get; set; }

// Not allowed
public readonly int Prop7 { get; set; }
public int Prop8 { get; readonly set; }
```

La lecture seule peut être appliquée à des événements implémentés manuellement, mais pas à des événements de type champ. ReadOnly ne peut pas être appliqué à des accesseurs d'événement individuels (Add/Remove).

C#

```
// Allowed
public readonly event Action<EventArgs> Event1
{
    add { }
```

```
    remove { }
}

// Not allowed
public readonly event Action<EventArgs> Event2;
public event Action<EventArgs> Event3
{
    readonly add { }
    readonly remove { }
}
public static readonly event Event4
{
    add { }
    remove { }
}
```

Autres exemples de syntaxe :

- Membres de l'expression : `public readonly float ExpressionBodiedMember => (x * x) + (y * y);`
- Contraintes génériques : `public readonly void GenericMethod<T>(T value) where T : struct { }`

Le compilateur émettra le membre d'instance, comme d'habitude, et émettrait en outre un attribut reconnu par le compilateur indiquant que le membre d'instance ne modifie pas l'État. Le paramètre Hidden est alors `this` à la `in T` place de `ref T` .

Cela permet à l'utilisateur d'appeler sans risque cette méthode d'instance sans que le compilateur doive effectuer une copie.

Les restrictions sont les suivantes :

- Le `readonly` modificateur ne peut pas être appliqué à des méthodes statiques, à des constructeurs ou à des destructeurs.
- Le `readonly` modificateur ne peut pas être appliqué aux délégués.
- Le `readonly` modificateur ne peut pas être appliqué aux membres de la classe ou de l'interface.

Inconvénients

Mêmes inconvénients que pour les `readonly struct` méthodes actuelles. Certains codes peuvent toujours provoquer des copies masquées.

Remarques

L’utilisation d’un attribut ou d’un autre mot clé peut également être possible.

Cette proposition est quelque peu liée à (mais il s’agit plus d’un sous-ensemble de) `functional purity` et/ou `constant expressions` , qui ont tous les deux eu des propositions existantes.

Autoriser `stackalloc` dans les contextes imbriqués

Article • 13/05/2021

Allocation de piles

Nous modifions la section *allocation de pile* de la spécification du langage C# pour assouplir les endroits où une `stackalloc` expression peut apparaître. Nous supprimons

antlr

```
local_variable_initializer_unsafe
    : stackalloc_initializer
    ;

stackalloc_initializer
    : 'stackalloc' unmanaged_type '[' expression ']'
    ;
```

et remplacez-les par

antlr

```
primary_no_array_creation_expression
    : stackalloc_initializer
    ;

stackalloc_initializer
    : 'stackalloc' unmanaged_type '[' expression? ']' array_initializer?
    | 'stackalloc' '[' expression? ']' array_initializer
    ;
```

Notez que l'ajout d'un *array_initializer* à *stackalloc_initializer* (et la création d'une expression d'index facultative) était une extension en C# 7,3 et n'est pas décrit ici.

Le *type d'élément* de l' `stackalloc` expression est le *unmanaged_type* nommé dans l'expression stackalloc, le cas échéant, ou le type commun parmi les éléments de la *array_initializer* dans le cas contraire.

Le type de l' *stackalloc_initializer* avec le *type d'élément* `K` dépend de son contexte syntaxique :

- Si le *stackalloc_initializer* apparaît directement comme *local_variable_initializer* d'une instruction *local_variable_declaration* ou d'un *for_initializer*, son type est `K*` .
- Dans le cas contraire, son type est `System.Span<K>` .

Conversion stackalloc

La *conversion stackalloc* est une nouvelle conversion implicite intégrée à partir de l'expression. Lorsque le type d'un *stackalloc_initializer* est `K*` , il existe une *conversion stackalloc* implicite du *stackalloc_initializer* au type `System.Span<K>` .

Enregistrements

Article • 13/05/2021

Cette proposition effectue le suivi de la spécification de la fonctionnalité d'enregistrements C# 9, comme convenu par l'équipe de conception du langage C#.

La syntaxe d'un enregistrement est la suivante :

antlr

```
record_declaration
    : attributes? class_modifier* 'partial'? 'record' identifier
type_parameter_list?
      parameter_list? record_base? type_parameter_constraints_clause*
record_body
    ;

record_base
    : ':' class_type argument_list?
    | ':' interface_type_list
    | ':' class_type argument_list? ',' interface_type_list
    ;

record_body
    : '{' class_member_declaration* '}' ';'?
    | ';'
    ;
```

Les types d'enregistrements sont des types référence, similaires à une déclaration de classe. Si un enregistrement `record_base` `argument_list` `record_declaration` ne contient pas de, il s'agit d'une erreur `parameter_list` . Au plus une déclaration de type partiel d'un enregistrement partiel peut fournir un `parameter_list` .

Les paramètres d'enregistrement ne peuvent pas utiliser `ref` `out` `this` les modificateurs, ou (mais `in` `params` sont autorisés).

Héritage

Les enregistrements ne peuvent pas hériter de classes, sauf si la classe est `object` , et les classes ne peuvent pas hériter d'enregistrements. Les enregistrements peuvent hériter d'autres enregistrements.

Membres d'un type d'enregistrement

En plus des membres déclarés dans le corps d'enregistrement, un type d'enregistrement a des membres synthétisés supplémentaires. Les membres sont synthétisés, sauf si un membre avec une signature « correspondante » est déclaré dans le corps de l'enregistrement ou si un membre non virtuel concret accessible avec une signature « correspondante » est hérité. Deux membres sont considérés comme correspondants s'ils ont la même signature ou sont considérés comme « masqués » dans un scénario d'héritage. Le fait qu'un membre d'un enregistrement soit nommé « clone » est une erreur. Il s'agit d'une erreur pour un champ d'instance d'un enregistrement dont le type n'est pas sécurisé.

Les membres synthétisés sont les suivants :

Membres d'égalité

Si l'enregistrement est dérivé de `object` , le type d'enregistrement comprend une propriété en lecture seule synthétisée équivalant à une propriété déclarée comme suit :

```
C#

Type EqualityContract { get; };
```

La propriété est `private` si le type d'enregistrement est `sealed` . Sinon, la propriété est `virtual` et `protected` . La propriété peut être déclarée explicitement. Il s'agit d'une erreur si la déclaration explicite ne correspond pas à la signature ou l'accessibilité attendue, ou si la déclaration explicite ne permet pas de la substituer dans un type dérivé et que le type d'enregistrement n'est pas `sealed` .

Si le type d'enregistrement est dérivé d'un type d'enregistrement de base `Base` , le type d'enregistrement comprend une propriété ReadOnly synthétisée équivalant à une propriété déclarée comme suit :

```
C#

protected override Type EqualityContract { get; };
```

La propriété peut être déclarée explicitement. Il s'agit d'une erreur si la déclaration explicite ne correspond pas à la signature ou l'accessibilité attendue, ou si la déclaration explicite ne permet pas de la substituer dans un type dérivé et que le type d'enregistrement n'est pas `sealed` . Il s'agit d'une erreur si une propriété synthétisée ou explicitement déclarée ne remplace pas une propriété avec cette signature dans le type d'enregistrement `Base` (par exemple, si la propriété est manquante dans `Base` , ou

sealed, ou non virtuelle, etc.). La propriété synthétisée retourne `typeof(R)` où `R` est le type d'enregistrement.

Le type d'enregistrement implémente `System.IEquatable<R>` et comprend une surcharge fortement typée synthétisée de `Equals(R? other)` où `R` est le type d'enregistrement. La méthode est `public` , et la méthode est, `virtual` sauf si le type d'enregistrement est `sealed` . La méthode peut être déclarée explicitement. Il s'agit d'une erreur si la déclaration explicite ne correspond pas à la signature ou à l'accessibilité attendue, ou que la déclaration explicite ne permet pas de la substituer dans un type dérivé et que le type d'enregistrement n'est pas `sealed` .

Si `Equals(R? other)` est défini par l'utilisateur (non synthétisé) mais qu'il ne l'est `GetHashCode` pas, un avertissement est généré.

C#

```
public virtual bool Equals(R? other);
```

La synthèse `Equals(R?)` retourne `true` si et seulement si chacun des éléments suivants est `true` :

- `other` n'est pas `null` , et
- Pour chaque champ d'instance `fieldN` dans le type d'enregistrement qui n'est pas hérité, la valeur de `System.Collections.Generic.EqualityComparer<TN>.Default.Equals(fieldN, other.fieldN)` où `TN` est le type de champ, et
- S'il existe un type d'enregistrement de base, la valeur de `base.Equals(other)` (un appel non virtuel à `public virtual bool Equals(Base? other)`); sinon, la valeur de `EqualityContract == other.EqualityContract` .

Le type d'enregistrement comprend `==` des opérateurs synthétisés et `!=` équivalents aux opérateurs déclarés comme suit :

C#

```
public static bool operator==(R? left, R? right)
    => (object)left == right || (left?.Equals(right) ?? false);
public static bool operator!=(R? left, R? right)
    => !(left == right);
```

La `Equals` méthode appelée par l' `==` opérateur est la `Equals(R? other)` méthode spécifiée ci-dessus. L' `!=` opérateur délègue à l' `==` opérateur. Il s'agit d'une erreur si les

opérateurs sont déclarés explicitement.

Si le type d'enregistrement est dérivé d'un type d'enregistrement de base `Base` , le type d'enregistrement comprend un remplacement synthétisé équivalent à une méthode déclarée comme suit :

```
C#

public sealed override bool Equals(Base? other);
```

Il s'agit d'une erreur si la substitution est déclarée explicitement. Il s'agit d'une erreur si la méthode ne remplace pas une méthode avec la même signature dans le type d'enregistrement `Base` (par exemple, si la méthode est manquante dans `Base` , ou sealed, ou non virtuelle, etc.). La valeur de substitution synthétisée est retournée `Equals((object?)other)` .

Le type d'enregistrement comprend un remplacement synthétisé équivalent à une méthode déclarée comme suit :

```
C#

public override bool Equals(object? obj);
```

Il s'agit d'une erreur si la substitution est déclarée explicitement. Il s'agit d'une erreur si la méthode n'est pas substituée `object.Equals(object? obj)` (par exemple, en raison de l'occultation dans les types de base intermédiaires, etc.). La valeur de remplacement synthétisé retourne `Equals(other as R)` où `R` est le type d'enregistrement.

Le type d'enregistrement comprend un remplacement synthétisé équivalent à une méthode déclarée comme suit :

```
C#

public override int GetHashCode();
```

La méthode peut être déclarée explicitement. Il s'agit d'une erreur si la déclaration explicite ne permet pas de la remplacer dans un type dérivé et que le type d'enregistrement n'est pas `sealed` . Il s'agit d'une erreur si une méthode synthétisée ou explicitement déclarée ne se substitue pas `object.GetHashCode()` (par exemple, en raison de l'occultation dans les types de base intermédiaires, etc.).

Un avertissement est signalé si l'un des `Equals(R?)` et `GetHashCode()` est explicitement déclaré, mais que l'autre méthode n'est pas explicite.

La substitution synthétisée de `GetHashCode()` retourne un `int` résultat de la combinaison des valeurs suivantes :

- Pour chaque champ d'instance `fieldN` dans le type d'enregistrement qui n'est pas hérité, la valeur de `System.Collections.Generic.EqualityComparer<TN>.Default.GetHashCode(fieldN)` où `TN` est le type de champ, et
- S'il existe un type d'enregistrement de base, la valeur de `base.GetHashCode()` ; sinon, la valeur de `System.Collections.Generic.EqualityComparer<System.Type>.Default.GetHashCode(EqualityContract)` .

Par exemple, considérez les types d'enregistrements suivants :

C#

```
record R1(T1 P1);
record R2(T1 P1, T2 P2) : R1(P1);
record R3(T1 P1, T2 P2, T3 P3) : R2(P1, P2);
```

Pour ces types d'enregistrements, les membres de l'égalité synthétisée ressemblent à ceci :

C#

```
class R1 : IEquatable<R1>
{
    public T1 P1 { get; init; }
    protected virtual Type EqualityContract => typeof(R1);
    public override bool Equals(object? obj) => Equals(obj as R1);
    public virtual bool Equals(R1? other)
    {
        return !(other is null) &&
            EqualityContract == other.EqualityContract &&
            EqualityComparer<T1>.Default.Equals(P1, other.P1);
    }
    public static bool operator==(R1? left, R1? right)
        => (object)left == right || (left?.Equals(right) ?? false);
    public static bool operator!=(R1? left, R1? right)
        => !(left == right);
    public override int GetHashCode()
    {
        return
Combine(EqualityComparer<Type>.Default.GetHashCode(EqualityContract),
            EqualityComparer<T1>.Default.GetHashCode(P1));
    }
}
```

```
class R2 : R1, IEquatable<R2>
{
    public T2 P2 { get; init; }
    protected override Type EqualityContract => typeof(R2);
    public override bool Equals(object? obj) => Equals(obj as R2);
    public sealed override bool Equals(R1? other) => Equals((object?)other);
    public virtual bool Equals(R2? other)
    {
        return base.Equals((R1?)other) &&
            EqualityComparer<T2>.Default.Equals(P2, other.P2);
    }
    public static bool operator==(R2? left, R2? right)
        => (object)left == right || (left?.Equals(right) ?? false);
    public static bool operator!=(R2? left, R2? right)
        => !(left == right);
    public override int GetHashCode()
    {
        return Combine(base.GetHashCode(),
            EqualityComparer<T2>.Default.GetHashCode(P2));
    }
}

class R3 : R2, IEquatable<R3>
{
    public T3 P3 { get; init; }
    protected override Type EqualityContract => typeof(R3);
    public override bool Equals(object? obj) => Equals(obj as R3);
    public sealed override bool Equals(R2? other) => Equals((object?)other);
    public virtual bool Equals(R3? other)
    {
        return base.Equals((R2?)other) &&
            EqualityComparer<T3>.Default.Equals(P3, other.P3);
    }
    public static bool operator==(R3? left, R3? right)
        => (object)left == right || (left?.Equals(right) ?? false);
    public static bool operator!=(R3? left, R3? right)
        => !(left == right);
    public override int GetHashCode()
    {
        return Combine(base.GetHashCode(),
            EqualityComparer<T3>.Default.GetHashCode(P3));
    }
}
```

Copier et cloner des membres

Un type d'enregistrement contient deux membres de copie :

- Constructeur acceptant un seul argument du type d'enregistrement. Il est appelé « constructeur de copie ».

- Méthode « Clone » d'instance sans paramètre publique synthétisée avec un nom réservé par le compilateur

L'objectif du constructeur de copie est de copier l'état du paramètre vers la nouvelle instance en cours de création. Ce constructeur n'exécute aucun initialiseur de champ/propriété d'instance présent dans la déclaration d'enregistrement. Si le constructeur n'est pas explicitement déclaré, un constructeur est synthétisé par le compilateur. Si l'enregistrement est sealed, le constructeur est privé, sinon il est protégé. Un constructeur de copie explicitement déclaré doit être public ou protected, sauf si l'enregistrement est sealed. La première chose que le constructeur doit faire, est d'appeler un constructeur de copie de la base, ou un constructeur d'objet sans paramètre si l'enregistrement hérite de l'objet. Une erreur est signalée si un constructeur de copie défini par l'utilisateur utilise un initialiseur de constructeur implicite ou explicite qui ne respecte pas cette exigence. Après l'appel d'un constructeur de copie de base, un constructeur de copie synthétisé copie les valeurs de tous les champs d'instance implicitement ou explicitement déclarés dans le type d'enregistrement. La seule présence d'un constructeur de copie, qu'il soit explicite ou Implicit, n'empêche pas l'ajout automatique d'un constructeur d'instance par défaut.

Si une méthode « Clone » virtuelle est présente dans l'enregistrement de base, la méthode « Clone » synthétisée la remplace et le type de retour de la méthode est le type conteneur actuel si la fonctionnalité « covariant retourne » est prise en charge et le type de retour de substitution dans le cas contraire. Une erreur est générée si la méthode Clone de l'enregistrement de base est sealed. Si une méthode « Clone » virtuelle n'est pas présente dans l'enregistrement de base, le type de retour de la méthode Clone est le type conteneur et la méthode est virtuelle, sauf si l'enregistrement est sealed ou abstract. Si l'enregistrement conteneur est abstrait, la méthode de clonage synthétisé est également abstraite. Si la méthode « Clone » n'est pas abstraite, elle retourne le résultat d'un appel à un constructeur de copie.

Impression des membres : méthodes PrintMembers et ToString

Si l'enregistrement est dérivé de `object` , l'enregistrement contient une méthode synthétisée équivalente à une méthode déclarée comme suit :

```
C#

bool PrintMembers(System.Text.StringBuilder builder);
```

La méthode est `private` si le type d'enregistrement est `sealed` . Sinon, la méthode est `virtual` et `protected` .

La méthode :

1. pour chacun des membres imprimables de l'enregistrement (champ public non statique et membres de propriété lisibles), ajoute le nom de ce membre suivi de « = » suivi de la valeur du membre, séparée par « , ».
2. retourne la valeur true si l'enregistrement contient des membres imprimables.

Pour un membre qui a un type valeur, nous allons convertir sa valeur en représentation sous forme de chaîne à l'aide de la méthode la plus efficace disponible pour la plateforme cible. À l'heure actuelle, cela signifie appeler `ToString` avant de passer à `StringBuilder.Append` .

Si le type d'enregistrement est dérivé d'un enregistrement de base `Base` , l'enregistrement contient un remplacement synthétisé équivalent à une méthode déclarée comme suit :

```
C#

protected override bool PrintMembers(StringBuilder builder);
```

Si l'enregistrement n'a pas de membre imprimable, la méthode appelle la `PrintMembers` méthode de base à l'aide d'un argument (son `builder` paramètre) et retourne le résultat.

Sinon, la méthode :

1. appelle la `PrintMembers` méthode de base avec un argument (son `builder` paramètre).
2. Si la `PrintMembers` méthode a retourné la valeur true, ajoutez « , » au générateur,
3. pour chacun des membres imprimables de l'enregistrement, ajoute le nom de ce membre suivi de « = » suivi de la valeur du membre : `this.member` (ou `this.member.ToString()` pour les types valeur), séparée par « , »,
4. retourne la valeur true.

La `PrintMembers` méthode peut être déclarée explicitement. Il s'agit d'une erreur si la déclaration explicite ne correspond pas à la signature ou l'accessibilité attendue, ou si la déclaration explicite ne permet pas de la substituer dans un type dérivé et que le type d'enregistrement n'est pas `sealed` .

L'enregistrement contient une méthode synthétisée équivalente à une méthode déclarée comme suit :

```
C#

public override string ToString();
```

La méthode peut être déclarée explicitement. Il s'agit d'une erreur si la déclaration explicite ne correspond pas à la signature ou l'accessibilité attendue, ou si la déclaration explicite ne permet pas de la substituer dans un type dérivé et que le type d'enregistrement n'est pas `sealed` . Il s'agit d'une erreur si une méthode synthétisée ou explicitement déclarée ne se substitue pas `object.ToString()` (par exemple, en raison de l'occultation dans les types de base intermédiaires, etc.).

La méthode synthétisée :

1. crée une `StringBuilder` instance,
2. Ajoute le nom d'enregistrement au générateur, suivi de « { »,
3. appelle la méthode de l'enregistrement `PrintMembers` en lui donnant le générateur, suivi de «» si elle a retourné la valeur true,
4. Ajoute « } »,
5. retourne le contenu du générateur avec `builder.ToString()` .

Par exemple, considérez les types d'enregistrements suivants :

```
C#

record R1(T1 P1);
record R2(T1 P1, T2 P2, T3 P3) : R1(P1);
```

Pour ces types d'enregistrements, les membres de l'impression synthétisée ressemblent à ceci :

```
C#

class R1 : IEquatable<R1>
{
    public T1 P1 { get; init; }

    protected virtual bool PrintMembers(StringBuilder builder)
    {
        builder.Append(nameof(P1));
        builder.Append(" = ");
        builder.Append(this.P1); // or builder.Append(this.P1.ToString());
if P1 has a value type
```

```
        return true;
    }

    public override string ToString()
    {
        var builder = new StringBuilder();
        builder.Append(nameof(R1));
        builder.Append(" { ");

        if (PrintMembers(builder))
            builder.Append(" ");

        builder.Append("}");
        return builder.ToString();
    }
}

class R2 : R1, IEquatable<R2>
{
    public T2 P2 { get; init; }
    public T3 P3 { get; init; }

    protected override bool PrintMembers(StringBuilder builder)
    {
        if (base.PrintMembers(builder))
            builder.Append(", ");

        builder.Append(nameof(P2));
        builder.Append(" = ");
        builder.Append(this.P2); // or builder.Append(this.P2); if P2 has a
value type

        builder.Append(", ");

        builder.Append(nameof(P3));
        builder.Append(" = ");
        builder.Append(this.P3); // or builder.Append(this.P3); if P3 has a
value type

        return true;
    }

    public override string ToString()
    {
        var builder = new StringBuilder();
        builder.Append(nameof(R2));
        builder.Append(" { ");

        if (PrintMembers(builder))
            builder.Append(" ");

        builder.Append("}");
        return builder.ToString();
    }
}
```

Membres d'enregistrement positionnel

Outre les membres ci-dessus, les enregistrements avec une liste de paramètres (« enregistrements positionnels ») synthétisent les membres supplémentaires avec les mêmes conditions que les membres ci-dessus.

Constructeur principal

Un type d'enregistrement a un constructeur public dont la signature correspond aux paramètres de valeur de la déclaration de type. C'est ce que l'on appelle le constructeur principal pour le type, et le constructeur de classe par défaut déclaré implicitement, s'il est présent, doit être supprimé. C'est une erreur d'avoir un constructeur principal et un constructeur avec la même signature déjà présente dans la classe.

Au moment de l'exécution, le constructeur principal

1. exécute les initialiseurs d'instance qui apparaissent dans le corps de classe
2. appelle le constructeur de classe de base avec les arguments fournis dans la `record_base` clause, le cas échéant.

Si un enregistrement a un constructeur principal, tout constructeur défini par l'utilisateur, à l'exception de « constructeur de copie », doit avoir un `this` initialiseur de constructeur explicite.

Les paramètres du constructeur principal ainsi que les membres de l'enregistrement se trouvent dans la portée au sein `argument_list` de la `record_base` clause et dans les initialiseurs de champs d'instance ou de propriétés. Les membres d'instance seraient une erreur à ces emplacements (à l'instar de la façon dont les membres d'instance sont dans la portée dans les initialiseurs de constructeur standard aujourd'hui, mais une erreur à utiliser), mais les paramètres du constructeur principal seraient dans la portée et utilisables et créeraient un cliché instantané des membres. Les membres statiques seraient également utilisables, de la même façon que les appels de base et les initialiseurs fonctionnent aujourd'hui dans les constructeurs ordinaires.

Un avertissement est généré si un paramètre du constructeur principal n'est pas lu.

Les variables d'expression déclarées dans le `argument_list` sont dans la portée dans le `argument_list` . Les mêmes règles d'occultation qu'au sein d'une liste d'arguments d'un initialiseur de constructeur normal s'appliquent.

Propriétés

Pour chaque paramètre d'enregistrement d'une déclaration de type d'enregistrement, il existe un membre de propriété public correspondant dont le nom et le type sont extraits de la déclaration de paramètre de valeur.

Pour un enregistrement :

- Une `get` propriété publique et une `init` propriété automatique sont créées (voir `init` spécification d'accesseur distincte). Une propriété héritée `abstract` avec le type correspondant est substituée. Il s'agit d'une erreur si la propriété héritée n'a pas `public` `get` de `init` accesseurs substituables. Il s'agit d'une erreur si la propriété héritée est masquée.
  La propriété automatique est initialisée à la valeur du paramètre de constructeur principal correspondant. Les attributs peuvent être appliqués à la propriété automatique synthétisée et à son champ de stockage à l'aide `property:` `field:` de cibles ou pour les attributs qui sont appliqués de façon syntaxique au paramètre d'enregistrement correspondant.

Déconstruire

Un enregistrement positionnel avec au moins un paramètre synthétise une méthode d'instance publique qui retourne une void appelée Deconstruct avec une déclaration de paramètre out pour chaque paramètre de la déclaration du constructeur principal. Chaque paramètre de la méthode Deconstruct a le même type que le paramètre correspondant de la déclaration du constructeur principal. Le corps de la méthode assigne à la valeur d'un membre d'instance un accès à un membre du même nom. La méthode peut être déclarée explicitement. Il s'agit d'une erreur si la déclaration explicite ne correspond pas à la signature ou à l'accessibilité attendue, ou qu'elle est statique.

Expression `with`

Une `with` expression est une nouvelle expression utilisant la syntaxe suivante.

antlr

```
with_expression
    : switch_expression
    | switch_expression 'with' '{' member_initializer_list? '}'
    ;

member_initializer_list
    : member_initializer (',' member_initializer)*
```

```
    ;

member_initializer
    : identifier '=' expression
    ;
```

Une `with` expression n'est pas autorisée en tant qu'instruction.

Une `with` expression permet une « mutation non destructrice », conçue pour produire une copie de l'expression du récepteur avec les modifications des assignations dans le `member_initializer_list` .

Une `with` expression valide a un récepteur avec un type non void. Le type de récepteur doit être un enregistrement.

Sur le côté droit de l' `with` expression se trouve un `member_initializer_list` avec une séquence d'assignations à *identifier*, qui doit être un champ ou une propriété d'instance accessible du type du récepteur.

Tout d'abord, la méthode « Clone » du récepteur (spécifiée ci-dessus) est appelée et son résultat est converti en type du récepteur. Ensuite, chaque `member_initializer` est traité de la même façon qu'une assignation à un champ ou un accès à une propriété du résultat de la conversion. Les affectations sont traitées dans l'ordre lexical.

Instructions de niveau supérieur

Article • 13/05/2021

- [x] proposé
- [x] prototype : démarré
- [x] implémentation : démarré
- [] Spécification : non démarrée

Résumé

Autoriser une séquence d' *instructions* à se produire juste avant la *namespace_member_declaration* s d'un *compilation_unit* (c.-à-d. fichier source).

La sémantique est que, si une telle séquence d' *instructions* est présente, la déclaration de type suivante, Modulo le nom de type réel et le nom de la méthode, serait émise :

```c#
static class Program
{
    static async Task Main(string[] args)
    {
        // statements
    }
}
```

Voir aussi https://github.com/dotnet/csharplang/issues/3117 .

Motivation

Il y a une certaine quantité de programme, même la plus simple des programmes, en raison de la nécessité d'une `Main` méthode explicite. Cela semble être un moyen d'apprendre et de clarifier le programme. L'objectif principal de cette fonctionnalité est donc d'autoriser les programmes C# sans aucune réutilisabilité inutile, à des fins d'apprentissage et de clarté du code.

Conception détaillée

Syntax

La seule syntaxe supplémentaire autorise une séquence d' *instructions* dans une unité de compilation, juste avant la *namespace_member_declaration* s :

antlr

```
compilation_unit
    : extern_alias_directive* using_directive* global_attributes? statement*
namespace_member_declaration*
    ;
```

Un seul *compilation_unit* est autorisé à avoir des *instructions*.

Exemple :

c#

```
if (args.Length == 0
    || !int.TryParse(args[0], out int n)
    || n < 0) return;
Console.WriteLine(Fib(n).curr);

(int curr, int prev) Fib(int i)
{
    if (i == 0) return (1, 0);
    var (curr, prev) = Fib(i - 1);
    return (curr + prev, curr);
}
```

Sémantique

Si des instructions de niveau supérieur sont présentes dans une unité de compilation du programme, la signification est comme si elles étaient combinées dans le corps de bloc d'une `Main` méthode d'une `Program` classe de l'espace de noms global, comme suit :

c#

```
static class Program
{
    static async Task Main(string[] args)
    {
        // statements
    }
}
```

Notez que les noms « Program » et « main » sont utilisés uniquement à des fins d'illustration. les noms réels utilisés par le compilateur sont dépendants de

l'implémentation et ni du type, ni la méthode ne peut être référencée par nom à partir du code source.

La méthode est désignée comme point d'entrée du programme. Les méthodes déclarées explicitement qui, par Convention, peuvent être considérées comme des candidats à un point d'entrée sont ignorées. Un avertissement est signalé lorsque cela se produit. Il est erroné de spécifier le `-main:<type>` commutateur du compilateur lorsqu'il existe des instructions de niveau supérieur.

La méthode de point d'entrée a toujours un paramètre formel, `string[] args` . L'environnement d'exécution crée et passe un `string[]` argument contenant les arguments de ligne de commande spécifiés au démarrage de l'application. L' `string[]` argument n'est jamais null, mais peut avoir une longueur de zéro si aucun argument de ligne de commande n'a été spécifié. Le paramètre « args » se trouve dans la portée dans les instructions de niveau supérieur et n'est pas dans la portée en dehors de celui-ci. Des règles de conflit ou de masquage de noms standard s'appliquent.

Les opérations asynchrones sont autorisées dans les instructions de niveau supérieur jusqu'à ce qu'elles soient autorisées dans les instructions au sein d'une méthode de point d'entrée Async standard. Toutefois, elles ne sont pas requises, si des `await` expressions et d'autres opérations asynchrones sont omises, aucun avertissement n'est généré.

La signature de la méthode de point d'entrée générée est déterminée en fonction des opérations utilisées par les instructions de niveau supérieur comme suit :

Async-operations\Return-with-expression	Présent	Manqu
Présent	`static Task<int> Main(string[] args)`	`static Task Main(string[] args)`
Manqu	`static int Main(string[] args)`	`static void Main(string[] args)`

L'exemple ci-dessus produirait la `$Main` déclaration de méthode suivante :

```c#
static class $Program
{
    static void $Main(string[] args)
    {
        if (args.Length == 0
            || !int.TryParse(args[0], out int n)
```

```
            || n < 0) return;
        Console.WriteLine(Fib(n).curr);

        (int curr, int prev) Fib(int i)
        {
            if (i == 0) return (1, 0);
            var (curr, prev) = Fib(i - 1);
            return (curr + prev, curr);
        }
    }
}
```

En même temps, un exemple semblable à celui-ci :

```
c#

await System.Threading.Tasks.Task.Delay(1000);
System.Console.WriteLine("Hi!");
```

donne :

```
c#

static class $Program
{
    static async Task $Main(string[] args)
    {
        await System.Threading.Tasks.Task.Delay(1000);
        System.Console.WriteLine("Hi!");
    }
}
```

Voici un exemple :

```
c#

await System.Threading.Tasks.Task.Delay(1000);
System.Console.WriteLine("Hi!");
return 0;
```

donne :

```
c#

static class $Program
{
    static async Task<int> $Main(string[] args)
    {
        await System.Threading.Tasks.Task.Delay(1000);
```

```
        System.Console.WriteLine("Hi!");
        return 0;
    }
}
```

Et un exemple similaire à celui-ci :

```c#
System.Console.WriteLine("Hi!");
return 2;
```

donne :

```c#
static class $Program
{
    static int $Main(string[] args)
    {
        System.Console.WriteLine("Hi!");
        return 2;
    }
}
```

Portée des variables locales de niveau supérieur et des fonctions locales

Même si les variables locales de niveau supérieur et les fonctions sont « encapsulées » dans la méthode de point d'entrée générée, elles doivent toujours être dans la portée tout au long du programme dans chaque unité de compilation. Dans le cadre de l'évaluation de nom simple, une fois l'espace de noms global atteint :

- Tout d'abord, une tentative est faite pour évaluer le nom dans la méthode de point d'entrée générée et uniquement si cette tentative échoue
- L'évaluation « normale » dans la déclaration d'espace de noms globale est effectuée.

Cela peut entraîner l'occultation de nom des espaces de noms et des types déclarés dans l'espace de noms global, ainsi que l'occultation de noms importés.

Si l'évaluation de nom simple se produit en dehors des instructions de niveau supérieur et que l'évaluation génère une variable locale ou une fonction de niveau supérieur, cela devrait entraîner une erreur.

De cette façon, nous protégeons notre future capacité à mieux répondre aux « fonctions de niveau supérieur » (scénario 2 dans https://github.com/dotnet/csharplang/issues/3117) , et pouvons fournir des diagnostics utiles aux utilisateurs qui pensent par erreur qu'ils sont pris en charge.

Spécification des types référence Nullable

Article • 13/05/2021

Il s'agit d'un travail en cours : plusieurs parties sont manquantes ou incomplètes.

Cette fonctionnalité ajoute deux nouveaux types de types Nullable (types de référence Nullable et types génériques Nullable) aux types valeur Nullable existants, et introduit une analyse de fluide statique à des fins de sécurité null.

Syntax

Types référence Nullable et paramètres de type Nullable

Les types référence Nullable et les paramètres de type Nullable ont la même syntaxe `T?` que la forme abrégée des types valeur Nullable, mais n'ont pas de forme longue correspondante.

Dans le cadre de la spécification, la `nullable_type` production actuelle est renommée `nullable_value_type` , et les `nullable_reference_type` `nullable_type_parameter` productions sont ajoutées :

antlr

```
type
    : value_type
    | reference_type
    | nullable_type_parameter
    | type_parameter
    | type_unsafe
    ;

reference_type
    : ...
    | nullable_reference_type
    ;

nullable_reference_type
    : non_nullable_reference_type '?'
    ;

non_nullable_reference_type
    : reference_type
    ;
```

```
nullable_type_parameter
    : non_nullable_non_value_type_parameter '?'
    ;

non_nullable_non_value_type_parameter
    : type_parameter
    ;
```

Le `non_nullable_reference_type` dans un `nullable_reference_type` doit être un type référence non null (classe, interface, délégué ou tableau).

`non_nullable_non_value_type_parameter`Dans `nullable_type_parameter` doit être un paramètre de type qui n'est pas imposé comme étant un type valeur.

Les types référence Nullable et les paramètres de type Nullable ne peuvent pas se produire aux emplacements suivants :

- comme une classe ou une interface de base
- en tant que récepteur d'un `member_access`
- comme `type` dans un `object_creation_expression`
- comme `delegate_type` dans un `delegate_creation_expression`
- comme `type` dans un `is_expression` , un `catch_clause` ou un `type_pattern`
- en tant que `interface` dans un nom de membre d'interface qualifié complet

Un avertissement est fourni sur un `nullable_reference_type` et `nullable_type_parameter` dans un contexte d'annotation Nullable *désactivé* .

`class` et `class?` contrainte

La `class` contrainte a un équivalent Nullable `class?` :

antlr

```
primary_constraint
    : ...
    | 'class' '?'
    ;
```

Un paramètre de type restreint avec `class` (dans un contexte d'annotation *activé*) doit être instancié avec un type référence qui n'est pas null.

Un paramètre de type restreint avec `class?` (ou `class` dans un contexte d'annotation *désactivé*) peut être instancié avec un type de référence Nullable ou non null.

Un avertissement est fourni sur une `class?` contrainte dans un contexte d'annotation *désactivé* .

`notnull` restrictions

Un paramètre de type restreint avec `notnull` ne peut pas être un type Nullable (type valeur Nullable, type référence Nullable ou paramètre de type Nullable).

antlr

```
primary_constraint
    : ...
    | 'notnull'
    ;
```

`default` restrictions

La `default` contrainte peut être utilisée sur une substitution de méthode ou une implémentation explicite pour lever l'ambiguïté `T?` « paramètre de type Nullable » de « type valeur Nullable » (`Nullable<T>`). En l'absence de la `default` contrainte, une `T?` syntaxe dans une implémentation de substitution ou explicite est interprétée comme `Nullable<T>`

Voir https://github.com/dotnet/csharplang/blob/master/proposals/csharp-9.0/unconstrained-type-parameter-annotations.md#default-constraint

Opérateur null-indulgent avec

L'opérateur de postconnexion `!` est appelé opérateur null-indulgent avec. Il peut être appliqué à un *primary_expression* ou au sein d'un *null_conditional_expression*:

antlr

```
primary_expression
    : ...
    | null_forgiving_expression
    ;

null_forgiving_expression
    : primary_expression '!'
    ;

null_conditional_expression
    : primary_expression null_conditional_operations_no_suppression
```

```
suppression?
    ;

null_conditional_operations_no_suppression
    : null_conditional_operations? '?' '.' identifier type_argument_list?
    | null_conditional_operations? '?' '[' argument_list ']'
    | null_conditional_operations '.' identifier type_argument_list?
    | null_conditional_operations '[' argument_list ']'
    | null_conditional_operations '(' argument_list? ')'
    ;

null_conditional_operations
    : null_conditional_operations_no_suppression suppression?
    ;

suppression
    : '!'
    ;
```

Par exemple :

C#

```
var v = expr!;
expr!.M();
_ = a?.b!.c;
```

`primary_expression` Et `null_conditional_operations_no_suppression` doivent être d'un type Nullable.

L'opérateur postfix n' `!` a aucun effet d'exécution, il prend la valeur du résultat de l'expression sous-jacente. Son seul rôle consiste à modifier l'État null de l'expression en « not null » et à limiter les avertissements en fonction de son utilisation.

Directives de compilateur Nullable

`#nullable` les directives contrôlent l'annotation Nullable et les contextes d'avertissement.

antlr

```
pp_directive
    : ...
    | pp_nullable
    ;

pp_nullable
    : whitespace? '#' whitespace? 'nullable' whitespace nullable_action
```

```
    (whitespace nullable_target)? pp_new_line
        ;

    nullable_action
        : 'disable'
        | 'enable'
        | 'restore'
        ;

    nullable_target
        : 'warnings'
        | 'annotations'
        ;
```

`#pragma warning` les directives sont développées pour permettre la modification du contexte d'avertissement Nullable :

antlr

```
pragma_warning_body
    : ...
    | 'warning' whitespace warning_action whitespace 'nullable'
    ;
```

Par exemple :

C#

```
#pragma warning disable nullable
```

Contextes nullables

Chaque ligne de code source a un *contexte d'annotation Nullable* et un *contexte d'avertissement Nullable*. Ils contrôlent si les annotations Nullable ont un effet et si les avertissements de possibilité de valeur NULL sont fournis. Le contexte d'annotation d'une ligne donnée est *désactivé* ou *activé*. Le contexte d'avertissement d'une ligne donnée est *désactivé* ou *activé*.

Les deux contextes peuvent être spécifiés au niveau du projet (en dehors du code source C#), ou n'importe où dans un fichier source via des `#nullable` directives de préprocesseur. Si aucun paramètre de niveau projet n'est fourni, la valeur par défaut est pour que les deux contextes soient *désactivés*.

La `#nullable` directive contrôle les contextes d'annotation et d'avertissement dans le texte source et a priorité sur les paramètres au niveau du projet.

Une directive définit le ou les contexte (s) qu'elle contrôle pour les lignes de code suivantes, jusqu'à ce qu'une autre directive la remplace, ou jusqu'à la fin du fichier source.

L'effet des directives est le suivant :

- `#nullable disable`: Définit l'annotation Nullable et les contextes d'avertissement sur *désactivé*
- `#nullable enable`: Définit l'annotation Nullable et les contextes d'avertissement sur *activé*
- `#nullable restore`: Restaure l'annotation Nullable et les contextes d'avertissement dans les paramètres du projet
- `#nullable disable annotations`: Définit le contexte d'annotation Nullable sur *désactivé*
- `#nullable enable annotations`: Définit le contexte d'annotation Nullable sur *activé*
- `#nullable restore annotations`: Restaure le contexte d'annotation Nullable aux paramètres du projet
- `#nullable disable warnings`: Définit le contexte d'avertissement Nullable sur *désactivé*
- `#nullable enable warnings`: Définit le contexte d'avertissement Nullable sur *activé*
- `#nullable restore warnings`: Restaure le contexte d'avertissement Nullable aux paramètres du projet

Nullabilité des types

Un type donné peut avoir l'un des trois nullabilities : *oublie*, non *null* et *Nullable*.

Les types non *null* peuvent provoquer des avertissements si une `null` valeur potentielle leur est assignée. Toutefois, les types *oublie* et *Nullable* sont «*null-assignables*» et peuvent avoir des `null` valeurs qui leur sont assignées sans avertissements.

Les valeurs de *oublie* et de types non *nullables* peuvent être déréférencées ou assignées sans avertissements. Toutefois, les valeurs de types *Nullable* sont «*yield null*» et peuvent provoquer des avertissements lorsqu'ils sont déréférencés ou assignés sans vérification de valeur null correcte.

L' *État NULL par défaut* d'un type qui donne NULL est « peut-être null » ou « peut-être par défaut ». L'État NULL par défaut d'un type qui ne génère pas de valeur null est « not null ».

Le genre de type et le contexte d'annotation Nullable qu'il contient se trouvent dans déterminer sa possibilité de valeur NULL :

- Un type valeur `S` non null est toujours non *null*
- Un type valeur Nullable `S?` est toujours *Nullable*
- Un type référence non annoté `C` dans un contexte d'annotation *désactivé* est *oublie*
- Un type référence non annoté `C` dans un contexte d'annotation *activé* n'est pas *null*
- Un type de référence Nullable `C?` est *Nullable* (mais un avertissement peut être généré dans un contexte d'annotation *désactivé*)

Les paramètres de type prennent également en compte les contraintes :

- Un paramètre de type `T` où toutes les contraintes (le cas échéant) sont des types Nullable ou la `class?` contrainte *accepte* la valeur null
- Un paramètre de type `T` dans lequel au moins une contrainte est *oublie* ou non *null* , ou l'une `struct` des `class` contraintes ou `notnull` est
  - *oublie* dans un contexte d'annotation *désactivé*
  - non *null* dans un contexte d'annotation *activé*
- Un paramètre de type Nullable `T?` est *Nullable*, mais un avertissement est généré dans un contexte d'annotation *désactivé* si `T` n'est pas un type valeur

Oublie et non null

Un `type` est considéré comme ayant lieu dans un contexte d'annotation donné lorsque le dernier jeton du type est dans ce contexte.

Si un type de référence donné `C` dans le code source est interprété comme oublie ou non null, dépend du contexte d'annotation de ce code source. Mais une fois qu'elle est établie, elle est considérée comme faisant partie de ce type et « suit », par exemple, lors de la substitution des arguments de type générique. C'est comme s'il s'agissait d'une annotation telle que `?` sur le type, mais invisible.

Contraintes

Les types de référence Nullable peuvent être utilisés en tant que contraintes génériques.

`class?` est une nouvelle contrainte qui signale « peut-être un type de référence Nullable », alors que `class` dans un contexte d'annotation *activé* désigne « type référence non null ».

`default` est une nouvelle contrainte qui signale un paramètre de type qui n'est pas connu comme étant une référence ou un type valeur. Elle ne peut être utilisée que sur

des méthodes substituées et explicitement implémentées. Avec cette contrainte, `T?` correspond à un paramètre de type Nullable, par opposition à un raccourci pour `Nullable<T>` .

`notnull` est une nouvelle contrainte qui désigne un paramètre de type qui n'est pas null.

La possibilité de valeur null d'un argument de type ou d'une contrainte n'a pas d'incidence sur la satisfaction de la contrainte par le type, sauf si ce n'est pas le cas aujourd'hui (les types valeur Nullable ne répondent pas à la `struct` contrainte). Toutefois, si l'argument de type ne satisfait pas aux exigences de possibilité de valeur null de la contrainte, un avertissement peut être fourni.

État null et suivi null

Chaque expression d'un emplacement source donné a un *État null*, ce qui indique s'il est supposé avoir une valeur null. L'État null est « not null », « peut-être null » ou « peut-être par défaut ». L'État null est utilisé pour déterminer si un avertissement doit être fourni à propos des conversions et déréférencements non sécurisés par des valeurs NULL.

La distinction entre « peut-être null » et « peut-être par défaut » est subtile et s'applique aux paramètres de type. La distinction est qu'un paramètre `T` de type qui a l'État « peut-être null » signifie que la valeur est dans le domaine des valeurs autorisées pour, `T` mais cette valeur légale peut inclure `null` . Lorsque la valeur par défaut est « peut-être », cela signifie que la valeur peut être en dehors du domaine légal des valeurs pour `T` .

Exemple :

```c#
// The value `t` here has the state "maybe null". It's possible for `T` to
be instantiated
// with `string?` in which case `null` would be within the domain of legal
values here. The
// assumption though is the value provided here is within the legal values
of `T`. Hence
// if `T` is `string` then `null` will not be a value, just as we assume
that `null` is not
// provided for a normal `string` parameter
void M<T>(T t)
{
    // There is no guarantee that default(T) is within the legal values for
T hence the
    // state *must* be "maybe-default" and hence `local` must be `T?`
```

```
    T? local = default(T);
}
```

Suivi de valeurs NULL pour les variables

Pour certaines expressions qui désignent des variables, des champs ou des propriétés, l'État null est suivi entre les occurrences, en fonction des assignations à celles-ci, les tests effectués sur ceux-ci et le workflow de contrôle entre eux. Cela est similaire à la façon dont l'assignation définie est suivie pour les variables. Les expressions suivies sont les suivantes :

antlr

```
tracked_expression
    : simple_name
    | this
    | base
    | tracked_expression '.' identifier
    ;
```

Où les identificateurs désignent des champs ou des propriétés.

L'État null des variables suivies est « not null » dans le code inaccessible. Cela suit d'autres décisions concernant le code inaccessible, comme considérer tous les variables locales comme étant assignées définitivement.

Décrire les transitions d'État null similaires à l'assignation définie

État null pour les expressions

L'État null d'une expression est dérivé de son formulaire et de son type, et de l'État null des variables qui y sont impliquées.

Littéraux

L'État null d'un `null` littéral dépend du type de cible de l'expression. Si le type cible est un paramètre de type restreint à un type référence, il s'agit de « peut-être par défaut ». Dans le cas contraire, il s'agit de « peut-être null ».

L'État null d'un `default` littéral dépend du type de cible du `default` littéral. Un `default` littéral avec le type de cible `T` a le même État null que l' `default(T)` expression.

L'État null de tout autre littéral est « not null ».

Noms simples

Si un `simple_name` n'est pas classé comme une valeur, son état null est « not null ». Dans le cas contraire, il s'agit d'une expression suivie, et son état null est son état de suivi null à cet emplacement source.

Accès au membre

Si un `member_access` n'est pas classé comme une valeur, son état null est « not null ». Sinon, s'il s'agit d'une expression suivie, son état null est son état de suivi null à cet emplacement source. Dans le cas contraire, son état null est l'État NULL par défaut de son type.

c#

```
var person = new Person();

// The receiver is a tracked expression hence the member_access of the
property
// is tracked as well
if (person.FirstName is not null)
{
    Use(person.FirstName);
}

// The return of an invocation is not a tracked expression hence the
member_access
// of the return is also not tracked
if (GetAnonymous().FirstName is not null)
{
    // Warning: Cannot convert null literal to non-nullable reference type.
    Use(GetAnonymous().FirstName);
}

void Use(string s)
{
    // ...
}

public class Person
{
    public string? FirstName { get; set; }
    public string? LastName { get; set; }

    private static Person s_anonymous = new Person();
    public static Person GetAnonymous() => s_anonymous;
}
```

Appel d'expressions

Si un `invocation_expression` appelle un membre qui est déclaré avec un ou plusieurs attributs pour un comportement spécial Null, l'État null est déterminé par ces attributs. Dans le cas contraire, l'État null de l'expression est l'État NULL par défaut de son type.

L'État null d'un `invocation_expression` n'est pas suivi par le compilateur.

```c#

// The result of an invocation_expression is not tracked
if (GetText() is not null)
{
    // Warning: Converting null literal or possible null value to non-
nullable type.
    string s = GetText();
    // Warning: Dereference of a possibly null reference.
    Use(s);
}

// Nullable friendly pattern
if (GetText() is string s)
{
    Use(s);
}

string? GetText() => ...
Use(string s) {  }
```

Accès aux éléments

Si un `element_access` appelle un indexeur qui est déclaré avec un ou plusieurs attributs pour un comportement spécial Null, l'État null est déterminé par ces attributs. Dans le cas contraire, l'État null de l'expression est l'État NULL par défaut de son type.

```c#

object?[] array = ...;
if (array[0] != null)
{
    // Warning: Converting null literal or possible null value to non-
nullable type.
    object o = array[0];
    // Warning: Dereference of a possibly null reference.
    Console.WriteLine(o.ToString());
}

// Nullable friendly pattern
```

```
if (array[0] is {} o)
{
    Console.WriteLine(o.ToString());
}
```

Accès de base

Si `B` indique le type de base du type englobant, `base.I` a le même État null que `((B)this).I` et `base[E]` a le même État null que `((B)this)[E]` .

Expressions par défaut

`default(T)` a l'État null en fonction des propriétés du type `T` :

- Si le type est un type non *null* , il a l'État null « not null ».
- Sinon, si le type est un paramètre de type, il a l'État NULL « peut-être par défaut ».
- Sinon, il a l'État NULL « peut-être null »

Expressions conditionnelles null ?.

`null_conditional_expression`A l'État null selon le type d'expression. Notez que cela fait référence au type du `null_conditional_expression` , et non au type d'origine du membre appelé :

- Si le type est un type valeur *Nullable* , il a l'État NULL « peut-être null »
- Sinon, si le type est un paramètre de type *Nullable* , il a l'État NULL « peut-être par défaut ».
- Sinon, il a l'État NULL « peut-être null »

Expressions cast

Si une expression de Cast `(T)E` appelle une conversion définie par l'utilisateur, l'État null de l'expression est l'État NULL par défaut pour le type de la conversion définie par l'utilisateur. Sinon :

- Si `T` est un type valeur non *null* , `T` a alors l'État null « not null »
- Sinon `T` , si est un type valeur *Nullable* , `T` a l'État NULL « peut-être null »
- Sinon `T` , si est un type *Nullable* dans le formulaire `U?` où `U` est un paramètre de type, `T` a alors l'État NULL « peut-être par défaut »
- Sinon, si `T` est un type *Nullable* et `E` a un État NULL « peut-être null » ou « peut-être par défaut », `T` l'État null est « peut-être null ».

- Sinon, si `T` est un paramètre de type et qu'il `E` a l'État NULL « peut-être null » ou « peut-être par défaut », `T` l'État null est « peut-être par défaut ».
- Else `T` a le même État null que `E`

Opérateurs unaires et binaires

Si un opérateur unaire ou binaire appelle un opérateur défini par l'utilisateur, l'État null de l'expression est l'État NULL par défaut pour le type de l'opérateur défini par l'utilisateur. Dans le cas contraire, il s'agit de l'État null de l'expression.

Quoi faire pour les chaînes binaires `+` et les délégués ?

Expressions await

L'État null de `await E` est l'État NULL par défaut de son type.

L'opérateur `as`

L'État null d'une `E as T` expression dépend d'abord des propriétés du type `T` . Si le type de `T` n'est pas *null* , l'État null est « not null ». Dans le cas contraire, l'État null dépend de la conversion du type de `E` en type `T` :

- Si la conversion est une identité, une conversion boxing, une référence implicite ou une conversion Nullable implicite, l'État null est l'État null de `E`
- Sinon, si `T` est un paramètre de type, il a l'État NULL « peut-être par défaut »
- Sinon, il a l'État NULL « peut-être null »

Opérateur de fusion Null

L'État null de `E1 ?? E2` est l'État null de `E2`

l'opérateur conditionnel ;

L'État null de `E1 ? E2 : E3` est basé sur l'État null de `E2` et `E3` :

- Si les deux sont « not null », l'État null est « not null ».
- Sinon, si est « peut-être par défaut », l'État null est « peut-être par défaut »
- Sinon, l'État null est « not null »

Expressions de requête

L'État null d'une expression de requête est l'État NULL par défaut de son type.

Travail supplémentaire nécessaire ici

Opérateurs d'assignation

`E1 = E2` et `E1 op= E2` ont le même État null qu' `E2` après l'application de toutes les conversions implicites.

Expressions qui propagent l'État null

`(E)`, `checked(E)` et `unchecked(E)` ont tous le même État null que `E` .

Expressions qui ne sont jamais null

L'État null des formulaires d'expression suivants est toujours « not null » :

- `this` Accès Transact-SQL
- chaînes interpolées
- `new` expressions (objets, délégués, objets anonymes et expressions de création de tableau)
- Expressions `typeof`
- Expressions `nameof`
- fonctions anonymes (méthodes anonymes et expressions lambda)
- expressions null-indulgent avec
- Expressions `is`

Fonctions imbriquées

Les fonctions imbriquées (expressions lambda et fonctions locales) sont traitées comme des méthodes, sauf en ce qui concerne leurs variables capturées. L'état initial d'une variable capturée à l'intérieur d'une fonction lambda ou locale est l'intersection de l'État Nullable de la variable à toutes les « utilisations » de cette fonction imbriquée ou de cette expression lambda. L'utilisation d'une fonction locale est un appel à cette fonction ou l'endroit où elle est convertie en délégué. L'utilisation d'une expression lambda est le point auquel elle est définie dans la source.

Inférence de type

variables locales implicitement typées Nullable

`var` déduit un type annoté pour les types référence et les paramètres de type qui ne sont pas limités comme étant un type valeur. Exemple :

- dans `var s = "";` , `var` est déduit en tant que `string?` .
- dans `var t = new T();` avec un sans contrainte `T` `var` , est déduit en tant que `T?` .

Inférence de type générique

L'inférence de type générique est améliorée pour déterminer si les types référence déduits doivent avoir la valeur null ou non. C'est un meilleur effort. Elle peut générer des avertissements concernant les contraintes de possibilité de valeur null et peut entraîner des avertissements Nullable lorsque les types inférés de la surcharge sélectionnée sont appliqués aux arguments.

La première phase

Les types référence Nullable circulent dans les limites des expressions initiales, comme décrit ci-dessous. En outre, deux nouveaux types de limites, à savoir `null` et `default` sont introduits. Leur but est de traverser des occurrences de `null` ou `default` dans les expressions d'entrée, ce qui peut entraîner la création d'une valeur null dans un type inféré, même si ce n'est pas le cas. Cela fonctionne même pour les types *valeur* Nullable, qui sont améliorés pour récupérer la « nullité » dans le processus d'inférence.

La détermination des limites à ajouter dans la première phase est améliorée comme suit :

Si un argument `Ei` a un type référence, le type `U` utilisé pour l'inférence dépend de l'État null de, ainsi `Ei` que de son type déclaré :

- Si le type déclaré est un type référence non null `U0` ou un type référence Nullable `U0?` , alors
  - Si l'État null de `Ei` est « not null », alors `U` est `U0`
  - Si l'État null de `Ei` est « peut-être null », alors `U` est `U0?`
- Sinon `Ei` , si a un type déclaré, `U` est ce type
- Sinon, si `Ei` est `null` ensuite `U` la limite spéciale `null`
- Sinon, si `Ei` est `default` ensuite `U` la limite spéciale `default`
- Sinon, aucune inférence n'est effectuée.

Inférences exactes, limites supérieures et limites inférieures

Dans les inférences *du* type `U` *vers* le type `V` , si `V` est un type référence Nullable `V0?` , `V0` est utilisé à la place de `V` dans les clauses suivantes.

- Si `V` est l'une des variables de type non fixed, `U` est ajouté comme limite exacte, supérieure ou inférieure comme avant
- Sinon, si `U` est `null` ou `default` , aucune inférence n'est effectuée
- Sinon, si `U` est un type référence Nullable `U0?` , `U0` est utilisé `U` à la place de dans les clauses suivantes.

L'essence est que la possibilité de valeur null liée directement à l'une des variables de type non Fixed est conservée dans ses limites. Pour les inférences qui parcourent davantage les types source et cible, en revanche, la possibilité de valeur null est ignorée. Il est possible qu'il ne corresponde pas, mais si ce n'est pas le cas, un avertissement est émis ultérieurement si la surcharge est choisie et appliquée.

Résolution

Actuellement, la spécification ne permet pas de décrire ce qui se produit lorsque plusieurs limites sont convertibles de l'identité les unes des autres, mais sont différentes. Cela peut se produire entre `object` et `dynamic` , entre les types tuple qui diffèrent uniquement dans les noms d'éléments, entre les types construits et maintenant entre `C` et `C?` pour les types référence.

En outre, nous devons propager la « valeur null » des expressions d'entrée vers le type de résultat.

Pour gérer ces éléments, nous allons ajouter des phases à la résolution, ce qui est maintenant :

1. Rassemblez tous les types dans toutes les limites en tant que candidats, en supprimant `?` de tous les types de référence Nullable
2. Éliminer les candidats en fonction des exigences relatives aux limites exactes, inférieures et supérieures (maintien `null` et `default` limites)
3. Éliminer les candidats qui n'ont pas de conversion implicite vers tous les autres candidats
4. Si les candidats restants n'ont pas tous une conversion d'identité entre eux, l'inférence de type échoue
5. *Fusionnez* les candidats restants comme indiqué ci-dessous.

6. Si le candidat résultant est un type référence ou un type valeur non null et que *toutes* les limites exactes ou *une* des limites inférieures sont des types valeur Nullable, des types référence Nullable ou `null` `default` , `?` est ajouté au candidat résultant, ce qui en fait un type valeur Nullable ou un type référence.

La *fusion* est décrite entre deux types candidats. Elle est transitive et transmutative, de sorte que les candidats peuvent être fusionnés dans n'importe quel ordre avec le même résultat final. Elle n'est pas définie si les deux types candidats ne sont pas convertibles en identité.

La fonction *Merge* accepte deux types candidats et une direction (+ ou -) :

- *Fusionner*(`T` , `T` , *d*) = T
- *Merge*(`S` , `T?` , +) = *fusion*(`S?` , `T` , +) = *Merge*(`S` , `T` , +)`?`
- *Merge*(`S` , `T?` , -) = *fusion*(`S?` , `T` , -) = *Merge*(`S` , `T` , -)
- *Merge*(`C<S1,...,Sn>` , `C<T1,...,Tn>` , +) = `C<` *fusion*(`S1` , `T1` *D1*) `,...,` *fusionner*(`Sn` , `Tn` , *DN*) `>` , *où*
  - `di` = + Si le `i` paramètre de type « TH » de `C<...>` est covariant
  - `di` = - Si le `i` paramètre de type « TH » de `C<...>` est en contre-ou invariant
- *Merge*(`C<S1,...,Sn>` , `C<T1,...,Tn>` , -) = `C<` *fusion*(`S1` , `T1` *D1*) `,...,` *fusionner*(`Sn` , `Tn` , *DN*) `>` , *où*
  - `di` = - Si le `i` paramètre de type « TH » de `C<...>` est covariant
  - `di` = + Si le `i` paramètre de type « TH » de `C<...>` est en contre-ou invariant
- *Merge*(`(S1 s1,..., Sn sn)` , `(T1 t1,..., Tn tn)` , *d*) = `(` *fusion*(`S1` , `T1` , *d*) `n1,...,` *fusion*(`Sn` , `Tn` , *d*) `nn)` , *où*
  - `ni` est absent si `si` et `ti` diffère, ou si les deux sont absents
  - `ni` est `si` si `si` et `ti` sont identiques
- *Merge*(`object` , `dynamic`) = *fusion*(`dynamic` , `object`) = `dynamic`

Avertissements

Attribution de valeur null potentielle

Déréférencement null potentiel

Incompatibilité des valeurs null de contrainte

Types Nullable dans un contexte d'annotation désactivé

Incompatibilité de substitution et d’implémentation de la possibilité de valeur null

Attributs pour le comportement de la valeur null spéciale

Modifications de critères spéciaux pour C# 9,0

Article • 13/05/2021

Nous envisageons un petit nombre d'améliorations apportées à la mise en correspondance de modèles pour C# 9,0 qui ont une synergie naturelle et qui fonctionnent bien pour traiter un certain nombre de problèmes de programmation courants :

- https://github.com/dotnet/csharplang/issues/2925 Modèles de type
- https://github.com/dotnet/csharplang/issues/1350 Modèles entre parenthèses pour appliquer ou mettre en évidence la précédence des nouveaux combinateurs
- https://github.com/dotnet/csharplang/issues/1350 `and` Modèles conjonctives qui requièrent deux modèles différents pour correspondre ;
- https://github.com/dotnet/csharplang/issues/1350 Les `or` modèles disjonctive qui requièrent l'un des deux modèles différents pour correspondre ;
- https://github.com/dotnet/csharplang/issues/1350 `not` Les modèles de négation qui requièrent un modèle donné *ne correspondent pas* ; et
- https://github.com/dotnet/csharplang/issues/812 Les modèles relationnels qui requièrent que la valeur d'entrée soit inférieur à, inférieur ou égal à, etc. une constante donnée.

Modèles entre parenthèses

Les modèles entre parenthèses permettent au programmeur de placer des parenthèses autour d'un modèle. Cela n'est pas si utile avec les modèles existants en C# 8,0, mais les nouvelles combinaisons de modèles introduisent une précédence que le programmeur peut vouloir substituer.

antlr

```
primary_pattern
    : parenthesized_pattern
    | // all of the existing forms
    ;
parenthesized_pattern
    : '(' pattern ')'
    ;
```

Modèles de type

Nous autorisons un type en tant que modèle :

antlr

```
primary_pattern
    : type-pattern
    | // all of the existing forms
    ;
type_pattern
    : type
    ;
```

Ce retcons l' *expression is-type-expression* existante en *est une expression de modèle is* dans laquelle le modèle est un *modèle de type*, bien que nous ne modifiions pas l'arborescence de syntaxe produite par le compilateur.

Un problème d'implémentation subtil est que cette grammaire est ambiguë. Une chaîne telle que `a.b` peut être analysée en tant que nom qualifié (dans un contexte de type) ou en tant qu'expression séparée par des points (dans un contexte d'expression). Le compilateur est déjà en mesure de traiter un nom qualifié de la même manière qu'une expression en pointillés afin de gérer des éléments comme `e is Color.Red` . L'analyse sémantique du compilateur serait encore plus étendue pour être en mesure de lier un modèle de constante (syntaxique) (par exemple, une expression en pointillés) en tant que type afin de le traiter comme un modèle de type lié afin de prendre en charge cette construction.

Après cette modification, vous seriez en mesure d'écrire

C#

```
void M(object o1, object o2)
{
    var t = (o1, o2);
    if (t is (int, string)) {} // test if o1 is an int and o2 is a string
    switch (o1) {
        case int: break; // test if o1 is an int
        case System.String: break; // test if o1 is a string
    }
}
```

Modèles relationnels

Les modèles relationnels permettent au programmeur d'exprimer qu'une valeur d'entrée doit satisfaire une contrainte relationnelle par rapport à une valeur de constante :

C#

```
    public static LifeStage LifeStageAtAge(int age) => age switch
    {
        < 0 =>  LifeStage.Prenatal,
        < 2 =>  LifeStage.Infant,
        < 4 =>  LifeStage.Toddler,
        < 6 =>  LifeStage.EarlyChild,
        < 12 => LifeStage.MiddleChild,
        < 20 => LifeStage.Adolescent,
        < 40 => LifeStage.EarlyAdult,
        < 65 => LifeStage.MiddleAdult,
        _ =>    LifeStage.LateAdult,
    };
```

Les modèles relationnels prennent en charge les opérateurs relationnels `<` ,, `<=` `>` et `>=` sur tous les types intégrés qui prennent en charge ces opérateurs relationnels binaires avec deux opérandes du même type dans une expression. Plus précisément, nous prenons en charge tous ces modèles relationnels pour `sbyte` , `byte` , `short` , `ushort` , `int` , `uint` , `long` , `ulong` , `char` , `float` , `double` , `decimal` , `nint` et `nuint` .

antlr

```
primary_pattern
    : relational_pattern
    ;
relational_pattern
    : '<' relational_expression
    | '<=' relational_expression
    | '>' relational_expression
    | '>=' relational_expression
    ;
```

L'expression est requise pour évaluer une valeur constante. Il s'agit d'une erreur si cette valeur de constante est `double.NaN` ou `float.NaN` . Il s'agit d'une erreur si l'expression est une constante null.

Quand l'entrée est un type pour lequel un opérateur relationnel prédéfini approprié est défini et s'applique à l'entrée comme opérande de gauche et à la constante donnée comme son opérande de droite, l'évaluation de cet opérateur est prise comme la signification du modèle relationnel. Dans le cas contraire, nous convertirons l'entrée dans le type de l'expression à l'aide d'une conversion explicite Nullable ou unboxing. Il s'agit d'une erreur au moment de la compilation si aucune conversion de ce type n'existe. Le

modèle est considéré comme ne correspondant pas si la conversion échoue. Si la conversion réussit, le résultat de l'opération de correspondance de modèle est le résultat de l'évaluation de l'expression `e OP v` où `e` est l'entrée convertie, `OP` est l'opérateur relationnel et `v` est l'expression constante.

Combinateurs de modèle

Les *combinateurs* de modèle permettent de mettre en correspondance les deux modèles différents à l'aide de `and` (cela peut être étendu à un nombre quelconque de modèles par l'utilisation répétée de `and`), l'un ou l'autre des deux modèles `or` (Ditto), ou la *négation* d'un modèle à l'aide de `not` .

Une utilisation courante d'un combinateur est l'idiome

```c#
if (e is not null) ...
```

Plus lisible que l'idiome actuel `e is object` , ce modèle exprime clairement qu'il vérifie une valeur non null.

Les `and` `or` combinateurs et sont utiles pour tester des plages de valeurs

```c#
bool IsLetter(char c) => c is >= 'a' and <= 'z' or >= 'A' and <= 'Z';
```

Cet exemple montre que `and` aura une priorité d'analyse plus élevée (c'est-à-dire qu'elle sera liée plus étroitement) que `or` . Le programmeur peut utiliser le *modèle entre parenthèses* pour rendre la précédence explicite :

```c#
bool IsLetter(char c) => c is (>= 'a' and <= 'z') or (>= 'A' and <= 'Z');
```

Comme tous les modèles, ces combinateurs peuvent être utilisés dans n'importe quel contexte dans lequel un modèle est attendu, y compris les modèles imbriqués, l' *expression-modèle*, l'expression de *commutateur* et le modèle de l'étiquette de cas d'une instruction switch.

```antlr
```

```
pattern
    : disjunctive_pattern
    ;
disjunctive_pattern
    : disjunctive_pattern 'or' conjunctive_pattern
    | conjunctive_pattern
    ;
conjunctive_pattern
    : conjunctive_pattern 'and' negated_pattern
    | negated_pattern
    ;
negated_pattern
    : 'not' negated_pattern
    | primary_pattern
    ;
primary_pattern
    : // all of the patterns forms previously defined
    ;
```

Modification des ambiguïtés de grammaire 7.5.4.2

En raison de l'introduction du *modèle de type*, il est possible qu'un type générique apparaisse avant le jeton `=>` . Par conséquent, nous ajoutons `=>` à l'ensemble de jetons répertoriés dans les *ambiguïtés de grammaire 7.5.4.2* pour permettre la désambiguïsation de la `<` qui commence la liste d'arguments de type. Voir aussi https://github.com/dotnet/roslyn/issues/47614 .

Problèmes ouverts avec les modifications proposées

Syntaxe pour les opérateurs relationnels

Est `and` , `or` et `not` un type de mot clé contextuel ? Dans ce cas, il existe une modification avec rupture (par exemple, par rapport à son utilisation en tant que désignateur dans un *modèle de déclaration*).

Sémantique (par exemple, type) pour les opérateurs relationnels

Nous prévoyons de prendre en charge tous les types primitifs qui peuvent être comparés dans une expression à l'aide d'un opérateur relationnel. La signification dans les cas simples est claire

```c#
bool IsValidPercentage(int x) => x is >= 0 and <= 100;
```

Mais lorsque l'entrée n'est pas de type primitif, quel type tentez-vous de la convertir ?

```c#
bool IsValidPercentage(object x) => x is >= 0 and <= 100;
```

Nous avons proposé que lorsque le type d'entrée est déjà une primitive comparable, il s'agit du type de la comparaison. Toutefois, lorsque l'entrée n'est pas une primitive comparable, nous traitons le relationnel comme incluant un test de type implicite vers le type de la constante sur le côté droit de la relationnelle. Si le programmeur envisage de prendre en charge plusieurs types d'entrée, cela doit être fait de manière explicite :

```c#
bool IsValidPercentage(object x) => x is
    >= 0 and <= 100 or    // integer tests
    >= 0F and <= 100F or  // float tests
    >= 0D and <= 100D;    // double tests
```

Passage des informations de type de gauche à droite de `and`

Il a été suggéré que lorsque vous écrivez un `and` combinant, les informations de type apprises à gauche sur le type de niveau supérieur peuvent circuler vers la droite. Par exemple

```C#
bool isSmallByte(object o) => o is byte and < 100;
```

Ici, le *type d'entrée* du deuxième modèle est limité par les exigences de *type étroit* de gauche du `and` . Nous définissons le type de sémantique restrictive pour tous les modèles comme suit. Le *type étroit* d'un modèle `P` est défini comme suit :

1. Si `P` est un modèle de type, le type *restreint* est le type du type du modèle de type.
2. Si `P` est un modèle de déclaration, le type *restreint* est le type du type du modèle de déclaration.
3. Si `P` est un modèle récursif qui donne un type explicite, le *type restreint* est ce type.
4. Si `P` est mis en correspondance via les règles pour `ITuple`, le *type restreint* est le type `System.Runtime.CompilerServices.ITuple`.
5. Si `P` est un modèle de constante dans lequel la constante n'est pas la constante null et où l'expression n'a pas de *conversion d'expression constante* vers le *type d'entrée*, le *type restreint* est le type de la constante.
6. Si `P` est un modèle relationnel dans lequel l'expression constante n'a pas de *conversion d'expression constante* vers le *type d'entrée*, le *type restreint* est le type de la constante.
7. Si `P` est un `or` modèle, le *type restreint* est le type commun du *type étroit des sous*-modèles si ce type commun existe. À cet effet, l'algorithme de type commun prend uniquement en compte les conversions d'identité, de boxing et de référence implicites, et il prend en compte tous les sous-modèles d'une séquence de `or` modèles (en ignorant les modèles entre parenthèses).
8. Si `P` est un `and` modèle, le *type restreint* est le *type étroit* du modèle de droite. En outre, le *type étroit* du modèle de gauche est le *type d'entrée* du modèle de droite.
9. Sinon, le *type étroit* de `P` est le `P` type d'entrée de.

Définitions de variables et assignation définie

L'ajout de `or` et des `not` modèles crée des problèmes intéressants concernant les variables de modèle et l'assignation définie. Étant donné que les variables peuvent normalement être déclarées une fois au maximum, toute variable de modèle déclarée d'un côté d'un `or` modèle n'est pas définitivement assignée lorsque le modèle correspond. De même, une variable déclarée à l'intérieur d'un `not` modèle n'est pas censée être assignée définitivement lorsque le modèle correspond. La façon la plus simple de résoudre ce cas consiste à interdire la déclaration des variables de modèle dans ces contextes. Toutefois, cela peut être trop restrictif. D'autres approches sont à prendre en compte.

L'un des scénarios qui méritent d'être envisagé est le

```
C#
```

```
if (e is not int i) return;
M(i); // is i definitely assigned here?
```

Cela ne fonctionne pas aujourd'hui car, pour une *expression de modèle is*, les variables de modèle sont considérées comme *définitivement assignées* uniquement lorsque l'*expression is-pattern-expression* a la valeur true (« définitivement assignée quand true »).

La prise en charge de cela serait plus simple (du point de vue du programmeur) que d'ajouter également la prise en charge d'une instruction de condition négative `if`. Même si nous ajoutons une telle prise en charge, les programmeurs se demandent pourquoi l'extrait de code ci-dessus ne fonctionne pas. D'un autre côté, le même scénario dans un `switch` fait moins de sens, car il n'y a aucun point correspondant dans le programme, où l'attribution de la *valeur false* serait significative. Pouvons-nous l'autoriser dans une *expression is-Pattern,* mais pas dans d'autres contextes où les modèles sont autorisés ? Cela semble irrégulier.

Le problème de l'assignation définie dans un *modèle disjonctive* est lié à ce problème.

C#

```
if (e is 0 or int i)
{
    M(i); // is i definitely assigned here?
}
```

Nous devrions uniquement `i` être assignés lorsque l'entrée n'est pas égale à zéro. Toutefois, étant donné que nous ne savons pas si l'entrée est nulle ou non à l'intérieur du bloc, `i` n'est pas définitivement assignée. Toutefois, que se passe-t-il si nous autorisons `i` à être déclarés dans différents modèles mutuellement exclusifs ?

C#

```
if ((e1, e2) is (0, int i) or (int i, 0))
{
    M(i);
}
```

Ici, la variable `i` est définitivement assignée à l'intérieur du bloc et prend la valeur de l'autre élément du tuple lorsqu'un élément zéro est trouvé.

Il a également été suggéré d'autoriser la définition de variables (multiplier) dans tous les cas d'un bloc case :

C#

```
    case (0, int x):
    case (int x, 0):
        Console.WriteLine(x);
```

Pour effectuer l'un de ces travaux, nous devrions définir soigneusement l'emplacement où ces définitions multiples sont autorisées et dans quelles conditions une variable est considérée comme assignée de manière définitive.

Si nous choisissons de différer ce travail jusqu'à une date ultérieure (ce que je conseille), nous pourrions le faire en C# 9

- sous `not` ou `or` , les variables de modèle ne peuvent pas être déclarées.

Ensuite, nous aurions eu le temps de développer une expérience qui fournirait une vision de la valeur possible de la relaxation plus tard.

Diagnostics, subsumption et exhaustivité

Ces nouvelles formes de modèle introduisent de nombreuses nouvelles opportunités pour diagnostiquer les erreurs des programmeurs. Nous devons déterminer les types d'erreurs que nous allons diagnostiquer et comment procéder. Voici quelques exemples :

```
C#

case >= 0 and <= 100D:
```

Ce cas ne peut jamais correspondre (car l'entrée ne peut pas être à la fois un `int` et un `double`). Nous avons déjà une erreur quand nous détectons un cas qui ne peut jamais correspondre, mais que son libellé (« le cas de commutateur a déjà été géré par un cas précédent » et « le modèle a déjà été traité par un ARM précédent de l'expression de switch ») peut être trompeur dans de nouveaux scénarios. Nous pouvons être amenés à modifier la formulation pour simplement indiquer que le modèle ne correspondra jamais à l'entrée.

```
C#

case 1 and 2:
```

De même, il s'agit d'une erreur, car une valeur ne peut pas être à la fois `1` et `2` .

```
C#

case 1 or 2 or 3 or 1:
```

Ce cas peut correspondre, mais le `or 1` à la fin n'ajoute aucune signification au modèle. Je vous conseille de faire en sorte de générer une erreur chaque fois que conjonction ou disjunct d'un modèle composé ne définit pas une variable de modèle ou n'affecte pas l'ensemble des valeurs correspondantes.

C#

```
case < 2: break;
case 0 or 1 or 2 or 3 or 4 or 5: break;
```

Ici, `0 or 1 or` n'ajoute rien au second cas, car ces valeurs auraient été gérées par le premier cas. Cela mérite trop d'être une erreur.

C#

```
byte b = ...;
int x = b switch { <100 => 0, 100 => 1, 101 => 2, >101 => 3 };
```

Une expression de switch telle que celle-ci doit être considérée comme *exhaustive* (elle gère toutes les valeurs d'entrée possibles).

En C# 8,0, une expression de switch avec une entrée de type `byte` est considérée comme exhaustive uniquement si elle contient un ARM final dont le modèle correspond à tout (un *motif de rejet* ou un *modèle de variation*). Même une expression de commutateur qui a un ARM pour chaque `byte` valeur distincte n'est pas considérée comme exhaustive dans C# 8. Pour gérer correctement l'exhaustivité des modèles relationnels, nous devons également gérer ce cas de figure. Il s'agit techniquement d'une modification avec rupture, mais aucun utilisateur n'est susceptible de le constater.

Les Setters init uniquement

Article • 13/05/2021

Résumé

Cette proposition ajoute le concept de propriétés et d'indexeurs init uniquement à C#. Ces indexeurs peuvent être définis au moment de la création de l'objet, mais ne deviennent effectifs `get` qu'une fois la création de l'objet terminée. Cela permet un modèle immuable bien plus flexible en C#.

Motivation

Les mécanismes sous-jacents pour la génération de données immuables en C# n'ont pas changé depuis 1,0. Elles sont conservées :

1. Déclaration de champs comme `readonly` .
2. Déclarer des propriétés qui contiennent uniquement un `get` accesseur.

Ces mécanismes sont efficaces pour permettre la construction de données immuables, mais ils le font en ajoutant des coûts au code réutilisable de types et en choisissant ces types en dehors des fonctionnalités telles que les initialiseurs d'objets et de collections. Cela signifie que les développeurs doivent choisir entre facilité d'utilisation et immuabilité.

Un objet immuable simple comme `Point` nécessite deux fois plus de code de plaque de chaudière pour prendre en charge la construction, comme c'est le cas pour déclarer le type. Plus le type est grand, plus le coût de cette plaque de chaudière est important :

C#

```
struct Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y)
    {
        this.X = x;
        this.Y = y;
    }
}
```

L' `init` accesseur rend les objets immuables plus flexibles en permettant à l'appelant de muter les membres pendant l'acte de construction. Cela signifie que les propriétés immuables de l'objet peuvent participer à des initialiseurs d'objets et, par conséquent, supprimer la nécessité d'utiliser tout le constructeur dans le type. Le `Point` type est désormais simplement :

C#

```
struct Point
{
    public int X { get; init; }
    public int Y { get; init; }
}
```

Le consommateur peut ensuite utiliser des initialiseurs d'objet pour créer l'objet

C#

```
var p = new Point() { X = 42, Y = 13 };
```

Conception détaillée

accesseurs init

Une propriété init only (ou indexeur) est déclarée à l'aide de l' `init` accesseur à la place de l' `set` accesseur :

C#

```
class Student
{
    public string FirstName { get; init; }
    public string LastName { get; init; }
}
```

Une propriété d'instance contenant un `init` accesseur est considérée comme définissable dans les cas suivants, sauf dans une fonction locale ou une expression lambda :

- Au cours d'un initialiseur d'objet
- Au cours d'un `with` initialiseur d'expression

- À l'intérieur d'un constructeur d'instance du type conteneur ou dérivé, sur `this` ou `base`
- À l'intérieur `init` de l'accesseur d'une propriété, sur `this` ou `base`
- À l'intérieur des utilisations d'attributs avec des paramètres nommés

Les heures ci-dessus dans lesquelles les `init` accesseurs sont définissables sont collectivement appelées dans ce document en tant que phase de construction de l'objet.

Cela signifie que la `Student` classe peut être utilisée des manières suivantes :

C#

```
var s = new Student()
{
    FirstName = "Jared",
    LastName = "Parosns",
};
s.LastName = "Parsons"; // Error: LastName is not settable
```

Les règles autour desquelles les `init` accesseurs sont définissables s'étendent sur les hiérarchies de types. Si le membre est accessible et que l'objet est connu pour être dans la phase de construction, le membre est définissable. Cela autorise spécifiquement les éléments suivants :

C#

```
class Base
{
    public bool Value { get; init; }
}

class Derived : Base
{
    Derived()
    {
        // Not allowed with get only properties but allowed with init
        Value = true;
    }
}

class Consumption
{
    void Example()
    {
        var d = new Derived() { Value = true; };
    }
}
```

Au point où un `init` accesseur est appelé, l'instance est connue pour être dans la phase de construction ouverte. Par conséquent `init` , un accesseur est autorisé à effectuer les actions suivantes en plus de ce qu'un `set` accesseur normal peut faire :

1. Appeler d'autres `init` accesseurs disponibles via `this` ou `base`
2. Assigner `readonly` des champs déclarés sur le même type via `this`

C#

```
class Complex
{
    readonly int Field1;
    int Field2;
    int Prop1 { get; init ; }
    int Prop2
    {
        get => 42;
        init
        {
            Field1 = 13; // okay
            Field2 = 13; // okay
            Prop1 = 13; // okay
        }
    }
}
```

La possibilité d'assigner `readonly` des champs à partir d'un `init` accesseur est limitée aux champs déclarés sur le même type que l'accesseur. Elle ne peut pas être utilisée pour assigner `readonly` des champs dans un type de base. Cette règle garantit que les auteurs de type restent en contrôle sur le comportement de mutabilité de leur type. Les développeurs qui ne souhaitent pas utiliser `init` ne peuvent pas être affectés par d'autres types :

C#

```
class Base
{
    internal readonly int Field;
    internal int Property
    {
        get => Field;
        init => Field = value; // Okay
    }

    internal int OtherProperty { get; init; }
}

class Derived : Base
{
```

```
    internal readonly int DerivedField;
    internal int DerivedProperty
    {
        get => DerivedField;
        init
        {
            DerivedField = 42;  // Okay
            Property = 0;       // Okay
            Field = 13;         // Error Field is readonly
        }
    }

    public Derived()
    {
        Property = 42;  // Okay
        Field = 13;     // Error Field is readonly
    }
}
```

Lorsque `init` est utilisé dans une propriété virtuelle, toutes les substitutions doivent également être marquées comme `init` . De même, il n'est pas possible de remplacer un simple `set` par `init` .

C#

```
class Base
{
    public virtual int Property { get; init; }
}

class C1 : Base
{
    public override int Property { get; init; }
}

class C2 : Base
{
    // Error: Property must have init to override Base.Property
    public override int Property { get; set; }
}
```

Une `interface` déclaration peut également participer à l' `init` initialisation de style à l'aide du modèle suivant :

C#

```
interface IPerson
{
    string Name { get; init; }
}
```

```
class Init
{
    void M<T>() where T : IPerson, new()
    {
        var local = new T()
        {
            Name = "Jared"
        };
        local.Name = "Jraed"; // Error
    }
}
```

Restrictions de cette fonctionnalité :

- L' `init` accesseur ne peut être utilisé que sur les propriétés d'instance
- Une propriété ne peut pas contenir à la fois un `init` `set` accesseur and
- Toutes les substitutions d'une propriété doivent avoir `init` si la base avait `init` . Cette règle s'applique également à l'implémentation de l'interface.

Structs en lecture seule

`init` les accesseurs (à la fois les accesseurs implémentés automatiquement et les accesseurs implémentés manuellement) sont autorisés sur les propriétés de `readonly` `struct` s, ainsi que les `readonly` Propriétés. `init` les accesseurs ne sont pas autorisés à être marqués `readonly` eux-mêmes, dans `readonly` et non `readonly` `struct` .

C#

```
readonly struct ReadonlyStruct1
{
    public int Prop1 { get; init; } // Allowed
}

struct ReadonlyStruct2
{
    public readonly int Prop2 { get; init; } // Allowed

    public int Prop3 { get; readonly init; } // Error
}
```

Encodage des métadonnées

`init`Les accesseurs de propriété sont émis en tant qu' `set` accesseur standard avec le type de retour marqué avec un modreq de `IsExternalInit` . Il s'agit d'un nouveau type

qui aura la définition suivante :

```C#
namespace System.Runtime.CompilerServices
{
    public sealed class IsExternalInit
    {
    }
}
```

Le compilateur mettra en correspondance le type par nom complet. Il n'est pas nécessaire qu'il s'affiche dans la bibliothèque principale. S'il existe plusieurs types portant ce nom, le compilateur lie l'arrêt dans l'ordre suivant :

1. Celui défini dans le projet en cours de compilation
2. Celui défini dans corelib

Si aucune de ces options n'existe, une erreur d'ambiguïté de type sera émise.

La conception de `IsExternalInit` est plus loin dans ce numéro

Questions

Changements cassants

L'un des principaux points du pivot dans la manière dont cette fonctionnalité est encodée va à la question suivante :

> S'agit-il d'une modification avec rupture binaire à remplacer `init` par `set` ?

`init`Le remplacement de par `set` , ce qui rend une propriété entièrement accessible en écriture, n'est jamais une modification avec rupture source sur une propriété non virtuelle. Il développe simplement l'ensemble des scénarios dans lesquels la propriété peut être écrite. Le seul comportement en question est le fait qu'il reste ou non une modification avec rupture binaire.

Si vous souhaitez apporter la modification `init` à `set` une source et une modification compatible binaire, elle forcera notre main sur la décision modreq et les attributs ci-dessous, car elle va éliminer modreqs en tant que solution. Si, en revanche, cela est considéré comme une opération non intéressante, cela rendra la décision modreq et d'attribut moins affectée.

Résolution Ce scénario n'est pas considéré comme intéressant par LDM.

Modreqs et attributs

La stratégie d'émission pour les `init` accesseurs de propriété doit choisir entre l'utilisation d'attributs ou de modreqs lors de l'émission pendant les métadonnées. Ils ont des compromis différents qui doivent être pris en compte.

L'annotation d'un accesseur Set de propriété à l'aide d'une déclaration modreq signifie que les compilateurs compatibles CLI ignoreront l'accesseur, sauf s'il comprend le modreq. Cela signifie que seuls les compilateurs reconnaissants de `init` lisent le membre. Les compilateurs qui ne `init` sont pas conscients de ignorent l' `set` accesseur et, par conséquent, ne traitent pas accidentellement la propriété comme étant en lecture/écriture.

L'inconvénient de modreq est de `init` devenir une partie de la signature binaire de l' `set` accesseur. L'ajout ou `init` la suppression entraînera l'arrêt du compatbility binaire de l'application.

L'utilisation d'attributs pour annoter l' `set` accesseur signifie que seuls les compilateurs qui comprennent l'attribut sauront pouvoir limiter l'accès à celui-ci. Un compilateur qui n'est pas conscient de `init` le verra comme une simple propriété en lecture/écriture et autorisera l'accès.

Cela signifie apparemment que cette décision est un choix entre la sécurité supplémentaire au détriment de la compatibilité binaire. Le fait d'examiner un peu la sécurité supplémentaire n'est pas exactement ce qu'il semble. Elle ne protège pas contre les circonstances suivantes :

1. Réflexion sur `public` les membres
2. L'utilisation de `dynamic`
3. Compilateurs qui ne reconnaissent pas modreqs

Il convient également de considérer que, lorsque nous effectuons les règles de vérification IL pour .NET 5, `init` est l'une de ces règles. Cela signifie qu'une mise en œuvre supplémentaire sera obtenue en vérifiant simplement les compilateurs émettant un IL vérifiable.

Les principaux langages pour .NET (C#, F # et VB) seront tous mis à jour pour reconnaître ces `init` accesseurs. Par conséquent, le seul scénario réaliste ici est lorsqu'un compilateur C# 9 émet des `init` Propriétés et qu'ils sont visibles par un

ensemble d'outils plus ancien, tel que C# 8, VB 15, etc. C# 8. C'est le compromis à prendre en compte et à peser sur la compatibilité binaire.

Remarque Cette discussion s'applique principalement aux membres uniquement, pas aux champs. Alors que `init` les champs ont été rejetés par LDM, ils sont toujours intéressants à prendre en compte pour la discussion modreq et d'attribut. La `init` fonctionnalité pour les champs est un assouplissement de la restriction existante de `readonly` . Cela signifie que si nous émettez les champs sous la forme de `readonly` + un attribut, il n'y a aucun risque que des compilateurs plus anciens ne soient en utilisant le champ parce qu'ils se reconnaissent déjà `readonly` . Par conséquent, l'utilisation d'un modreq ici n'ajoute pas de protection supplémentaire.

Résolution La fonctionnalité utilise un modreq pour encoder la méthode setter de propriété `init` . Les facteurs intéressants étaient (dans aucun ordre particulier) :

- Souhaitant dissuader les compilateurs plus anciens de violer la `init` sémantique
- Vous souhaitez ajouter ou supprimer `init` dans une `virtual` déclaration, ou à `interface` la fois une modification avec rupture source et binaire.

Étant donné qu'il n'y avait pas non plus de prise en charge significative pour la suppression de en tant `init` que modification compatible binaire, il a été possible de choisir d'utiliser modreq.

init et initonly

Il existe trois formes de syntaxe qui ont été particulièrement étudiées pendant notre réunion LDM :

C#

```
// 1. Use init
int Option1 { get; init; }
// 2. Use init set
int Option2 { get; init set; }
// 3. Use initonly
int Option3 { get; initonly; }
```

Résolution Il n'existait pas de syntaxe très difficilement favorisée dans LDM.

Une attention particulière était de savoir comment le choix de la syntaxe aurait un impact sur notre capacité à faire `init` des membres en tant que fonctionnalité générale dans le futur. Le choix de l'option 1 signifie qu'il serait difficile de définir une propriété qui avait `init` une `get` méthode de style à l'avenir. Finalement, il a été décidé que, si

nous avons décidé de continuer avec les `init` membres généraux à l'avenir, nous pourrions autoriser `init` à être un modificateur dans la liste des accesseurs de propriété, ainsi qu'une petite main pour `init set` . Fondamentalement, les deux déclarations suivantes sont identiques.

C#

```
int Property1 { get; init; }
int Property1 { get; init set; }
```

La décision a été prise de passer `init` en tant qu'accesseur autonome dans la liste des accesseurs de propriété.

Avertir en cas d'échec de l'initialisation

Considérons le scénario suivant. Un type déclare un `init` membre uniquement qui n'est pas défini dans le constructeur. Le code qui construit l'objet doit-il recevoir un avertissement s'il n'a pas pu initialiser la valeur ?

À ce stade, il est clair que le champ ne sera jamais défini et a donc beaucoup de similitudes avec l'avertissement autour de l'échec de l'initialisation des `private` données. Par conséquent, un avertissement aurait apparemment une valeur ici ?

Toutefois, cet avertissement présente des inconvénients importants :

1. Cela complique la modification de la compatibilité `readonly` avec `init` .
2. Elle nécessite de fournir des métadonnées supplémentaires pour indiquer les membres qui doivent être initialisés par l'appelant.

En outre, si nous pensons qu'il y a de la valeur ici dans le scénario global de forcer les créateurs d'objets à être avertis/Errors sur des champs spécifiques, il est probable qu'il s'agit d'une fonctionnalité générale. Il n'y a aucune raison d'être limité à des `init` membres uniquement.

Résolution Il n'y aura aucun avertissement sur la consommation des `init` champs et des propriétés.

LDM souhaite avoir une discussion plus large sur l'idée des champs et propriétés requis. Cela peut nous amener à revenir en arrière et à reconsidérer notre position sur `init` les membres et la validation.

Autoriser init en tant que modificateur de champ

De la même façon `init` peut servir d'accesseur de propriété, il peut également servir de désignation sur les champs pour leur attribuer des comportements similaires en tant que `init` Propriétés. Cela permet d'assigner le champ avant que la construction soit terminée par le type, les types dérivés ou les initialiseurs d'objets.

C#

```
class Student
{
    public init string FirstName;
    public init string LastName;
}

var s = new Student()
{
    FirstName = "Jarde",
    LastName = "Parsons",
}

s.FirstName = "Jared"; // Error FirstName is readonly
```

Dans les métadonnées, ces champs sont marqués de la même façon que les `readonly` champs, mais avec un attribut ou modreq supplémentaire pour indiquer qu'il s'agit de `init` champs de style.

Résolution Le LDM accepte que cette proposition soit un son, mais le scénario est globalement disjoint des propriétés. La décision consistait à continuer uniquement avec les `init` Propriétés pour le moment. Cela offre un niveau de flexibilité approprié, car une `init` propriété peut muter un `readonly` champ sur le type déclarant de la propriété. Ce sera le cas si des commentaires client importants justifient le scénario.

Autoriser init en tant que modificateur de type

De la même façon `readonly` , le modificateur peut être appliqué à un `struct` pour déclarer automatiquement tous les champs comme `readonly` , le `init` seul modificateur peut être déclaré sur `struct` ou `class` pour marquer automatiquement tous les champs comme `init` . Cela signifie que les deux déclarations de type suivantes sont équivalentes :

C#

```
struct Point
{
    public init int X;
    public init int Y;
}

// vs.

init struct Point
{
    public int X;
    public int Y;
}
```

Résolution Cette fonctionnalité est trop *jolie* ici et est en conflit avec la `readonly struct` fonctionnalité sur laquelle elle est basée. La `readonly struct` fonctionnalité est simple en ce qu'elle s'applique `readonly` à tous les membres : champs, méthodes, etc... La `init struct` fonctionnalité s'applique uniquement aux propriétés. Cela finit par compliquer les utilisateurs.

Étant donné que `init` n'est valide que sur certains aspects d'un type, nous avons rejeté l'idée de le faire en tant que modificateur de type.

Considérations

Compatibilité

Cette `init` fonctionnalité est conçue pour être compatible avec les `get` propriétés existantes uniquement. Plus précisément, il s'agit d'une modification entièrement additive pour une propriété qui est `get` uniquement aujourd'hui, mais qui souhaite une plus grande sémantique de création d'objets Flexbile.

Prenons l'exemple du type suivant :

C#

```
class Name
{
    public string First { get; }
    public string Last { get; }

    public Name(string first, string last)
    {
        First = first;
        Last = last;
```

```
    }
}
```

L'ajout `init` à ces propriétés est une modification sans rupture :

```C#
class Name
{
    public string First { get; init; }
    public string Last { get; init; }

    public Name(string first, string last)
    {
        First = first;
        Last = last;
    }
}
```

Vérification IL

Lorsque .NET Core décide de réimplémenter la vérification IL, les règles doivent être ajustées pour prendre en compte les `init` membres. Cela doit être inclus dans les modifications de règle pour l'accession non transmutante aux `readonly` données.

Les règles de vérification IL devront être divisées en deux parties :

1. Autorisation des `init` membres à définir un `readonly` champ.
2. Détermination du moment où un `init` membre peut être appelé légalement.

La première est un ajustement simple aux règles existantes. Le vérificateur IL peut être enseigné pour reconnaître les `init` membres et, à partir de là, il suffit de considérer un champ comme pouvant `readonly` être définissable `this` dans ce membre.

La deuxième règle est plus compliquée. Dans le cas simple des initialiseurs d'objets, la règle est directe. Il doit être légal d'appeler `init` des membres lorsque le résultat d'une `new` expression est toujours sur la pile. Ainsi, tant que la valeur n'a pas été stockée dans un élément ou un élément de tableau local ou passé comme argument à une autre méthode, il sera toujours légal d'appeler `init` les membres. Cela garantit qu'une fois que le résultat de l' `new` expression est publié dans un identificateur nommé (autre que `this`), il n'est plus autorisé à appeler `init` les membres.

Le cas le plus compliqué cependant est quand nous mélangeons `init` les membres, les initialiseurs d'objets et `await` . Cela peut entraîner le placement temporaire de l'objet

nouvellement créé dans une machine à États et le placer dans un champ.

```
C#
```

```
var student = new Student()
{
    Name = await SomeMethod()
};
```

Ici, le résultat de `new Student()` sera hoised dans une machine à États en tant que champ avant l'ensemble de `Name` . Le compilateur doit marquer ces champs levés de telle sorte que le vérificateur IL comprenne qu'ils ne sont pas accessibles par l'utilisateur et ne viole donc pas la sémantique prévue de `init` .

membres init

Le `init` modificateur peut être étendu pour s'appliquer à tous les membres d'instance. Cela généraliserait le concept de la `init` construction de l'objet et autorisera les types à déclarer des méthodes d'assistance qui pouvaient partipate dans le processus de construction pour initialiser les `init` champs et les propriétés.

De tels membres auraient tous les restricions qu'un `init` accesseur effectue dans cette conception. La nécessité est cependant douteuse, mais elle peut être ajoutée en toute sécurité dans une future version du langage de manière compatible.

Générer trois accesseurs

L'une des implémentations potentielles des `init` Propriétés consiste à effectuer une `init` séparation complète de `set` . Cela signifie qu'une propriété peut potentiellement avoir trois accesseurs différents : `get` , `set` et `init` .

Cela présente l'avantage potentiel de permettre l'utilisation de modreq pour garantir l'exactitude tout en conservant la compatibilité binaire. L'implémentation serait à peu près ce qui suit :

1. Un `init` accesseur est toujours émis s'il y a un `set` . Lorsqu'il n'est pas défini par le développeur, il s'agit simplement d'une référence à `set` .
2. Le jeu d'une propriété dans un initialiseur d'objet utilise toujours `init` s'il est présent, mais revient à `set` s'il est manquant.

Cela signifie qu'un développeur peut toujours supprimer en toute sécurité `init` une propriété.

L’inconvénient de cette conception est que n’est utile que si `init` est **toujours** émis lorsqu’il y a un `set` . La langue ne peut pas savoir si `init` a été supprimé par le passé, elle doit supposer qu’elle était et, par conséquent, `init` doit toujours être émis. Cela entraînerait une expansion significative des métadonnées et ne justifie pas ici le coût de la compatibilité.

Expressions de type cible `new`

Article • 13/05/2021

- [x] proposé
- [x] prototype
- [] Implémentation
- [] Spécification

Résumé

Ne requièrent pas de spécification de type pour les constructeurs lorsque le type est connu.

Motivation

Autorisez l'initialisation des champs sans dupliquer le type.

C#

```
Dictionary<string, List<int>> field = new() {
    { "item1", new() { 1, 2, 3 } }
};
```

Autoriser l'omission du type lorsqu'il peut être déduit de l'utilisation.

C#

```
XmlReader.Create(reader, new() { IgnoreWhitespace = true });
```

Instanciez un objet sans épeler le type.

C#

```
private readonly static object s_syncObj = new();
```

Caractéristique

Une nouvelle forme syntaxique, *target_typed_new* de la *object_creation_expression* est acceptée dans laquelle le *type* est facultatif.

antlr

```
object_creation_expression
    : 'new' type '(' argument_list? ')' object_or_collection_initializer?
    | 'new' type object_or_collection_initializer
    | target_typed_new
    ;
target_typed_new
    : 'new' '(' argument_list? ')' object_or_collection_initializer?
    ;
```

Une expression *target_typed_new* n'a pas de type. Toutefois, il existe une nouvelle *conversion de création d'objet* qui est une conversion implicite à partir d'une expression, qui existe d'un *target_typed_new* à chaque type.

Pour un type de cible donné `T` , le type `T0` est `T` sous-jacent du type si `T` est une instance de `System.Nullable` . Sinon, `T0` est `T` . La signification d'une expression *target_typed_new* qui est convertie vers le type `T` est la même que la signification d'un *object_creation_expression* correspondant qui spécifie `T0` comme type.

Il s'agit d'une erreur au moment de la compilation si un *target_typed_new* est utilisé comme opérande d'un opérateur unaire ou binaire, ou s'il est utilisé là où il n'est pas soumis à une *conversion de création d'objet*.

> **Problème d'ouverture :** devons-nous autoriser les délégués et les tuples en tant que type cible ?

Les règles ci-dessus incluent les délégués (type référence) et les tuples (type struct). Bien que les deux types soient constructible, si le type est déduire, une fonction anonyme ou un littéral de tuple peut déjà être utilisé.

C#

```
(int a, int b) t = new(1, 2); // "new" is redundant
Action a = new(() => {}); // "new" is redundant

(int a, int b) t = new(); // OK; same as (0, 0)
Action a = new(); // no constructor found
```

Divers

Les conséquences de la spécification sont les suivantes :

- `throw new()` est autorisé (le type cible est `System.Exception`)

- Le type cible `new` n'est pas autorisé avec les opérateurs binaires.
- Elle n'est pas autorisée lorsqu'il n'y a aucun type à cibler : opérateurs unaires, collection d'un, dans un `foreach` `using` , dans une déconstruction, dans une `await` expression, en tant que propriété de type anonyme (), dans une instruction, dans une instruction, dans un `new { Prop = new() }` accès de membre (), dans une `lock` `sizeof` `fixed` `new().field` opération distribuée dynamiquement (`someDynamic.Method(new())`), dans une requête LINQ, en tant qu'opérande de l' `is` opérateur, comme opérande gauche de l' `??` opérateur,...
- Il est également interdit en tant que `ref` .
- Les genres de types suivants ne sont pas autorisés en tant que cibles de la conversion
  - **Types enum :** `new()` fonctionnera (car `new Enum()` fonctionne pour fournir la valeur par défaut), mais ne `new(1)` fonctionnera pas, car les types enum n'ont pas de constructeur.
  - **Types d'interface :** Cela fonctionnerait de la même façon que l'expression de création correspondante pour les types COM.
  - **Types tableau :** les tableaux ont besoin d'une syntaxe spéciale pour fournir la longueur.
  - **dynamique :** nous n'autorisons pas `new dynamic()` , donc nous n'autorisons pas `new()` avec `dynamic` comme type de cible.
  - **tuples :** Celles-ci ont la même signification que la création d'un objet à l'aide du type sous-jacent.
  - Tous les autres types qui ne sont pas autorisés dans le *object_creation_expression* sont également exclus, par exemple les types pointeur.

Inconvénients

Certains problèmes liés à la création de nouvelles catégories de modifications avec rupture ont été apportés par la cible `new` , mais nous en avons déjà `null` un avec et `default` , et cela n'a pas été un problème significatif.

Autres solutions

La plupart des plaintes sur les types trop longs pour être dupliqués dans l'initialisation de champ concerne les *arguments de type* qui ne sont pas le type lui-même, mais nous pourrions déduire uniquement les arguments de type comme `new Dictionary(...)` (ou similaires) et déduire les arguments de type localement à partir des arguments ou de l'initialiseur de collection.

Questions

- Devons-nous interdire les utilisations dans les arborescences d'expressions ? º
- Comment la fonctionnalité interagit-elle avec les `dynamic` arguments ? (aucun traitement spécial)
- Comment IntelliSense doit-il fonctionner `new()` ? (uniquement s'il existe un seul type cible)

Réunions de conception

- LDM-2017-10-18
- LDM-2018-05-21
- LDM-2018-06-25
- LDM-2018-08-22
- LDM-2018-10-17
- LDM-2020-03-25

Initialiseurs de module

Article • 13/05/2021

- [x] proposé
- [] Prototype : en cours
- [] Implémentation : en cours
- [] Spécification : non démarrée

Résumé

Bien que la plateforme .NET dispose d'une fonctionnalité qui prend directement en charge l'écriture du code d'initialisation de l'assembly (techniquement, le module), elle n'est pas exposée en C#. Il s'agit d'un scénario plutôt que des niches, mais une fois que vous y êtes dans, les solutions semblent assez pénibles. Il existe des rapports d' un certain nombre de clients (à l'intérieur et à l'extérieur de Microsoft) qui éprouvent le problème et il n'y a aucun doute sur les cas non documentés.

Motivation

- Permettez aux bibliothèques d'effectuer des initialisations opportunes et ponctuelles lorsqu'elles sont chargées, avec une charge minimale et sans que l'utilisateur ait besoin d'appeler explicitement quoi que ce soit
- Un point délicat particulier des approches actuelles du `static` constructeur est que le runtime doit effectuer des vérifications supplémentaires sur l'utilisation d'un type avec un constructeur statique, afin de déterminer si le constructeur statique doit être exécuté ou non. Cela ajoute une surcharge mesurable.
- Activer les générateurs de source pour exécuter une logique d'initialisation globale sans que l'utilisateur doive appeler explicitement quoi que ce soit

Conception détaillée

Une méthode peut être désignée en tant qu'initialiseur de module en la décorant avec un `[ModuleInitializer]` attribut.

C#

```
using System;
namespace System.Runtime.CompilerServices
{
    [AttributeUsage(AttributeTargets.Method, AllowMultiple = false)]
```

```
    public sealed class ModuleInitializerAttribute : Attribute { }
}
```

L'attribut peut être utilisé comme suit :

C#

```
using System.Runtime.CompilerServices;
class C
{
    [ModuleInitializer]
    internal static void M1()
    {
        // ...
    }
}
```

Certaines exigences sont imposées sur la méthode ciblée avec cet attribut :

1. La méthode doit être `static` .
2. La méthode doit être sans paramètre.
3. La méthode doit retourner `void`.
4. La méthode ne doit pas être générique ou être contenue dans un type générique.
5. La méthode doit être accessible à partir du module conteneur.
    - Cela signifie que l'accessibilité effective de la méthode doit être `internal` ou `public` .
    - Cela signifie également que la méthode ne peut pas être une fonction locale.

Quand une ou plusieurs méthodes valides avec cet attribut sont trouvées dans une compilation, le compilateur émet un initialiseur de module qui appelle chacune des méthodes attribuées. Les appels seront émis dans un ordre réservé, mais déterministe.

Inconvénients

Pourquoi n'est-ce *pas* ?

- Les outils tiers existants pour « l'injection » d'initialiseurs de module sont peut-être suffisants pour les utilisateurs qui ont demandé cette fonctionnalité.

Réunions de conception

8 avril 2020

Étendre des méthodes partielles

Article • 13/05/2021

Résumé

Cette proposition vise à supprimer toutes les restrictions concernant les signatures des `partial` méthodes en C#. L'objectif est de développer l'ensemble des scénarios dans lesquels ces méthodes peuvent fonctionner avec les générateurs de code source, ainsi qu'une forme de déclaration plus générale pour les méthodes C#.

Consultez également la spécification des méthodes partielles d'origine.

Motivation

C# offre une prise en charge limitée pour les développeurs qui fractionnent les méthodes en déclarations et définitions/implémentations.

C#

```
partial class C
{
    // The declaration of C.M
    partial void M(string message);
}

partial class C
{
    // The definition of C.M
    partial void M(string message) => Console.WriteLine(message);
}
```

L'un des comportements des `partial` méthodes est que lorsque la définition est absente, le langage efface simplement tous les appels à la `partial` méthode. Il se comporte essentiellement comme un appel à une `[Conditional]` méthode dans laquelle la condition a été évaluée à false.

C#

```
partial class D
{
    partial void M(string message);

    void Example()
    {
```

```
        M(GetIt()); // Call to M and GetIt erased at compile time
    }

    string GetIt() => "Hello World";
}
```

La motivation initiale de cette fonctionnalité était la génération de la source sous la forme d'un code généré par le concepteur. Les utilisateurs modifient en permanence le code généré parce qu'ils souhaitaient raccorder certains aspects du code généré. Plus particulièrement les parties du processus de démarrage Windows Forms, après l'initialisation des composants.

La modification du code généré était sujette aux erreurs, car toute action ayant entraîné la régénération du code par le concepteur entraînerait l'effacement de la modification de l'utilisateur. La `partial` fonctionnalité de méthode a facilité la traction, car elle a permis aux concepteurs d'émettre des raccordements sous la forme de `partial` méthodes.

Les concepteurs peuvent émettre `partial void OnComponentInit()` des points de raccordement comme et les développeurs peuvent définir des déclarations pour eux ou ne pas les définir. Dans les deux cas, même si le code généré est compilé et que les développeurs intéressés par le processus pouvaient se raccorder en fonction des besoins.

Cela signifie que les méthodes partielles ont plusieurs restrictions :

1. Doit avoir un `void` type de retour.
2. Impossible d'avoir des `out` paramètres.
3. Impossible d'avoir une accessibilité (implicitement `private`).

Ces restrictions existent, car la langue doit être en mesure d'émettre du code lorsque le site d'appel est effacé. Étant donné qu'ils peuvent être effacés `private` , la seule accessibilité possible est que le membre ne peut pas être exposé dans les métadonnées de l'assembly. Ces restrictions servent également à limiter l'ensemble des scénarios dans lesquels les `partial` méthodes peuvent être appliquées.

La proposition ici consiste à supprimer toutes les restrictions existantes autour des `partial` méthodes. Ils les laissent `out` , les types de retour non void ou n'importe quel type d'accessibilité. Ces `partial` déclarations auront alors la nécessité ajoutée qu'une définition doit exister. Cela signifie que le langage n'a pas à prendre en compte l'impact de l'effacement des sites d'appel.

Cela développerait l'ensemble des scénarios de générateur auxquels les `partial` méthodes pourraient participer et, par conséquent, nous permettons de créer une liaison avec notre fonctionnalité de générateurs de source. Par exemple, une expression régulière peut être définie à l'aide du modèle suivant :

C#

```
[RegexGenerated("(dog|cat|fish)")]
partial bool IsPetMatch(string input);
```

Cela donne aux développeurs un moyen déclaratif simple d'opter pour les générateurs et de donner aux générateurs un ensemble très facile de déclarations à examiner dans le code source pour piloter leur sortie générée.

Comparez cela à la difficulté qu'un générateur aurait raccordé à l'extrait de code suivant.

C#

```
var regex = new RegularExpression("(dog|cat|fish)");
if (regex.IsMatch(someInput))
{

}
```

Étant donné que le compilateur n'autorise pas les générateurs à modifier le raccordement de code, ce modèle serait beaucoup impossible pour les générateurs. Ils doivent recourir à la réflexion dans l' `IsMatch` implémentation ou demander aux utilisateurs de remplacer leurs sites d'appel par une nouvelle méthode + refactoriser l'expression régulière pour passer le littéral de chaîne en tant qu'argument. C'est assez compliqué.

Conception détaillée

Le langage change pour permettre l' `partial` annotation des méthodes avec un modificateur d'accessibilité explicite. Cela signifie qu'ils peuvent être étiquetés comme `private` , `public` , etc.

Quand une `partial` méthode a un modificateur d'accessibilité explicite, bien que le langage exige que la déclaration ait une définition correspondante même lorsque l'accessibilité est `private` :

C#

```
partial class C
{
    // Okay because no definition is required here
    partial void M1();

    // Okay because M2 has a definition
    private partial void M2();

    // Error: partial method M3 must have a definition
    private partial void M3();
}

partial class C
{
    private partial void M2() { }
}
```

En outre, le langage supprimera toutes les restrictions sur ce qui peut apparaître sur une `partial` méthode qui a une accessibilité explicite. Ces déclarations peuvent contenir des types de retour non void, des `out` paramètres, des `extern` modificateurs, etc. Ces signatures auront l'intégralité de l'expression du langage C#.

C#

```
partial class D
{
    // Okay
    internal partial bool TryParse(string s, out int i);
}

partial class D
{
    internal partial bool TryParse(string s, out int i) { }
}
```

Cela permet explicitement aux `partial` méthodes de participer aux `overrides` `interface` implémentations et :

C#

```
interface IStudent
{
    string GetName();
}

partial class C : IStudent
{
    public virtual partial string GetName();
}
```

```
partial class C
{
    public virtual partial string GetName() => "Jarde";
}
```

Le compilateur modifie l'erreur qu'il émet lorsqu'une `partial` méthode contient un élément illégal pour dire :

> Impossible `ref` d'utiliser sur une `partial` méthode qui n'a pas d'accessibilité explicite

Cela aidera les développeurs dans le sens adapté à l'utilisation de cette fonctionnalité.

Restrictions :

- `partial` les déclarations avec accessibilité explicite doivent avoir une définition
- `partial` les déclarations et les signatures de définition doivent correspondre sur tous les modificateurs de méthode et de paramètre. Les seuls aspects qui peuvent différer sont les noms de paramètres et les listes d'attributs (cela n'est pas nouveau, mais plutôt une exigence existante des `partial` méthodes).

Questions

partielle sur tous les membres

Étant donné que nous développons `partial` pour être plus conviviaux pour les générateurs de code source, devons-nous également le développer pour travailler sur tous les membres de classe ? Par exemple, vous devriez pouvoir déclarer des `partial` constructeurs, des opérateurs, etc.

Résolution L'idée est un son, mais à ce stade de la planification C# 9, nous essayons d'éviter la compensation des fonctionnalités inutiles. Vous souhaitez résoudre le problème immédiat lié au développement de la fonctionnalité afin qu'elle fonctionne avec les générateurs de source modernes.

`partial`L'extension de pour prendre en charge d'autres membres sera prise en compte pour la version C# 10. Il semble que nous envisageons cette extension.

Utiliser abstract au lieu de partial

Le essentiel de cette proposition consiste essentiellement à s'assurer qu'une déclaration a une définition/implémentation correspondante. Étant donné que nous devons utiliser, `abstract` car il s'agit déjà d'un mot clé de langage qui oblige le développeur à réfléchir à la mise en œuvre d'une implémentation ?

Résolution Il y a eu une discussion saine sur ce sujet, mais il a finalement été décidé. Oui, les spécifications sont familières, mais les concepts sont très différents. Peut permettre au développeur de croire qu'il a créé des emplacements virtuels alors qu'il ne l'a pas fait.

Fonctions anonymes statiques

Article • 13/05/2021

Résumé

Autorise un modificateur‘static’sur les expressions lambda et les méthodes anonymes, ce qui interdit la capture de paramètres régionaux ou de l’état de l’instance de contenant des étendues.

Motivation

Évitez de capturer de manière involontaire un État dans le contexte englobant, ce qui peut entraîner une rétention inattendue des objets capturés ou des allocations supplémentaires inattendues.

Conception détaillée

Une méthode lambda ou anonyme peut avoir un `static` modificateur. Le `static` modificateur indique que la méthode lambda ou anonyme est une *fonction anonyme statique*.

Une *fonction anonyme statique* ne peut pas capturer l’État à partir de la portée englobante. Par conséquent, les variables locales, les paramètres et `this` de la portée englobante ne sont pas disponibles dans une *fonction anonyme statique*.

Une *fonction anonyme statique* ne peut pas faire référence à des membres d’instance à partir d’une référence implicite ou explicite `this` ou `base` .

Une *fonction anonyme statique* peut faire référence `static` à des membres de la portée englobante.

Une *fonction anonyme statique* peut référencer `constant` des définitions de la portée englobante.

`nameof()` dans une *fonction anonyme statique* peut faire référence à des variables locales, des paramètres ou `this` ou `base` à partir de la portée englobante.

Les règles d’accessibilité des `private` membres de la portée englobante sont les mêmes pour les `static` fonctions et non `static` anonymes.

Aucune garantie n’est faite pour déterminer si une définition de *fonction anonyme statique* est émise comme une `static` méthode dans les métadonnées. Il s’agit de l’implémentation du compilateur à optimiser.

Une fonction non `static` locale ou une fonction anonyme peut capturer l’État à partir d’une *fonction anonyme statique* englobante, mais ne peut pas capturer l’État en dehors de la *fonction anonyme statique* englobante.

La suppression du `static` modificateur d’une fonction anonyme dans un programme valide ne modifie pas la signification du programme.

Expression conditionnelle Target-Typed

Article • 13/05/2021

Conversion d'expression conditionnelle

Pour une expression conditionnelle `c ? e1 : e2` , lorsque

1. Il n'existe aucun type commun pour `e1` et `e2` , ou
2. pour lequel un type commun existe, mais l'une des expressions `e1` ou n' `e2` a pas de conversion implicite vers ce type

Nous définissons une nouvelle *conversion d'expression conditionnelle* implicite qui autorise une conversion implicite de l'expression conditionnelle en n'importe quel type `T` pour lequel il existe une conversion de-from-expression de `e1` en `T` et également de à `e2` `T` . Il s'agit d'une erreur si une expression conditionnelle n'a pas de type commun entre `e1` et ou `e2` n'est pas soumise à une *conversion d'expression conditionnelle*.

Meilleure conversion à partir de l'expression

Nous changeons

> ### Meilleure conversion à partir de l'expression
>
> Dans le cas d'une conversion implicite `C1` qui convertit une expression `E` en type `T1` , et d'une conversion implicite `C2` qui convertit une expression `E` en type `T2` , `C1` est une ***meilleure conversion*** que `C2` si `E` ne correspond pas exactement à `T2` et au moins l'un des éléments suivants :
>
> - `E` correspond exactement à `T1` (expression correspondant exactement)
> - `T1` est une meilleure cible de conversion que `T2` (meilleure cible de conversion)

to

> ### Meilleure conversion à partir de l'expression
>
> Dans le cas d'une conversion implicite `C1` qui convertit une expression `E` en type `T1` , et d'une conversion implicite `C2` qui convertit une expression `E` en type `T2` , `C1`

> est une ***meilleure conversion*** que `C2` si `E` ne correspond pas exactement à `T2` et au moins l'un des éléments suivants :
>
> - `E` correspond exactement à `T1` (expression correspondant exactement)
> - **`C1` n'est pas une *conversion d'expression conditionnelle* et `C2` est une * conversion d'expression conditionnelle * * *.
> - `T1` est une meilleure cible de conversion que `T2` (meilleure cible de conversion) * * et soit `C1` et `C2` sont des *conversions d'expressions conditionnelles* , soit aucune n'est une conversion d'expression conditionnelle * * *.

Expression cast

La spécification du langage C# actuel indique

> *Cast_expression* de la forme `(T)E` , où `T` est un *type* et `E` est un *unary_expression*, effectue une conversion explicite (conversions explicites) de la valeur de `E` en type `T` .

En présence de la *conversion d'expression conditionnelle* , il peut y avoir plusieurs conversions possibles de `E` à `T` . Avec l'ajout de la *conversion d'expression conditionnelle*, nous préférons toute conversion en une *conversion d'expression conditionnelle* et n'utilisez la *conversion d'expression conditionnelle* qu'en dernier recours.

Notes de création

La raison de la modification pour une *meilleure conversion à partir de l'expression* consiste à gérer un cas de ce type :

C#

```
M(b ? 1 : 2);

void M(short);
void M(long);
```

Cette approche présente deux petits inconvénients. Tout d'abord, il n'est pas tout à fait identique à l'expression de Switch :

C#

```
M(b ? 1 : 2); // calls M(long)
M(b switch { true => 1, false => 2 }); // calls M(short)
```

Il s'agit toujours d'une modification avec rupture, mais son étendue est moins susceptible d'affecter des programmes réels :

C#

```
M(b ? 1 : 2, 1); // calls M(long, long) without this feature; ambiguous with this feature.

M(short, short);
M(long, long);
```

Cela devient ambigu, car la conversion en `long` est préférable pour le premier argument (parce qu'il n'utilise pas la *conversion d'expression conditionnelle*), mais la conversion en `short` est préférable pour le deuxième argument (car `short` est une *meilleure cible de conversion* que `long`). Cette modification avec rupture semble moins grave, car elle ne modifie pas le comportement d'un programme existant en mode silencieux.

La raison des notes sur l'expression de cast consiste à gérer un cas de ce type :

C#

```
_ = (short)(b ? 1 : 2);
```

Ce programme utilise actuellement la conversion explicite de `int` en `short` et nous souhaitons conserver le sens du langage actuel de ce programme. La modification serait impossible à observer lors de l'exécution, mais avec le programme suivant, la modification serait observable :

C#

```
_ = (A)(b ? c : d);
```

où `c` est de type `C` , `d` est de type `D` , et il existe une conversion implicite définie par l'utilisateur de `C` en `D` , et une conversion implicite définie par l'utilisateur de `D` en `A` , et une conversion implicite définie par l'utilisateur de `C` en `A` . Si ce code est compilé avant C# 9,0, lorsque `b` a la valeur true, nous convertissons de `c` en en `D` `A` . Si nous utilisons la *conversion d'expression conditionnelle*, lorsque `b` a la valeur true, la conversion de `c` en est `A` exécutée directement, ce qui exécute une autre séquence de code utilisateur.

Par conséquent, nous traitons la *conversion d'expression conditionnelle* en dernier recours dans un cast pour préserver le comportement existant.

Retours covariants

Article • 13/05/2021

Résumé

Prennent en charge les *types de retour covariants*. En particulier, autorisez la substitution d'une méthode à déclarer un type de retour plus dérivé que la méthode qu'elle substitue, et de la même manière pour autoriser la substitution d'une propriété en lecture seule pour déclarer un type plus dérivé. Les déclarations override qui apparaissent dans des types plus dérivés sont requises pour fournir un type de retour au moins aussi spécifique que celui qui apparaît dans les substitutions dans ses types de base. Les appelants de la méthode ou de la propriété recevaient statiquement le type de retour plus affiné à partir d'un appel.

Motivation

Il s'agit d'un modèle courant dans le code selon lequel différents noms de méthode doivent être inventés pour contourner la contrainte de langage que les substitutions doivent retourner au même type que la méthode substituée.

Cela peut être utile dans le modèle de fabrique. Par exemple, dans la base de code Roslyn, nous aurions

C#

```
class Compilation ...
{
    public virtual Compilation WithOptions(Options options)...
}
```

C#

```
class CSharpCompilation : Compilation
{
    public override CSharpCompilation WithOptions(Options options)...
}
```

Conception détaillée

Il s'agit d'une spécification pour les types de retour covariants en C#. Notre intention est de permettre à la substitution d'une méthode de retourner un type de retour plus dérivé que la méthode qu'elle substitue, et de la même façon d'autoriser la substitution d'une propriété en lecture seule pour retourner un type de retour plus dérivé. Les appelants de la méthode ou de la propriété recevaient statiquement le type de retour plus affiné à partir d'un appel, et les substitutions apparaissant dans des types plus dérivés seraient nécessaires pour fournir un type de retour au moins aussi spécifique que celui qui apparaît dans les substitutions dans ses types de base.

Substitution de méthode de classe

Contrainte existante sur les méthodes de substitution de classe

> - La méthode de substitution et la méthode de base substituée ont le même type de retour.

est modifié en

> - La méthode override doit avoir un type de retour convertible par une conversion d'identité ou (si la méthode a une valeur retournée et non un retour Ref) une conversion de référence implicite vers le type de retour de la méthode de base substituée.

Et les conditions supplémentaires suivantes sont ajoutées à cette liste :

> - La méthode de substitution doit avoir un type de retour convertible par une conversion d'identité ou (si la méthode a une valeur retournée et non un retour de référence) une conversion de référence implicite vers le type de retour de chaque substitution de la méthode de base substituée qui est déclarée dans un type de base (direct ou indirect) de la méthode de substitution.
> - Le type de retour de la méthode de substitution doit être au moins aussi accessible que la méthode de substitution (domaines d'accessibilité).

Cette contrainte permet à une méthode de substitution dans une `private` classe d'avoir un `private` type de retour. Toutefois, une `public` méthode de substitution dans un `public` type doit avoir un `public` type de retour.

Substitution de la propriété de classe et de l'indexeur

Contrainte existante sur les propriétés de substitution de classe

> Une déclaration de propriété de substitution doit spécifier exactement les mêmes modificateurs d'accessibilité et nom que la propriété héritée, et il doit y avoir une conversion d'identité ~~entre le type de la substitution et la propriété héritée~~. Si la propriété héritée n'a qu'un seul accesseur (autrement dit, si la propriété héritée est en lecture seule ou en écriture seule), la propriété de substitution doit inclure uniquement cet accesseur. Si la propriété héritée inclut des accesseurs (c'est-à-dire que si la propriété héritée est en lecture-écriture), la propriété de substitution peut inclure un seul accesseur ou les deux accesseurs.

est modifié en

> Une déclaration de propriété de substitution doit spécifier exactement les mêmes modificateurs d'accessibilité et Name que la propriété héritée, et il doit y avoir une conversion d'identité **ou (si la propriété héritée est en lecture seule et a une valeur Return-not a ref Return) une conversion de référence implicite du type de la propriété de substitution vers le type de la propriété héritée**. Si la propriété héritée n'a qu'un seul accesseur (autrement dit, si la propriété héritée est en lecture seule ou en écriture seule), la propriété de substitution doit inclure uniquement cet accesseur. Si la propriété héritée inclut des accesseurs (c'est-à-dire que si la propriété héritée est en lecture-écriture), la propriété de substitution peut inclure un seul accesseur ou les deux accesseurs. **Le type de la propriété de substitution doit être au moins aussi accessible que la propriété de substitution (domaines d'accessibilité).**

Le reste de la spécification préliminaire ci-dessous propose une extension supplémentaire des retours covariants de méthodes d'interface à prendre en compte ultérieurement.

Substitution de la méthode d'interface, de la propriété et de l'indexeur

En ajoutant aux genres de membres autorisés dans une interface avec l'ajout de la fonctionnalité DIM en C# 8,0, nous ajoutons davantage de prise en charge des membres avec les `override` retours covariants. Celles-ci suivent les règles des `override` membres telles qu'elles sont spécifiées pour les classes, avec les différences suivantes :

Le texte suivant dans les classes :

> La méthode substituée par une déclaration override est connue sous le nom de ***méthode de base substituée***. Pour une méthode de substitution `M` déclarée dans une classe `C` , la méthode de base substituée est déterminée en examinant chaque classe de base de `C` , en commençant par la classe de base directe de `C` et en continuant avec chaque classe de base directe, jusqu'à ce que dans un type de classe de base donné, au moins une méthode accessible ait la même signature que `M` après la substitution des arguments de type.

reçoit la spécification correspondante pour les interfaces :

> La méthode substituée par une déclaration override est connue sous le nom de **méthode de base substituée** _. Pour une méthode de substitution `M` déclarée dans une interface `I` , la méthode de base substituée est déterminée en examinant chaque interface de base directe ou indirecte de `I` , en collectant le jeu d'interfaces déclarant une méthode accessible qui a la même signature qu' `M` après la substitution des arguments de type. Si cet ensemble d'interfaces a un _most type dérivé *, vers lequel il existe une conversion d'identité ou de référence implicite à partir de chaque type de cet ensemble, et que ce type contient une déclaration de méthode unique, il s'agit de la *méthode de base substituée*.

Nous autorisons de manière similaire les `override` Propriétés et les indexeurs dans les interfaces comme spécifié pour les classes dans les *accesseurs 15.7.6 Virtual, sealed, override et abstract*.

Recherche de nom

La recherche de nom en présence de `override` déclarations de classe modifie actuellement le résultat de la recherche de nom en imposant sur les détails de membre trouvés à partir de la déclaration la plus dérivée `override` dans la hiérarchie de classes à partir du type du qualificateur de l'identificateur (ou `this` en l'absence de qualificateur). Par exemple, dans *12.6.2.2 paramètres correspondants* , nous avons

> Pour les méthodes et les indexeurs virtuels définis dans les classes, la liste de paramètres est choisie à partir de la première déclaration ou substitution du membre de fonction trouvé lors du démarrage avec le type statique du récepteur, et la recherche dans ses classes de base.

pour cela, nous ajoutons

> Pour les méthodes et les indexeurs virtuels définis dans les interfaces, la liste de paramètres est prélevée à partir de la déclaration ou de la substitution du membre de fonction trouvé dans le type le plus dérivé parmi les types contenant la déclaration de substitution de la fonction membre. Il s'agit d'une erreur au moment de la compilation si aucun type unique n'existe.

Pour le type de résultat d'un accès à une propriété ou un indexeur, le texte existant

> - Si I identifie une propriété d'instance, le résultat est un accès à une propriété avec une expression d'instance associée de E et un type associé qui est le type de la propriété. Si T est un type de classe, le type associé est choisi dans la première déclaration ou la substitution de la propriété trouvée lors du démarrage de T, et la recherche dans ses classes de base.

est augmenté avec

> Si T est un type interface, le type associé est choisi dans la déclaration ou la substitution de la propriété trouvée dans le le plus dérivé de T ou ses interfaces de base directes ou indirectes. Il s'agit d'une erreur au moment de la compilation si aucun type unique n'existe.

Une modification similaire doit être apportée à l'accès à l' *indexeur 12.7.7.3*

Dans les *expressions d'appel 12.7.6* , nous enrichissons le texte existant

> - Sinon, le résultat est une valeur, avec un type associé du type de retour de la méthode ou du délégué. Si l'appel est d'une méthode d'instance et que le récepteur est d'un type de classe T, le type associé est choisi dans la première déclaration ou la substitution de la méthode trouvée lors du démarrage avec T et la recherche dans ses classes de base.

par

> Si l'appel est d'une méthode d'instance et que le récepteur est d'un type d'interface T, le type associé est choisi dans la déclaration ou la substitution de la méthode trouvée dans l'interface la plus dérivée parmi T et ses interfaces de base directes et indirectes. Il s'agit d'une erreur au moment de la compilation si aucun type unique n'existe.

Implémentations d'interface implicites

Cette section de la spécification

> Dans le cadre du mappage d'interface, un membre de classe `A` correspond à un membre d'interface `B` lorsque :
>
> - `A` et `B` sont des méthodes, et le nom, le type et les listes de paramètres formels de `A` et `B` sont identiques.
> - `A` et `B` sont des propriétés, le nom et le type de `A` et `B` sont identiques et `A` ont les mêmes accesseurs que `B` (`A` est autorisé à avoir des accesseurs supplémentaires s'il ne s'agit pas d'une implémentation de membre d'interface explicite).
> - `A` et `B` sont des événements, et le nom et le type de `A` et `B` sont identiques.
> - `A` et `B` sont des indexeurs, les listes de paramètres de type et formels de `A` et `B` sont identiques, et `A` ont les mêmes accesseurs que `B` (`A` est autorisé à avoir des accesseurs supplémentaires s'il ne s'agit pas d'une implémentation de membre d'interface explicite).

est modifié comme suit :

> Dans le cadre du mappage d'interface, un membre de classe `A` correspond à un membre d'interface `B` lorsque :
>
> - `A` et `B` sont des méthodes, et les listes de noms et de paramètres formels de `A` et `B` sont identiques, et le type de retour de `A` est convertible en type de retour `B` via une identité de conversion de référence implicite vers le type de retour de `B` .
> - `A` et `B` sont des propriétés, le nom de `A` et `B` sont identiques, `A` a les mêmes accesseurs que `B` (`A` est autorisé à avoir des accesseurs supplémentaires s'il ne s'agit pas d'une implémentation de membre d'interface explicite) et le type de `A` est convertible en type de retour `B` via une conversion d'identité ou, si `A` est une propriété ReadOnly, une conversion de référence implicite.
> - `A` et `B` sont des événements, et le nom et le type de `A` et `B` sont identiques.
> - `A` et `B` sont des indexeurs, les listes de paramètres formels de `A` et `B` sont identiques, `A` ont les mêmes accesseurs que `B` (`A` est autorisé à avoir des accesseurs supplémentaires s'il ne s'agit pas d'une implémentation de membre d'interface explicite) et le type de `A` est convertible en type de retour `B` via une conversion d'identité ou, si `A` est un indexeur ReadOnly, une conversion de référence implicite.

Il s'agit techniquement d'une modification avec rupture, car le programme ci-dessous affiche « C1 ». M « aujourd'hui, mais cela imprimerait «C2 ». M» sous la révision proposée.

```c#
using System;

interface I1 { object M(); }
class C1 : I1 { public object M() { return "C1.M"; } }
class C2 : C1, I1 { public new string M() { return "C2.M"; } }
class Program
{
    static void Main()
    {
        I1 i = new C2();
        Console.WriteLine(i.M());
    }
}
```

En raison de cette modification avec rupture, nous pouvons envisager de ne pas prendre en charge les types de retour covariants sur les implémentations implicites.

Contraintes sur l'implémentation de l'interface

Nous aurons besoin d'une règle qui indique qu'une implémentation d'interface explicite doit déclarer un type de retour non moins dérivé que le type de retour déclaré dans une substitution de ses interfaces de base.

Implications en matière de compatibilité des API

TBD

Problèmes ouverts

La spécification n'indique pas comment l'appelant obtient le type de retour le plus affiné. Vraisemblablement, cela se fait de manière similaire à la façon dont les appelants obtiennent les spécifications de paramètre de substitution les plus dérivées.

Si nous disposons des interfaces suivantes :

```C#
interface I1 { I1 M(); }
interface I2 { I2 M(); }
```

```
interface I3: I1, I2 { override I3 M(); }
```

Notez que dans `I3` , les méthodes `I1.M()` et `I2.M()` ont été fusionnées. Lors de l'implémentation de `I3` , il est nécessaire de les implémenter ensemble.

En général, nous avons besoin d'une implémentation explicite pour faire référence à la méthode d'origine. La question est, dans une classe

C#

```
class C : I1, I2, I3
{
    C IN.M();
}
```

Qu'est-ce que cela signifie ? Que dois - je faire ?

Je vous suggère de permettre l'implémentation `I1.M` `I2.M` de ou (mais pas les deux), et de le considérer comme une implémentation des deux.

Inconvénients

- [] Chaque modification de langue doit payer pour elle-même.
- [] Nous devons nous assurer que les performances sont raisonnables, même dans le cas des hiérarchies d'héritage profond
- [] Nous devons nous assurer que les artefacts de la stratégie de traduction n'affectent pas la sémantique de la langue, même lors de la consommation du nouveau langage intermédiaire à partir des compilateurs anciens.

Autres solutions

Nous pourrions assouplir légèrement les règles de langage pour autoriser, dans la source,

C#

```
abstract class Cloneable
{
    public abstract Cloneable Clone();
}

class Digit : Cloneable
{
    public override Cloneable Clone()
```

```
    {
        return this.Clone();
    }

    public new Digit Clone() // Error: 'Digit' already defines a member
called 'Clone' with the same parameter types
    {
        return this;
    }
}
```

Questions non résolues

- [] Comment les API qui ont été compilées pour utiliser cette fonctionnalité fonctionnent-elles dans des versions antérieures du langage ?

Réunions de conception

- une discussion sur https://github.com/dotnet/roslyn/issues/357 .
- https://github.com/dotnet/csharplang/blob/master/meetings/2020/LDM-2020-01-08.md
- Discussion hors connexion vers une décision pour prendre en charge la substitution des méthodes de classe uniquement en C# 9,0.

`GetEnumerator` Prise en charge des extensions pour les `foreach` boucles.

Article • 13/05/2021

Résumé

Autoriser les boucles Foreach à reconnaître une méthode d'extension GetEnumerator qui, sinon, répond au modèle foreach, et effectue une boucle sur l'expression lorsqu'il s'agit d'une erreur.

Motivation

Foreach sera incorporé avec la manière dont les autres fonctionnalités en C# sont implémentées, notamment Async et la déconstruction basée sur des modèles.

Conception détaillée

La modification des spécifications est relativement simple. Nous modifions `The foreach statement` la section à ce texte :

> Le traitement au moment de la compilation d'une instruction foreach détermine d'abord le * type de **collection** , *le type d' *énumérateur* et le type d'élément _ de l'expression.* and _ *element type** of the expression. Cette détermination se déroule comme suit :
>
> - Si le type `X` d' *expression* est un type tableau, il existe une conversion de référence implicite de `X` en `IEnumerable` interface (puisque `System.Array` implémente cette interface). Le ***type de collection*** _ *est l'* `IEnumerable` *interface, le _type d'énumérateur_ est l'* `IEnumerator` *interface et le type d'élément** _ est le type d'élément du type tableau `X` .
>
> - Si le type `X` d' *expression* est `dynamic` , il y a une conversion implicite d' *expression* en `IEnumerable` interface (conversions dynamiques implicites). Le ***type de collection*** _ *est l'* `IEnumerable` *interface et le _type d'énumérateur*_* est l' `IEnumerator` interface. Si l' `var` identificateur est fourni en tant que _local_variable_type *, le ***type d'élément*** est `dynamic` , sinon, `object` .
>
> - Sinon, déterminez si le type `X` a une `GetEnumerator` méthode appropriée :

- Effectuez une recherche de membre sur le type `X` avec l'identificateur `GetEnumerator` et aucun argument de type. Si la recherche de membre ne produit pas de correspondance, si elle génère une ambiguïté ou si elle produit une correspondance qui n'est pas un groupe de méthodes, recherchez une interface énumérable comme décrit ci-dessous. Il est recommandé d'émettre un avertissement si la recherche de membres produit tout sauf un groupe de méthodes ou aucune correspondance.
- Exécutez la résolution de surcharge à l'aide du groupe de méthodes résultant et d'une liste d'arguments vide. Si la résolution de surcharge entraîne l'absence de méthodes applicables, provoque une ambiguïté ou produit une seule meilleure méthode, mais que cette méthode est statique ou non publique, recherchez une interface énumérable comme décrit ci-dessous. Il est recommandé d'émettre un avertissement si la résolution de surcharge produit tout sauf une méthode d'instance publique non ambiguë ou aucune méthode applicable.
- Si le type `E` de retour de la `GetEnumerator` méthode n'est pas un type de classe, de struct ou d'interface, une erreur est générée et aucune autre étape n'est prise.
- La recherche de membres est effectuée sur `E` avec l'identificateur `Current` et aucun argument de type. Si la recherche de membre ne produit aucune correspondance, le résultat est une erreur, ou le résultat est tout sauf une propriété d'instance publique qui autorise la lecture, une erreur est générée et aucune autre étape n'est effectuée.
- La recherche de membres est effectuée sur `E` avec l'identificateur `MoveNext` et aucun argument de type. Si la recherche de membre ne produit aucune correspondance, le résultat est une erreur, ou le résultat est tout sauf un groupe de méthodes, une erreur est générée et aucune autre étape n'est prise.
- La résolution de surcharge est effectuée sur le groupe de méthodes avec une liste d'arguments vide. Si la résolution de surcharge entraîne l'absence de méthodes applicables, entraîne une ambiguïté ou produit une seule meilleure méthode, mais cette méthode est statique ou non publique, ou son type de retour n'est pas `bool` , une erreur est générée et aucune autre étape n'est prise.
- Le ***type de collection*** *_ est `X` , le _type d'énumérateur_ est `E` , et le type d'élément** _ est le type de la `Current` propriété.

- Sinon, recherchez une interface énumérable :
  - Si parmi tous les types `Ti` pour lesquels il existe une conversion implicite de `X` en `IEnumerable<Ti>` , il existe un type unique `T` , qui n' `T` est pas `dynamic`

et pour tout l'autre, `Ti` il existe une conversion implicite de `IEnumerable<T>` en `IEnumerable<Ti>` , puis le * type de **collection** _ est l'interface `IEnumerable<T>` , le type d' **énumérateur** est l'interface `IEnumerator<T>` et le type d'*élément** _ *est* `T` ._ is the interface `IEnumerator<T>`, and the _ *element type** is `T`.

- Sinon, s'il existe plusieurs types de ce type `T` , une erreur est générée et aucune autre étape n'est effectuée.
- Sinon, s'il existe une conversion implicite de `X` en `System.Collections.IEnumerable` interface, le ***type de collection*** _ *est cette interface, le _type d'énumérateur** _ est l'interface `System.Collections.IEnumerator` et le *type d'élément* _ est `object` .

- Sinon, déterminez si le type'X'a une `GetEnumerator` méthode d'extension appropriée :
  - Effectue une recherche de méthode d'extension sur le type `X` avec l'identificateur `GetEnumerator` . Si la recherche de membre ne produit pas de correspondance, si elle génère une ambiguïté ou si elle produit une correspondance qui n'est pas un groupe de méthodes, une erreur est générée et aucune autre étape n'est effectuée. Il est recommandé qu'un avertissement soit émis si la recherche de membres produit n'importe quoi, sauf un groupe de méthodes ou aucune correspondance.
  - Exécutez la résolution de surcharge à l'aide du groupe de méthodes résultant et d'un argument unique de type `X` . Si la résolution de surcharge ne produit aucune méthode applicable, provoque une ambiguïté ou produit une seule meilleure méthode mais que cette méthode n'est pas accessible, une erreur est générée. aucune autre étape n'est effectuée.
    - Cette résolution permet au premier argument d'être passé par Ref si `X` est un type struct et le genre Ref est `in` .
  - Si le type `E` de retour de la `GetEnumerator` méthode n'est pas un type de classe, de struct ou d'interface, une erreur est générée et aucune autre étape n'est prise.
  - La recherche de membres est effectuée sur `E` avec l'identificateur `Current` et aucun argument de type. Si la recherche de membre ne produit aucune correspondance, le résultat est une erreur, ou le résultat est tout sauf une propriété d'instance publique qui autorise la lecture, une erreur est générée et aucune autre étape n'est effectuée.
  - La recherche de membres est effectuée sur `E` avec l'identificateur `MoveNext` et aucun argument de type. Si la recherche de membre ne produit aucune correspondance, le résultat est une erreur, ou le résultat est tout sauf un

groupe de méthodes, une erreur est générée et aucune autre étape n'est prise.
    - La résolution de surcharge est effectuée sur le groupe de méthodes avec une liste d'arguments vide. Si la résolution de surcharge entraîne l'absence de méthodes applicables, entraîne une ambiguïté ou produit une seule meilleure méthode, mais cette méthode est statique ou non publique, ou son type de retour n'est pas `bool` , une erreur est générée et aucune autre étape n'est prise.
    - Le ***type de collection*** _ *est* `X` *, le _type d'énumérateur_ est* `E` *, et le type d'élément** _ est le type de la `Current` propriété.

- Dans le cas contraire, une erreur est générée et aucune autre étape n'est prise.

Pour `await foreach` , les règles sont modifiées de façon similaire. La seule modification requise pour cette spécification consiste à supprimer la `Extension methods do not contribute.` ligne de la description, car le reste de cette spécification est basé sur les règles ci-dessus avec des noms différents substitués pour les méthodes de modèle.

Inconvénients

Chaque modification ajoute une complexité supplémentaire à la langue, ce qui peut permettre d'effectuer des tâches qui n'ont pas été conçues pour être `foreach` `foreach` Ed, comme `Range` .

Autres solutions

Ne rien faire.

Questions non résolues

Aucun à ce stade.

Paramètres d'abandon lambda

Article • 13/05/2021

Résumé

Autorisez l'utilisation des éléments ignorés (`_`) comme paramètres des expressions lambda et des méthodes anonymes. Par exemple :

- expressions lambda : `(_, _) => 0` , `(int _, int _) => 0`
- méthodes anonymes : `delegate(int _, int _) { return 0; }`

Motivation

Les paramètres inutilisés n'ont pas besoin d'être nommés. L'objectif des éléments ignorés est clair, c'est-à-dire qu'ils sont inutilisés/ignorés.

Conception détaillée

Paramètres de méthode Dans la liste des paramètres d'une méthode lambda ou anonyme avec plusieurs paramètres nommés `_` , ces paramètres sont ignorés. Remarque : si un seul paramètre est nommé `_` , il s'agit d'un paramètre régulier pour des raisons de compatibilité descendante.

Les paramètres d'abandon n'introduisent aucun nom dans les étendues. Notez que cela implique qu'ils n'entraînent pas le `_` masquage des noms de traits de soulignement.

Noms simples Si `K` est égal à zéro et que le *simple_name* apparaît dans un *bloc* et si l'espace de déclaration de variable locale (déclarations) du *bloc*(ou du *bloc* englobant) contient une variable locale, un paramètre (à l'exception des paramètres d'abandon) ou une constante portant `I` le nom, le *simple_name* fait référence à cette variable locale, ce paramètre ou cette constante et est classée comme variable ou valeur

Étendues À l'exception des paramètres d'abandon, la portée d'un paramètre déclaré dans *une lambda_expression* (expressions de fonction anonymes) est la *anonymous_function_body* de cette *lambda_expression* à l'exception des paramètres d'abandon, la portée d'un paramètre déclaré dans une *anonymous_method_expression* (expressions de fonction anonymes) est le *bloc* de cette *anonymous_method_expression*.

Sections relatives aux spécifications

- Paramètres correspondants

Attributs sur des fonctions locales

Article • 13/05/2021

Attributs

Les déclarations de fonction locale sont désormais autorisées à avoir des attributs. Les paramètres et les paramètres de type sur les fonctions locales sont également autorisés à avoir des attributs.

Les attributs ayant une signification spécifiée lorsqu'ils sont appliqués à une méthode, à ses paramètres ou à ses paramètres de type ont la même signification lorsqu'ils sont appliqués à une fonction locale, à ses paramètres ou à ses paramètres de type, respectivement.

Une fonction locale peut être rendue conditionnelle dans le même sens qu'une méthode conditionnelle en la décorant avec un `[ConditionalAttribute]` . Une fonction locale conditionnelle doit également être `static` . Toutes les restrictions sur les méthodes conditionnelles s'appliquent également aux fonctions locales conditionnelles, y compris que le type de retour doit être `void` .

Externes

Le `extern` modificateur est désormais autorisé sur les fonctions locales. Cela rend la fonction locale externe dans le même sens qu'une méthode externe.

À l'instar d'une méthode externe, le *corps de fonction locale* d'une fonction locale externe doit être un point-virgule. Un *corps de fonction locale* de point-virgule est uniquement autorisé sur une fonction locale externe.

Une fonction locale externe doit également être `static` .

Syntax

La grammaire Functions locale est modifiée comme suit :

```
local-function-header
    : attributes? local-function-modifiers? return-type identifier type-
parameter-list?
        ( formal-parameter-list? ) type-parameter-constraints-clauses
```

```
    ;

local-function-modifiers
    : (async | unsafe | static | extern)*
    ;

local-function-body
    : block
    | arrow-expression-body
    | ';'
    ;
```

Entiers de taille native

Article • 13/05/2021

Résumé

Prise en charge des langages pour les types d'entiers signés et non signés de taille native.

La motivation est pour les scénarios d'interopérabilité et pour les bibliothèques de bas niveau.

Conception

Les identificateurs `nint` et `nuint` sont de nouveaux mots clés contextuels qui représentent des types entiers signés et non signés natifs. Les identificateurs sont traités uniquement comme des mots clés lorsque la recherche de nom ne trouve pas de résultat viable à cet emplacement de programme.

C#

```
nint x = 3;
string y = nameof(nuint);
_ = nint.Equals(x, 3);
```

Les types `nint` et `nuint` sont représentés par les types sous-jacents et le compilateur signale les `System.IntPtr` `System.UIntPtr` conversions et les opérations supplémentaires pour ces types comme des ints natifs.

Constantes

Les expressions constantes peuvent être de type `nint` ou `nuint` . Il n'existe pas de syntaxe directe pour les littéraux int natifs. Les casts implicites ou explicites d'autres valeurs de constantes intégrales peuvent être utilisés à la place : `const nint i = (nint)42;` .

`nint` les constantes sont comprises dans la plage [`int.MinValue` , `int.MaxValue`].

`nuint` les constantes sont comprises dans la plage [`uint.MinValue` , `uint.MaxValue`].

Il n'y a aucun `MinValue` `MaxValue` champ ou sur `nint` ou `nuint` , car, à l'exception de `nuint.MinValue` , ces valeurs ne peuvent pas être émises en tant que constantes.

Le repli de constante est pris en charge pour tous les opérateurs unaires { `+` , `-` , `~` } et les opérateurs binaires {,,,,,, `+` `-` `*` `/` `%` `==` `!=` , `<` , `<=` , `>` , `>=` , `&` , `|` , `^` , `<<` , `>>` }. Les opérations de pliage de constante sont évaluées avec des `Int32` `UInt32` opérandes et plutôt que des ints natifs pour un comportement cohérent indépendamment de la plateforme du compilateur. Si l'opération génère une valeur constante en 32 bits, le repli de constante est effectué au moment de la compilation. Dans le cas contraire, l'opération est exécutée au moment de l'exécution et n'est pas considérée comme une constante.

Conversions

Il existe une conversion d'identité entre `nint` et `IntPtr` , et entre `nuint` et `UIntPtr` . Il existe une conversion d'identité entre les types composés qui diffèrent par les ints natifs et les types sous-jacents uniquement : les tableaux, `Nullable<>` , les types construits et les tuples.

Les tableaux ci-dessous couvrent les conversions entre les types spéciaux. (Le langage intermédiaire pour chaque conversion comprend les variantes pour `unchecked` et les `checked` contextes s'ils sont différents.)

Opérande	Cible	Conversion	IL
`object`	`nint`	Unboxing	`unbox`
`void*`	`nint`	PointerToVoid	`conv.i`
`sbyte`	`nint`	ImplicitNumeric	`conv.i`
`byte`	`nint`	ImplicitNumeric	`conv.u`
`short`	`nint`	ImplicitNumeric	`conv.i`
`ushort`	`nint`	ImplicitNumeric	`conv.u`
`int`	`nint`	ImplicitNumeric	`conv.i`
`uint`	`nint`	ExplicitNumeric	`conv.u` / `conv.ovf.u`
`long`	`nint`	ExplicitNumeric	`conv.i` / `conv.ovf.i`
`ulong`	`nint`	ExplicitNumeric	`conv.i` / `conv.ovf.i`
`char`	`nint`	ImplicitNumeric	`conv.i`

Opérande	Cible	Conversion	IL
`float`	`nint`	ExplicitNumeric	`conv.i` / `conv.ovf.i`
`double`	`nint`	ExplicitNumeric	`conv.i` / `conv.ovf.i`
`decimal`	`nint`	ExplicitNumeric	`long decimal.op_Explicit(decimal) conv.i` / `... conv.ovf.i`
`IntPtr`	`nint`	Identité	
`UIntPtr`	`nint`	None	
`object`	`nuint`	Unboxing	`unbox`
`void*`	`nuint`	PointerToVoid	`conv.u`
`sbyte`	`nuint`	ExplicitNumeric	`conv.u` / `conv.ovf.u`
`byte`	`nuint`	ImplicitNumeric	`conv.u`
`short`	`nuint`	ExplicitNumeric	`conv.u` / `conv.ovf.u`
`ushort`	`nuint`	ImplicitNumeric	`conv.u`
`int`	`nuint`	ExplicitNumeric	`conv.u` / `conv.ovf.u`
`uint`	`nuint`	ImplicitNumeric	`conv.u`
`long`	`nuint`	ExplicitNumeric	`conv.u` / `conv.ovf.u`
`ulong`	`nuint`	ExplicitNumeric	`conv.u` / `conv.ovf.u`
`char`	`nuint`	ImplicitNumeric	`conv.u`
`float`	`nuint`	ExplicitNumeric	`conv.u` / `conv.ovf.u`
`double`	`nuint`	ExplicitNumeric	`conv.u` / `conv.ovf.u`
`decimal`	`nuint`	ExplicitNumeric	`ulong decimal.op_Explicit(decimal) conv.u` / `... conv.ovf.u.un`
`IntPtr`	`nuint`	None	
`UIntPtr`	`nuint`	Identité	
Énumération	`nint`	ExplicitEnumeration	
Énumération	`nuint`	ExplicitEnumeration	

Opérande	Cible	Conversion	IL

Opérande	Cible	Conversion	IL
`nint`	`object`	Boxing	`box`
`nint`	`void*`	PointerToVoid	`conv.i`
`nint`	`nuint`	ExplicitNumeric	`conv.u` / `conv.ovf.u`
`nint`	`sbyte`	ExplicitNumeric	`conv.i1` / `conv.ovf.i1`
`nint`	`byte`	ExplicitNumeric	`conv.u1` / `conv.ovf.u1`
`nint`	`short`	ExplicitNumeric	`conv.i2` / `conv.ovf.i2`
`nint`	`ushort`	ExplicitNumeric	`conv.u2` / `conv.ovf.u2`
`nint`	`int`	ExplicitNumeric	`conv.i4` / `conv.ovf.i4`
`nint`	`uint`	ExplicitNumeric	`conv.u4` / `conv.ovf.u4`
`nint`	`long`	ImplicitNumeric	`conv.i8` / `conv.ovf.i8`
`nint`	`ulong`	ExplicitNumeric	`conv.i8` / `conv.ovf.i8`
`nint`	`char`	ExplicitNumeric	`conv.u2` / `conv.ovf.u2`
`nint`	`float`	ImplicitNumeric	`conv.r4`
`nint`	`double`	ImplicitNumeric	`conv.r8`
`nint`	`decimal`	ImplicitNumeric	`conv.i8 decimal decimal.op_Implicit(long)`
`nint`	`IntPtr`	Identité	
`nint`	`UIntPtr`	None	
`nint`	Énumération	ExplicitEnumeration	
`nuint`	`object`	Boxing	`box`
`nuint`	`void*`	PointerToVoid	`conv.u`
`nuint`	`nint`	ExplicitNumeric	`conv.i` / `conv.ovf.i`
`nuint`	`sbyte`	ExplicitNumeric	`conv.i1` / `conv.ovf.i1`
`nuint`	`byte`	ExplicitNumeric	`conv.u1` / `conv.ovf.u1`
`nuint`	`short`	ExplicitNumeric	`conv.i2` / `conv.ovf.i2`
`nuint`	`ushort`	ExplicitNumeric	`conv.u2` / `conv.ovf.u2`

Opérande	Cible	Conversion	IL
`nuint`	`int`	ExplicitNumeric	`conv.i4` / `conv.ovf.i4`
`nuint`	`uint`	ExplicitNumeric	`conv.u4` / `conv.ovf.u4`
`nuint`	`long`	ExplicitNumeric	`conv.i8` / `conv.ovf.i8`
`nuint`	`ulong`	ImplicitNumeric	`conv.u8` / `conv.ovf.u8`
`nuint`	`char`	ExplicitNumeric	`conv.u2` / `conv.ovf.u2.un`
`nuint`	`float`	ImplicitNumeric	`conv.r.un conv.r4`
`nuint`	`double`	ImplicitNumeric	`conv.r.un conv.r8`
`nuint`	`decimal`	ImplicitNumeric	`conv.u8 decimal decimal.op_Implicit(ulong)`
`nuint`	`IntPtr`	None	
`nuint`	`UIntPtr`	Identité	
`nuint`	Énumération	ExplicitEnumeration	

La conversion de `A` en `Nullable<B>` est :

- conversion Nullable implicite s'il existe une conversion d'identité ou une conversion implicite de `A` en `B` ;
- conversion Nullable explicite s'il existe une conversion explicite de `A` en `B` ;
- sinon non valide.

La conversion de `Nullable<A>` en `B` est :

- conversion Nullable explicite s'il existe une conversion d'identité ou une conversion numérique implicite ou explicite de `A` en `B` ;
- sinon non valide.

La conversion de `Nullable<A>` en `Nullable<B>` est :

- conversion d'identité s'il existe une conversion d'identité de `A` en `B` ;
- conversion Nullable explicite s'il existe une conversion numérique implicite ou explicite de `A` en `B` ;
- sinon non valide.

Opérateurs

Les opérateurs prédéfinis sont les suivants. Ces opérateurs sont pris en compte lors de la résolution de surcharge basée sur des règles normales pour les conversions implicites *si au moins l'un des opérandes est de type* `nint` *ou* `nuint`.

(Le langage intermédiaire de chaque opérateur comprend les variantes pour `unchecked` et les `checked` contextes s'ils sont différents.)

Unaire	Signature de l'opérateur	IL
`+`	`nint operator +(nint value)`	`nop`
`+`	`nuint operator +(nuint value)`	`nop`
`-`	`nint operator -(nint value)`	`neg`
`~`	`nint operator ~(nint value)`	`not`
`~`	`nuint operator ~(nuint value)`	`not`

Binary	Signature de l'opérateur	IL
`+`	`nint operator +(nint left, nint right)`	`add` / `add.ovf`
`+`	`nuint operator +(nuint left, nuint right)`	`add` / `add.ovf.un`
`-`	`nint operator -(nint left, nint right)`	`sub` / `sub.ovf`
`-`	`nuint operator -(nuint left, nuint right)`	`sub` / `sub.ovf.un`
`*`	`nint operator *(nint left, nint right)`	`mul` / `mul.ovf`
`*`	`nuint operator *(nuint left, nuint right)`	`mul` / `mul.ovf.un`
`/`	`nint operator /(nint left, nint right)`	`div`
`/`	`nuint operator /(nuint left, nuint right)`	`div.un`
`%`	`nint operator %(nint left, nint right)`	`rem`
`%`	`nuint operator %(nuint left, nuint right)`	`rem.un`
`==`	`bool operator ==(nint left, nint right)`	`beq` / `ceq`
`==`	`bool operator ==(nuint left, nuint right)`	`beq` / `ceq`
`!=`	`bool operator !=(nint left, nint right)`	`bne`
`!=`	`bool operator !=(nuint left, nuint right)`	`bne`
`<`	`bool operator <(nint left, nint right)`	`blt` / `clt`

Binary	Signature de l'opérateur	IL
`<`	`bool operator <(nuint left, nuint right)`	`blt.un` / `clt.un`
`<=`	`bool operator <=(nint left, nint right)`	`ble`
`<=`	`bool operator <=(nuint left, nuint right)`	`ble.un`
`>`	`bool operator >(nint left, nint right)`	`bgt` / `cgt`
`>`	`bool operator >(nuint left, nuint right)`	`bgt.un` / `cgt.un`
`>=`	`bool operator >=(nint left, nint right)`	`bge`
`>=`	`bool operator >=(nuint left, nuint right)`	`bge.un`
`&`	`nint operator &(nint left, nint right)`	`and`
`&`	`nuint operator &(nuint left, nuint right)`	`and`
`|`	`nint operator |(nint left, nint right)`	`or`
`|`	`nuint operator |(nuint left, nuint right)`	`or`
`^`	`nint operator ^(nint left, nint right)`	`xor`
`^`	`nuint operator ^(nuint left, nuint right)`	`xor`
`<<`	`nint operator <<(nint left, int right)`	`shl`
`<<`	`nuint operator <<(nuint left, int right)`	`shl`
`>>`	`nint operator >>(nint left, int right)`	`shr`
`>>`	`nuint operator >>(nuint left, int right)`	`shr.un`

Pour certains opérateurs binaires, les opérateurs IL prennent en charge des types d'opérandes supplémentaires (consultez la table de type d'opérande ECMA-335 III. 1.5). Toutefois, l'ensemble de types d'opérandes pris en charge par C# est limité pour des raisons de simplicité et de cohérence avec les opérateurs existants dans le langage.

Versions levées des opérateurs, où les arguments et les types de retour sont `nint?` et `nuint?` , sont pris en charge.

Les opérations d'assignation composée `x op= y` où `x` ou `y` sont des ints natifs suivent les mêmes règles qu'avec d'autres types primitifs avec des opérateurs prédéfinis. Plus précisément, l'expression est liée, `x = (T)(x op y)` où `T` est le type de `x` et où `x` est évalué une seule fois.

Les opérateurs de décalage doivent masquer le nombre de bits de décalage-à 5 bits si `sizeof(nint)` est 4, et à 6 bits si `sizeof(nint)` est 8. (consultez opérateurs de décalage dans la spécification C#).

Le compilateur C# 9 signale les erreurs de liaison à des opérateurs entiers natifs prédéfinis lors de la compilation avec une version de langage antérieure, mais autorise l'utilisation de conversions prédéfinies vers et à partir d'entiers natifs.

`csc -langversion:9 -t:library A.cs`

C#

```
public class A
{
    public static nint F;
}
```

`csc -langversion:8 -r:A.dll B.cs`

C#

```
class B : A
{
    static void Main()
    {
        F = F + 1; // error: nint operator+ not available with -
langversion:8
        F = (System.IntPtr)F + 1; // ok
    }
}
```

Dynamique

Les conversions et les opérateurs sont synthétisés par le compilateur et ne font pas partie des types sous-jacents `IntPtr` et `UIntPtr` . Par conséquent, les conversions et les opérateurs ne *sont pas disponibles* à partir du Binder d'exécution pour `dynamic` .

C#

```
nint x = 2;
nint y = x + x; // ok
dynamic d = x;
nint z = d + x; // RuntimeBinderException: '+' cannot be applied
'System.IntPtr' and 'System.IntPtr'
```

Membres de type

Le seul constructeur pour `nint` ou `nuint` est le constructeur sans paramètre.

Les membres suivants de `System.IntPtr` et de `System.UIntPtr` *sont explicitement exclus* de `nint` ou `nuint` :

C#

```
// constructors
// arithmetic operators
// implicit and explicit conversions
public static readonly IntPtr Zero; // use 0 instead
public static int Size { get; }     // use sizeof() instead
public static IntPtr Add(IntPtr pointer, int offset);
public static IntPtr Subtract(IntPtr pointer, int offset);
public int ToInt32();
public long ToInt64();
public void* ToPointer();
```

Les membres restants de `System.IntPtr` et `System.UIntPtr` *sont inclus implicitement* dans `nint` et `nuint` . Pour .NET Framework 4.7.2 :

C#

```
public override bool Equals(object obj);
public override int GetHashCode();
public override string ToString();
public string ToString(string format);
```

Les interfaces implémentées par `System.IntPtr` et `System.UIntPtr` *sont implicitement incluses* dans `nint` et `nuint` , avec les occurrences des types sous-jacents remplacés par les types d'entiers natifs correspondants. Par exemple `IntPtr` , si implémente `ISerializable, IEquatable<IntPtr>, IComparable<IntPtr>` , `nint` implémente `ISerializable, IEquatable<nint>, IComparable<nint>` .

Substitution, masquage et implémentation

`nint` et `System.IntPtr` , et `nuint` et `System.UIntPtr` , sont considérés comme équivalents pour la substitution, le masquage et l'implémentation.

Les surcharges ne peuvent pas différer par `nint` and `System.IntPtr` , and `nuint` et `System.UIntPtr` , seul. Les remplacements et les implémentations peuvent différer par `nint` and `System.IntPtr` , or `nuint` et `System.UIntPtr` , seul. Les méthodes masquent

les autres méthodes qui diffèrent par `nint` and `System.IntPtr` , ou `nuint` et `System.UIntPtr` , seul.

Divers

`nint` les `nuint` expressions et utilisées en tant qu'index de tableau sont émises sans conversion.

C#

```
static object GetItem(object[] array, nint index)
{
    return array[index]; // ok
}
```

`nint` et `nuint` peuvent être utilisés comme `enum` type de base.

C#

```
enum E : nint // ok
{
}
```

Les lectures et écritures sont atomiques pour les types `nint` , `nuint` et `enum` avec le type de base `nint` ou `nuint` .

Les champs peuvent être marqués `volatile` pour les types `nint` et `nuint` . L' ECMA-334 15.5.4 n'inclut pas `enum` avec le type de base `System.IntPtr` ou `System.UIntPtr` toutefois.

`default(nint)` et `new nint()` sont équivalents à `(nint)0` .

`typeof(nint)` a la valeur `typeof(IntPtr)`.

`sizeof(nint)` est pris en charge, mais nécessite une compilation dans un contexte unsafe (comme `sizeof(IntPtr)`). La valeur n'est pas une constante au moment de la compilation. `sizeof(nint)` est implémenté comme `sizeof(IntPtr)` au lieu de `IntPtr.Size` .

Diagnostics du compilateur pour les références de type impliquant `nint` ou `nuint` signalant ou `nint` `nuint` plutôt que `IntPtr` ou `UIntPtr` .

Métadonnées

`nint` et `nuint` sont représentés dans les métadonnées en tant que `System.IntPtr` et `System.UIntPtr` .

Les références de type qui incluent `nint` ou `nuint` sont émises avec un `System.Runtime.CompilerServices.NativeIntegerAttribute` pour indiquer quelles parties de la référence de type sont des ints natifs.

C#

```
namespace System.Runtime.CompilerServices
{
    [AttributeUsage(
        AttributeTargets.Class |
        AttributeTargets.Event |
        AttributeTargets.Field |
        AttributeTargets.GenericParameter |
        AttributeTargets.Parameter |
        AttributeTargets.Property |
        AttributeTargets.ReturnValue,
        AllowMultiple = false,
        Inherited = false)]
    public sealed class NativeIntegerAttribute : Attribute
    {
        public NativeIntegerAttribute()
        {
            TransformFlags = new[] { true };
        }
        public NativeIntegerAttribute(bool[] flags)
        {
            TransformFlags = flags;
        }
        public readonly bool[] TransformFlags;
    }
}
```

L'encodage des références de type avec `NativeIntegerAttribute` est couvert dans NativeIntegerAttribute.MD .

Autres solutions

Une alternative à l'approche « effacement de type » ci-dessus consiste à introduire de nouveaux types : `System.NativeInt` et `System.NativeUInt` .

C#

```
public readonly struct NativeInt
{
```

```
    public IntPtr Value;
}
```

Les types distincts permettent une surcharge différente de `IntPtr` et autorisent l'analyse distincte et `ToString()` . Toutefois, il y aurait plus de travail pour le CLR pour gérer efficacement ces types, ce qui a pour effet d'entraîner l'objectif principal de l'efficacité des fonctionnalités. Et l'interopérabilité avec un code int natif existant qui utilise `IntPtr` serait plus difficile.

Une autre solution consiste à ajouter une prise en charge native de int pour `IntPtr` dans le Framework, mais sans prise en charge spécifique du compilateur. Toutes les nouvelles conversions et opérations arithmétiques sont prises en charge automatiquement par le compilateur. Mais le langage ne fournirait pas de mots clés, de constantes ou d' `checked` opérations.

Réunions de conception

- https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-05-26.md
- https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-06-13.md
- https://github.com/dotnet/csharplang/blob/master/meetings/2017/LDM-2017-07-05.md#native-int-and-intptr-operators
- https://github.com/dotnet/csharplang/blob/master/meetings/2019/LDM-2019-10-23.md
- https://github.com/dotnet/csharplang/blob/master/meetings/2020/LDM-2020-03-25.md

Pointeurs fonction

Article • 13/05/2021

Résumé

Cette proposition fournit des constructions de langage qui exposent des opcodes IL qui ne sont pas actuellement accessibles efficacement, ou du tout, en C# : `ldftn` et `calli` . Ces opcodes IL peuvent être importants dans du code haute performance et les développeurs doivent disposer d'un moyen efficace pour y accéder.

Motivation

Les motivations et l'arrière-plan de cette fonctionnalité sont décrits dans le problème suivant (comme c'est une implémentation potentielle de la fonctionnalité) :

https://github.com/dotnet/csharplang/issues/191

Il s'agit d'une proposition de conception alternative aux intrinsèques du compilateur

Conception détaillée

Pointeurs fonction

Le langage autorisera la déclaration des pointeurs de fonction à l'aide de la `delegate*` syntaxe. La syntaxe complète est décrite en détail dans la section suivante, mais elle est destinée à ressembler à la syntaxe utilisée par `Func` et aux `Action` déclarations de type.

C#

```
unsafe class Example {
    void Example(Action<int> a, delegate*<int, void> f) {
        a(42);
        f(42);
    }
}
```

Ces types sont représentés à l'aide du type de pointeur de fonction, comme indiqué dans la norme ECMA-335. Cela signifie que l'appel d'un `delegate*` utilisera `calli` lorsque l'appel d'un `delegate` sera `callvirt` utilisé sur la `Invoke` méthode. Toutefois, la syntaxe de l'appel est identique pour les deux constructions.

La définition ECMA-335 des pointeurs de méthode comprend la Convention d'appel dans le cadre de la signature de type (section 7,1). La Convention d'appel par défaut sera `managed` . Les conventions d'appel non managées peuvent être spécifiées en plaçant un `unmanaged` mot clé après la `delegate*` syntaxe, qui utilise la valeur par défaut de la plateforme Runtime. Les conventions non managées spécifiques peuvent ensuite être spécifiées entre crochets au `unmanaged` mot clé en spécifiant tout type commençant par `CallConv` dans l' `System.Runtime.CompilerServices` espace de noms, sans le `CallConv` préfixe. Ces types doivent provenir de la bibliothèque principale du programme, et l'ensemble des combinaisons valides dépend de la plateforme.

C#

```
//This method has a managed calling convention. This is the same as leaving
the managed keyword off.
delegate* managed<int, int>;

// This method will be invoked using whatever the default unmanaged calling
convention on the runtime
// platform is. This is platform and architecture dependent and is
determined by the CLR at runtime.
delegate* unmanaged<int, int>;

// This method will be invoked using the cdecl calling convention
// Cdecl maps to System.Runtime.CompilerServices.CallConvCdecl
delegate* unmanaged[Cdecl] <int, int>;

// This method will be invoked using the stdcall calling convention, and
suppresses GC transition
// Stdcall maps to System.Runtime.CompilerServices.CallConvStdcall
// SuppressGCTransition maps to
System.Runtime.CompilerServices.CallConvSuppressGCTransition
delegate* unmanaged[Stdcall, SuppressGCTransition] <int, int>;
```

Les conversions entre `delegate*` les types sont effectuées en fonction de leur signature, y compris la Convention d'appel.

C#

```
unsafe class Example {
    void Conversions() {
        delegate*<int, int, int> p1 = ...;
        delegate* managed<int, int, int> p2 = ...;
        delegate* unmanaged<int, int, int> p3 = ...;

        p1 = p2; // okay p1 and p2 have compatible signatures
        Console.WriteLine(p2 == p1); // True
        p2 = p3; // error: calling conventions are incompatible
    }
}
```

Un `delegate*` type est un type pointeur, ce qui signifie qu'il possède toutes les fonctionnalités et restrictions d'un type pointeur standard :

- Valide uniquement dans un `unsafe` contexte.
- Les méthodes qui contiennent un `delegate*` paramètre ou un type de retour ne peuvent être appelées qu'à partir d'un `unsafe` contexte.
- Ne peut pas être converti en `object` .
- Ne peut pas être utilisé comme argument générique.
- Peut convertir implicitement `delegate*` en `void*` .
- Peut effectuer une conversion explicite de `void*` en `delegate*` .

Restrictions :

- Les attributs personnalisés ne peuvent pas être appliqués à un `delegate*` ou à l'un de ses éléments.
- Un `delegate*` paramètre ne peut pas être marqué comme `params`
- Un `delegate*` type présente toutes les restrictions d'un type pointeur normal.
- Les opérations arithmétiques sur les pointeurs ne peuvent pas être effectuées directement sur les types de pointeur fonction.

Syntaxe du pointeur de fonction

La syntaxe complète du pointeur fonction est représentée par la grammaire suivante :

antlr

```
pointer_type
    : ...
    | funcptr_type
    ;

funcptr_type
    : 'delegate' '*' calling_convention_specifier? '<'
funcptr_parameter_list funcptr_return_type '>'
    ;

calling_convention_specifier
    : 'managed'
    | 'unmanaged' ('[' unmanaged_calling_convention ']')?
    ;

unmanaged_calling_convention
    : 'Cdecl'
    | 'Stdcall'
    | 'Thiscall'
```

```
    | 'Fastcall'
    | identifier (',' identifier)*
    ;

funptr_parameter_list
    : (funcptr_parameter ',')*
    ;

funcptr_parameter
    : funcptr_parameter_modifier? type
    ;

funcptr_return_type
    : funcptr_return_modifier? return_type
    ;

funcptr_parameter_modifier
    : 'ref'
    | 'out'
    | 'in'
    ;

funcptr_return_modifier
    : 'ref'
    | 'ref readonly'
    ;
```

Si aucun `calling_convention_specifier` n'est fourni, la valeur par défaut est `managed` . L'encodage des métadonnées précis de `calling_convention_specifier` et de ce qui `identifier` sont valides dans le `unmanaged_calling_convention` est couvert dans la représentation des métadonnées des conventions d'appel.

C#

```
delegate int Func1(string s);
delegate Func1 Func2(Func1 f);

// Function pointer equivalent without calling convention
delegate*<string, int>;
delegate*<delegate*<string, int>, delegate*<string, int>>;

// Function pointer equivalent with calling convention
delegate* managed<string, int>;
delegate*<delegate* managed<string, int>, delegate*<string, int>>;
```

Conversions de pointeur de fonction

Dans un contexte non sécurisé, l'ensemble des conversions implicites disponibles (conversions implicites) est étendu pour inclure les conversions de pointeur implicites

suivantes :

- *Conversions existantes*
- De _ *type funcptr* `F0` à un *autre* _ *type funcptr* `F1` , à condition que les conditions suivantes soient remplies :
  - `F0` et `F1` ont le même nombre de paramètres, et chaque paramètre `D0n` dans `F0` a les mêmes `ref` `out` modificateurs,, ou `in` que le paramètre correspondant `D1n` dans `F1` .
  - Pour chaque paramètre de valeur (paramètre sans `ref` `out` modificateur, ou `in`), il existe une conversion d'identité, une conversion de référence implicite ou une conversion de pointeur implicite du type de paramètre dans `F0` le type de paramètre correspondant dans `F1` .
  - Pour chaque `ref` `out` paramètre, ou `in` , le type de paramètre dans `F0` est le même que le type de paramètre correspondant dans `F1` .
  - Si le type de retour est par valeur (no `ref` ou `ref readonly`), une identité, une référence implicite ou une conversion de pointeur implicite existe du type de retour de `F1` au type de retour de `F0` .
  - Si le type de retour est par référence (`ref` ou `ref readonly`), le type de retour et les `ref` modificateurs de `F1` sont les mêmes que le type de retour et les `ref` modificateurs de `F0` .
  - La Convention d'appel de `F0` est identique à la Convention d'appel de `F1` .

Autoriser l'adresse à cibler des méthodes

Les groupes de méthodes sont désormais autorisés comme arguments d'une expression d'adresse. Le type d'une telle expression sera un `delegate*` qui a la signature équivalente de la méthode cible et une convention d'appel managée :

C#

```
unsafe class Util {
    public static void Log() { }

    void Use() {
        delegate*<void> ptr1 = &Util.Log;

        // Error: type "delegate*<void>" not compatible with "delegate*
<int>";
        delegate*<int> ptr2 = &Util.Log;
   }
}
```

Dans un contexte non sécurisé, une méthode `M` est compatible avec un type de pointeur de fonction `F` si toutes les conditions suivantes sont vraies :

- `M` et `F` ont le même nombre de paramètres, et chaque paramètre dans `M` a les mêmes `ref` `out` modificateurs,, ou `in` que le paramètre correspondant dans `F` .
- Pour chaque paramètre de valeur (paramètre sans `ref` `out` modificateur, ou `in`), il existe une conversion d'identité, une conversion de référence implicite ou une conversion de pointeur implicite du type de paramètre dans `M` le type de paramètre correspondant dans `F` .
- Pour chaque `ref` `out` paramètre, ou `in` , le type de paramètre dans `M` est le même que le type de paramètre correspondant dans `F` .
- Si le type de retour est par valeur (no `ref` ou `ref readonly`), une identité, une référence implicite ou une conversion de pointeur implicite existe du type de retour de `F` au type de retour de `M` .
- Si le type de retour est par référence (`ref` ou `ref readonly`), le type de retour et les `ref` modificateurs de `F` sont les mêmes que le type de retour et les `ref` modificateurs de `M` .
- La Convention d'appel de `M` est identique à la Convention d'appel de `F` . Cela comprend le bit de convention d'appel, ainsi que tous les indicateurs de convention d'appel spécifiés dans l'identificateur non managé.
- `M` est une méthode statique.

Dans un contexte non sécurisé, une conversion implicite existe à partir d'une expression d'adresse dont la cible est un groupe `E` de méthodes vers un type de pointeur fonction compatible `F` si `E` contient au moins une méthode applicable dans sa forme normale à une liste d'arguments construite à l'aide des types de paramètres et des modificateurs de `F` , comme décrit dans l'exemple suivant.

- Une méthode unique `M` est sélectionnée correspondant à un appel de méthode de la forme `E(A)` avec les modifications suivantes :
  - La liste `A` d'arguments est une liste d'expressions, chacune classée comme une variable et avec le type et le modificateur (`ref` , `out` ou `in`) de *la _ _ liste de paramètres funcptr* correspondante de `F` .
  - Les méthodes candidates sont uniquement les méthodes applicables dans leur forme normale, et non celles applicables dans leur forme développée.
  - Les méthodes candidates sont uniquement les méthodes statiques.
- Si l'algorithme de la résolution de surcharge génère une erreur, une erreur de compilation se produit. Dans le cas contraire, l'algorithme produit une seule meilleure méthode qui `M` a le même nombre de paramètres que `F` et la conversion est considérée comme existante.

- La méthode sélectionnée `M` doit être compatible (comme défini ci-dessus) avec le type de pointeur de fonction `F` . Sinon, une erreur de compilation se produit.
- Le résultat de la conversion est un pointeur de fonction de type `F` .

Cela signifie que les développeurs peuvent dépendre de règles de résolution de surcharge pour fonctionner conjointement avec l'opérateur d'adresse :

C#

```
unsafe class Util {
    public static void Log() { }
    public static void Log(string p1) { }
    public static void Log(int i) { };

    void Use() {
        delegate*<void> a1 = &Log; // Log()
        delegate*<int, void> a2 = &Log; // Log(int i)

        // Error: ambiguous conversion from method group Log to "void*"
        void* v = &Log;
    }
```

L'opérateur d'adresse sera implémenté à l'aide de l' `ldftn` instruction.

Restrictions de cette fonctionnalité :

- S'applique uniquement aux méthodes marquées comme `static` .
- Les `static` fonctions non locales ne peuvent pas être utilisées dans `&` . Les détails de l'implémentation de ces méthodes ne sont délibérément pas spécifiés par le langage. Cela indique s'ils sont statiques ou d'instance ou exactement la signature avec laquelle ils sont émis.

Opérateurs sur les types de pointeur de fonction

La section du code unsafe sur les opérateurs est modifiée comme suit :

> Dans un contexte non sécurisé, plusieurs constructions sont disponibles pour fonctionner sur tous les _pointer _ type_s qui ne sont pas _funcptr _ type_s :
>
> - L' `*` opérateur peut être utilisé pour effectuer une indirection de pointeur (indirection de pointeur).
> - L' `->` opérateur peut être utilisé pour accéder à un membre d'un struct via un pointeur (accès aux membres du pointeur).
> - L' `[]` opérateur peut être utilisé pour indexer un pointeur (accès à un élément pointeur).

> - L' `&` opérateur peut être utilisé pour obtenir l'adresse d'une variable (opérateur d'adresse).
> - Les `++` `--` opérateurs et peuvent être utilisés pour incrémenter et décrémenter des pointeurs (incrément et décrémentation de pointeur).
> - Les `+` `-` opérateurs et peuvent être utilisés pour effectuer des opérations arithmétiques sur les pointeurs (opérations arithmétiquessur les pointeurs).
> - Les `==` `!=` opérateurs,, `<` ,, `>` `<=` et `=>` peuvent être utilisés pour comparer des pointeurs (Comparaison de pointeur).
> - L' `stackalloc` opérateur peut être utilisé pour allouer de la mémoire à partir de la pile des appels (mémoires tampons de taille fixe).
> - L' `fixed` instruction peut être utilisée pour corriger temporairement une variable afin que son adresse puisse être obtenue (l'instruction fixed).
>
> Dans un contexte non sécurisé, plusieurs constructions sont disponibles pour fonctionner sur tous les _funcptr _ type_s :
>
> - L' `&` opérateur peut être utilisé pour obtenir l'adresse des méthodes statiques (autoriser l'adresse-de aux méthodes cibles).
> - Les `==` `!=` opérateurs,, `<` ,, `>` `<=` et `=>` peuvent être utilisés pour comparer des pointeurs (Comparaison de pointeur).

En outre, nous modifions toutes les sections dans `Pointers in expressions` pour interdire les types de pointeur de fonction, à l'exception `Pointer comparison` de et `The sizeof operator` .

Meilleure fonction membre

La meilleure spécification de membre de fonction sera modifiée pour inclure la ligne suivante :

> Un `delegate*` est plus spécifique que `void*`

Cela signifie qu'il est possible de surcharger sur `void*` et, `delegate*` et d'utiliser judicieusement l'opérateur d'adresse.

Inférence de type

Dans du code non sécurisé, les modifications suivantes sont apportées aux algorithmes d'inférence de type :

Types d'entrée

https://github.com/dotnet/csharplang/blob/master/spec/expressions.md#input-types

Les éléments suivants sont ajoutés :

> Si `E` est un groupe de méthodes d'adresse et `T` est un type pointeur de fonction, tous les types de paramètres de `T` sont des types d'entrée de `E` avec le type `T` .

Types de sorties

https://github.com/dotnet/csharplang/blob/master/spec/expressions.md#output-types

Les éléments suivants sont ajoutés :

> Si `E` est un groupe de méthodes d'adresse et `T` est un type de pointeur de fonction, le type de retour de `T` est un type de sortie de `E` type `T` .

Inférences de type de sortie

https://github.com/dotnet/csharplang/blob/master/spec/expressions.md#output-type-inferences

La puce suivante est ajoutée entre les puces 2 et 3 :

> - Si `E` est un groupe de méthodes d'adresse et qu' `T` il s'agit d'un type de pointeur de fonction avec des types de paramètres `T1...Tk` et un type `Tb` de retour, et que la résolution de surcharge de `E` avec les types `T1..Tk` donne une seule méthode avec le type de retour `U` , une *inférence de la limite inférieure* est alors faite de `U` à `Tb` .

Inférences exactes

https://github.com/dotnet/csharplang/blob/master/spec/expressions.md#exact-inferences

La sous-puce suivante est ajoutée dans le cas de la puce 2 :

> - `V` est un type pointeur fonction `delegate*<V2..Vk, V1>` et `U` est un type pointeur fonction `delegate*<U2..Uk, U1>` , et la Convention d'appel de `V` est identique à `U` , et le refness de `Vi` est identique à `Ui` .

Inférences à limite inférieure

https://github.com/dotnet/csharplang/blob/master/spec/expressions.md#lower-bound-inferences

Le cas suivant est ajouté à la puce 3 :

> - `V` est un type pointeur fonction `delegate*<V2..Vk, V1>` et il existe un type pointeur fonction `delegate*<U2..Uk, U1>` qui `U` est identique à `delegate*<U2..Uk, U1>` , et la Convention d'appel de `V` est identique à `U` , et le refness de `Vi` est identique à `Ui` .

La première puce de l'inférence de `Ui` à `Vi` est modifiée en :

> - Si `U` n'est pas un type de pointeur fonction et qu'il n'est pas connu comme étant un type `Ui` référence, ou si `U` est un type pointeur fonction et qu' `Ui` il n'est pas connu comme étant un type pointeur fonction ou un type référence, une *inférence exacte* est effectuée

Ensuite, ajouté après la troisième puce de l'inférence de `Ui` à `Vi` :

> - Sinon, si `V` est `delegate*<V2..Vk, V1>` Then, l'inférence dépend du paramètre i-Th de `delegate*<V2..Vk, V1>` :
>   - Si v1 :
>   - Si le retour est par valeur, une *inférence de liaison inférieure* est effectuée.
>   - Si le retour est par référence, une *inférence exacte* est effectuée.
>   - Si v2.. VK
>   - Si le paramètre est par valeur, une *inférence de* la limite supérieure est effectuée.
>   - Si le paramètre est par référence, une *inférence exacte* est effectuée.

Inférences à la limite supérieure

https://github.com/dotnet/csharplang/blob/master/spec/expressions.md#upper-bound-inferences

Le cas suivant est ajouté à la puce 2 :

> - `U` est un type pointeur fonction `delegate*<U2..Uk, U1>` et `V` est un type pointeur fonction qui est identique à `delegate*<V2..Vk, V1>` , et la Convention d'appel de `U` est identique à `V` , et le refness de `Ui` est identique à `Vi` .

La première puce de l'inférence de `Ui` à `Vi` est modifiée en :

> - Si `U` n'est pas un type de pointeur fonction et qu'il n'est pas connu comme étant un type `Ui` référence, ou si `U` est un type pointeur fonction et qu' `Ui` il n'est pas connu comme étant un type pointeur fonction ou un type référence, une *inférence exacte* est effectuée

Ensuite ajouté après la troisième puce de l'inférence de `Ui` à `Vi` :

> - Sinon, si `U` est `delegate*<U2..Uk, U1>` Then, l'inférence dépend du paramètre i-Th de `delegate*<U2..Uk, U1>` :
>   - Si U1 :
>   - Si le retour est par valeur, une *inférence de* la limite supérieure est effectuée.
>   - Si le retour est par référence, une *inférence exacte* est effectuée.
>   - Si U2.. Londres
>   - Si le paramètre est par valeur, une *inférence de liaison inférieure* est effectuée.
>   - Si le paramètre est par référence, une *inférence exacte* est effectuée.

Représentation de métadonnées des `in` `out` paramètres,, et `ref readonly` et des types de retour

Les signatures de pointeur de fonction n'ont pas d'emplacement d'indicateurs de paramètres. nous devons donc Encoder si les paramètres et le type de retour sont `in` , `out` ou à `ref readonly` l'aide de modreqs.

`in`

Nous utilisons `System.Runtime.InteropServices.InAttribute` la réutilisation, appliquée en tant que `modreq` au spécificateur Ref sur un paramètre ou un type de retour, pour signifier ce qui suit :

- S'il est appliqué à un spécificateur de référence de paramètre, ce paramètre est traité comme `in` .
- S'il est appliqué au spécificateur Ref de type de retour, le type de retour est traité comme `ref readonly` .

`out`

Nous utilisons `System.Runtime.InteropServices.OutAttribute` , appliqué en tant que `modreq` au spécificateur Ref sur un type de paramètre, pour signifier que le paramètre est un `out` paramètre.

Erreurs

- Il s'agit d'une erreur à appliquer `OutAttribute` en tant que modreq à un type de retour.
- L'application de `InAttribute` et `OutAttribute` en tant que modreq à un type de paramètre est une erreur.
- Si les deux sont spécifiés via modopt, ils sont ignorés.

Représentation des métadonnées des conventions d'appel

Les conventions d'appel sont encodées dans une signature de méthode dans les métadonnées par une combinaison de l' `CallKind` indicateur dans la signature et zéro ou plusieurs `modopt` s au début de la signature. ECMA-335 déclare actuellement les éléments suivants dans l' `CallKind` indicateur :

antlr

```
CallKind
    : default
    | unmanaged cdecl
    | unmanaged fastcall
    | unmanaged thiscall
    | unmanaged stdcall
    | varargs
    ;
```

Parmi ces fonctionnalités, les pointeurs de fonction en C# prendront en charge tous sauf `varargs` .

En outre, le runtime (et finalement 335) sera mis à jour pour inclure un nouveau `CallKind` sur les nouvelles plateformes. Cela n'a pas de nom formel actuellement, mais ce document sera utilisé `unmanaged ext` comme espace réservé pour créer le nouveau format de convention d'appel extensible. Avec no `modopt` s, `unmanaged ext` est la Convention d'appel de la plateforme par défaut, `unmanaged` sans les crochets.

Mappage `calling_convention_specifier` de à un `CallKind`

`calling_convention_specifier`Qui est omis, ou spécifié comme `managed` , correspond à `default` `CallKind` . Il s'agit par défaut `CallKind` de toute méthode non attribuée avec `UnmanagedCallersOnly` .

C# reconnaît 4 identificateurs spéciaux qui mappent à des s non managés spécifiques existants `CallKind` à partir d'ECMA 335. Pour que ce mappage se produise, ces identificateurs doivent être spécifiés eux-mêmes, sans aucun autre identificateur, et cette spécification est encodée dans la spécification pour `unmanaged_calling_convention` s. Ces identificateurs sont `Cdecl` , `Thiscall` , `Stdcall` et `Fastcall` , qui correspondent respectivement à `unmanaged cdecl` , `unmanaged thiscall` , `unmanaged stdcall` et `unmanaged fastcall` . Si plusieurs `identifer` identificateurs sont spécifiés, ou si le seul ne `identifier` fait pas partie des identificateurs spécialement reconnus, nous effectuons une recherche de nom spéciale sur l'identificateur avec les règles suivantes :

- Nous allons ajouter la `identifier` chaîne `CallConv`
- Nous examinons uniquement les types définis dans l' `System.Runtime.CompilerServices` espace de noms.
- Nous examinons uniquement les types définis dans la bibliothèque principale de l'application, qui est la bibliothèque qui définit `System.Object` et n'a pas de dépendances.
- Nous examinons uniquement les types publics.

Si la recherche aboutit sur tous les `identifier` s spécifiés dans un `unmanaged_calling_convention` , nous encodez le `CallKind` en tant que `unmanaged ext` et encodez chacun des types résolus dans le jeu de `modopt` s au début de la signature du pointeur de fonction. En guise de note, ces règles signifient que les utilisateurs ne peuvent pas préfixer ces `identifier` s avec `CallConv` , car cela entraînera la recherche `CallConvCallConvVectorCall` .

Lors de l'interprétation des métadonnées, nous examinons d'abord le `CallKind` . Si ce n'est pas le cas `unmanaged ext` , nous ignorons tous les `modopt` sur le type de retour dans le cadre de la détermination de la Convention d'appel et n'utilisez que le `CallKind` . Si `CallKind` est `unmanaged ext` , nous examinons le modopts au début du type de pointeur de fonction, en prenant l'Union de tous les types qui remplissent les conditions suivantes :

- Est défini dans la bibliothèque principale, qui est la bibliothèque qui ne référence aucune autre bibliothèque et définit `System.Object` .
- Le type est défini dans l' `System.Runtime.CompilerServices` espace de noms.

- Le type commence par le préfixe `CallConv` .
- Le type est public.

Celles-ci représentent les types qui doivent être trouvés lors de l'exécution de la recherche sur les `identifier` dans une lors de la `unmanaged_calling_convention` définition d'un type de pointeur fonction dans la source.

La tentative d'utilisation d'un pointeur de fonction avec un `CallKind` de `unmanaged ext` si le runtime cible ne prend pas en charge la fonctionnalité est une erreur. Cela sera déterminé en recherchant la présence de la `System.Runtime.CompilerServices.RuntimeFeature.UnmanagedCallKind` constante. Si cette constante est présente, le runtime est pris en compte pour prendre en charge la fonctionnalité.

`System.Runtime.InteropServices.UnmanagedCallersOnlyAttribute`

`System.Runtime.InteropServices.UnmanagedCallersOnlyAttribute` est un attribut utilisé par le CLR pour indiquer qu'une méthode doit être appelée avec une convention d'appel spécifique. Pour cette raison, nous présentons la prise en charge suivante de l'utilisation de l'attribut :

- L'appel direct d'une méthode annotée avec cet attribut à partir de C# est une erreur. Les utilisateurs doivent obtenir un pointeur de fonction vers la méthode, puis appeler ce pointeur.
- L'application de l'attribut à tout autre chose qu'une méthode statique ordinaire ou une fonction locale statique ordinaire est une erreur. Le compilateur C# marque toutes les méthodes non statiques ou non ordinaires importées à partir des métadonnées avec cet attribut comme non prises en charge par le langage.
- C'est une erreur pour une méthode marquée avec l'attribut d'avoir un paramètre ou un type de retour qui n'est pas un `unmanaged_type` .
- C'est une erreur pour une méthode marquée avec l'attribut d'avoir des paramètres de type, même si ces paramètres de type sont limités à `unmanaged` .
- Une méthode dans un type générique doit être marquée avec l'attribut.
- La conversion d'une méthode marquée avec l'attribut en type délégué est une erreur.
- La spécification de tous les types pour `UnmanagedCallersOnly.CallConvs` qui ne satisfont pas aux conditions requises pour l'appel de Convention dans les métadonnées est une erreur `modopt` .

Lors de la détermination de la Convention d'appel d'une méthode marquée avec un `UnmanagedCallersOnly` attribut valide, le compilateur effectue les vérifications suivantes sur les types spécifiés dans la `CallConvs` propriété afin de déterminer les effectifs `CallKind` et les `modopt` à utiliser pour déterminer la Convention d'appel :

- Si aucun type n'est spécifié, `CallKind` est traité comme `unmanaged ext` , sans convention d'appel, `modopt` au début du type de pointeur de fonction.
- Si un type est spécifié et que ce type est nommé `CallConvCdecl` , `CallConvThiscall` , `CallConvStdcall` ou `CallConvFastcall` , le `CallKind` est traité comme `unmanaged cdecl` , `unmanaged thiscall` , `unmanaged stdcall` ou `unmanaged fastcall` , respectivement, sans convention d'appel `modopt` au début du type de pointeur de fonction.
- Si plusieurs types sont spécifiés ou si le type unique n'est pas nommé l'un des types spécialement appelés ci-dessus, le `CallKind` est traité comme `unmanaged ext` , avec l'Union des types spécifiés traités comme `modopt` s au début du type de pointeur de fonction.

Le compilateur examine ensuite cet effet `CallKind` et cette `modopt` collection et utilise des règles de métadonnées normales pour déterminer la Convention d'appel finale du type de pointeur de fonction.

Questions ouvertes

Détection de la prise en charge du runtime pour `unmanaged ext`

https://github.com/dotnet/runtime/issues/38135 effectue le suivi de l'ajout de cet indicateur. Selon les commentaires de la revue, nous utilisons la propriété spécifiée dans le problème ou utilisons la présence de `UnmanagedCallersOnlyAttribute` comme indicateur qui détermine si les runtimes prennent en charge `unmanaged ext` .

Considérations

Autoriser les méthodes d'instance

La proposition peut être étendue pour prendre en charge les méthodes d'instance en tirant parti de la `EXPLICITTHIS` Convention d'appel CLI (nommée `instance` dans le code

C#). Cette forme de pointeurs de fonction CLI place le `this` paramètre en tant que premier paramètre explicite de la syntaxe du pointeur de fonction.

C#

```
unsafe class Instance {
    void Use() {
        delegate* instance<Instance, string> f = &ToString;
        f(this);
    }
}
```

C'est un son, mais il ajoute une certaine compliquation à la proposition. En particulier parce que les pointeurs de fonction qui diffèrent par la Convention d'appel `instance` et `managed` seraient incompatibles même si les deux cas sont utilisés pour appeler des méthodes managées avec la même signature C#. Dans tous les cas, dans tous les cas, il est utile de disposer d'une solution de contournement simple : utilisez une `static` fonction locale.

C#

```
unsafe class Instance {
    void Use() {
        static string toString(Instance i) => i.ToString();
        delegate*<Instance, string> f = &toString;
        f(this);
    }
}
```

Ne nécessite pas de non-sécurité au niveau de la déclaration

Au lieu de demander `unsafe` à chaque utilisation d'un `delegate*` , n'en avez besoin qu'au point où un groupe de méthodes est converti en `delegate*` . C'est là qu'interviennent les principaux problèmes de sécurité (sachant que l'assembly conteneur ne peut pas être déchargé tant que la valeur est active). `unsafe`Le besoin d'un autre emplacement peut être considéré comme excessif.

C'est ainsi que la conception était prévue à l'origine. Mais les règles de langage qui en résultent sont très maladroites. Il est impossible de masquer le fait qu'il s'agit d'une valeur de pointeur et qu'elle reste en lecture seule sans le `unsafe` mot clé. Par exemple, la conversion vers `object` ne peut pas être autorisée, il ne peut pas être membre d'un

`class` , etc... La conception C# est nécessaire `unsafe` pour toutes les utilisations de pointeur. par conséquent, cette conception suit cela.

Les développeurs peuvent toujours présenter un wrapper *sécurisé* sur `delegate*` les valeurs de la même façon que pour les types pointeur normaux. Vous devez :

```
C#

unsafe struct Action {
    delegate*<void> _ptr;

    Action(delegate*<void> ptr) => _ptr = ptr;
    public void Invoke() => _ptr();
}
```

Utilisation de délégués

Au lieu d'utiliser un nouvel élément de syntaxe, `delegate*` , utilisez simplement `delegate` les types existants avec `*` le type suivant :

```
C#

Func<object, object, bool>* ptr = &object.ReferenceEquals;
```

La gestion de la Convention d'appel peut être effectuée en annotant les `delegate` types avec un attribut qui spécifie une `CallingConvention` valeur. L'absence d'un attribut signifie la Convention d'appel managée.

L'encodage de ce en IL pose problème. La valeur sous-jacente doit être représentée sous la forme d'un pointeur, mais elle doit également :

1. Avoir un type unique pour autoriser les surcharges avec différents types de pointeur de fonction.
2. Être équivalent à des fins de OHI dans les limites de l'assembly.

Le dernier point est particulièrement problématique. Cela signifie que chaque assembly qui utilise `Func<int>*` doit encoder un type équivalent dans les métadonnées même si `Func<int>*` est défini dans un assembly, même si ne contrôle pas. En outre, tout autre type défini avec le nom `System.Func<T>` dans un assembly qui n'est pas mscorlib doit être différent de la version définie dans mscorlib.

Une option qui a été explorée a émis un tel pointeur comme `mod_req(Func<int>) void*` . Cela ne fonctionne pas tant qu'un `mod_req` ne peut pas être lié à un `TypeSpec` et ne peut

donc pas cibler des instanciations génériques.

Pointeurs de fonction nommés

La syntaxe du pointeur de fonction peut être lourde, en particulier dans les cas complexes comme les pointeurs de fonction imbriqués. Au lieu de demander aux développeurs de taper la signature chaque fois que le langage peut autoriser des déclarations nommées de pointeurs de fonction comme c'est le cas avec `delegate` .

C#

```
func* void Action();

unsafe class NamedExample {
    void M(Action a) {
        a();
    }
}
```

Une partie du problème est ici que la primitive CLI sous-jacente n'a pas de nom, ce qui est purement une invention C# et nécessite un peu de métadonnées pour permettre l'activation. C'est réalisable, mais il s'agit d'un travail important. En fait, C# a besoin d'un compagnon pour la table de définition de type pour ces noms uniquement.

De même, lorsque les arguments des pointeurs de fonction nommés ont été examinés, nous avons trouvé qu'ils s'appliquent aussi bien à un certain nombre d'autres scénarios. Par exemple, il serait tout aussi commode de déclarer des tuples nommés pour réduire le besoin de taper la signature complète dans tous les cas.

C#

```
(int x, int y) Point;

class NamedTupleExample {
    void M(Point p) {
        Console.WriteLine(p.x);
    }
}
```

Après la discussion, nous avons décidé de ne pas autoriser la déclaration nommée de `delegate*` types. Si nous constatons qu'il y a des besoins importants en ce qui concerne les commentaires sur l'utilisation des clients, nous étudierons une solution d'attribution de noms qui fonctionne pour les pointeurs de fonction, les tuples, les génériques, etc.

Cela peut être similaire à d'autres suggestions, telles que `typedef` la prise en charge complète dans le langage.

Éléments futurs à prendre en considération

délégués statiques

Cela fait référence à la proposition pour permettre la déclaration de `delegate` types qui peuvent uniquement faire référence à des `static` membres. L'avantage est qu' `delegate` il est possible d'allouer gratuitement et mieux les scénarios de performances.

Si la fonctionnalité du pointeur de fonction est implémentée `static delegate` , la proposition sera probablement fermée. L'avantage proposé par cette fonctionnalité est la nature libre de l'allocation. Toutefois, des investigations récentes ont détecté que le déchargement d'assembly n'est pas possible en raison du déchargement de l'assembly. Il doit y avoir un handle fort de l'objet `static delegate` à la méthode à laquelle il fait référence pour empêcher le déchargement de l'assembly à partir de celui-ci.

Pour conserver chaque `static delegate` instance, il est nécessaire d'allouer un nouveau descripteur qui s'exécute sur les objectifs de la proposition. Il existait des conceptions où l'allocation pouvait être amortie en une seule allocation par site d'appel, mais cela était un peu complexe et ne semble pas avoir un intérêt pour le compromis.

Cela signifie que les développeurs doivent essentiellement décider entre les compromis suivants :

1. Sécurité face au déchargement d'un assembly : cela nécessite des allocations et `delegate` est donc déjà une option suffisante.
2. Aucune sécurité en face du déchargement d'assembly : utilisez un `delegate*` . Cela peut être encapsulé dans un `struct` pour permettre l'utilisation en dehors d'un `unsafe` contexte dans le reste du code.

Supprimer l'émission de l' `localsinit` indicateur.

Article • 13/05/2021

- [x] proposé
- [] Prototype : non démarré
- [] Implémentation : non démarrée
- [] Spécification : non démarrée

Résumé

Autoriser la suppression de l'émission de l' `localsinit` indicateur via l' `SkipLocalsInitAttribute` attribut.

Motivation

Arrière-plan

Par spécification CLR, les variables locales qui ne contiennent pas de références ne sont pas initialisées à une valeur particulière par la machine virtuelle/JIT. La lecture de ces variables sans initialisation est de type sécurisé, mais dans le cas contraire, le comportement est non défini et spécifique à l'implémentation. En général, les variables locales non initialisées contiennent les valeurs laissées dans la mémoire qui est maintenant occupée par le frame de pile. Cela peut entraîner un comportement non déterministe et des bogues difficiles à reproduire.

Il existe deux façons de « assigner » une variable locale :

- en stockant une valeur ou
- en spécifiant `localsinit` un indicateur qui force tout ce qui est alloué à former le pool de mémoire local à être initialisé à zéro Remarque : cela inclut à la fois les variables locales et les `stackalloc` données.

L'utilisation de données non initialisées est déconseillée et n'est pas autorisée dans du code vérifiable. Bien qu'il soit possible de prouver que, par le biais de l'analyse des fluides, l'algorithme de vérification peut être conservateur et nécessite simplement que `localsinit` soit défini.

L'historique du compilateur C# émet un `localsinit` indicateur sur toutes les méthodes qui déclarent des variables locales.

Tandis que C# utilise une analyse d'assignation définie qui est plus stricte que celle requise par le CLR (C# doit également prendre en compte l'étendue des variables locales), il n'est pas strictement garanti que le code résultant serait officiellement vérifiable :

- Les règles CLR et C# ne conviennent pas si le passage d'un `out` argument local As est un `use` .
- Les règles CLR et C# peuvent ne pas s'accorder sur le traitement des branches conditionnelles lorsque les conditions sont connues (propagation constante).
- Le CLR peut également nécessiter tout simplement `localinits` , puisque cela est autorisé.

Problème

Dans une application hautes performances, le coût de l'initialisation forcée de zéro peut être perceptible. Cela est particulièrement perceptible lorsque `stackalloc` est utilisé.

Dans certains cas, le JIT peut Elide initialiser l'initialisation zéro des variables locales individuelles lorsque cette initialisation est « supprimée » par les assignations suivantes. Toutes les JIT ne le font pas et une telle optimisation a des limites. Elle n'est pas utile avec `stackalloc` .

Pour illustrer le problème, il existe un bogue connu dans lequel une méthode ne contenant pas de `IL` variables locales n'aurait pas d' `localsinit` indicateur. Le bogue est déjà exploité par les utilisateurs en ajoutant des `stackalloc` méthodes de ce type, de manière intentionnelle, afin d'éviter les coûts d'initialisation. Cela est en dépit du fait que l'absence de `IL` variables locales est une mesure instable et peut varier en fonction des modifications apportées à la stratégie CodeGen. Le bogue doit être corrigé et les utilisateurs doivent disposer d'une méthode plus documentée et fiable pour supprimer l'indicateur.

Conception détaillée

Autorisez `System.Runtime.CompilerServices.SkipLocalsInitAttribute` la spécification comme un moyen d'indiquer au compilateur de ne pas émettre d' `localsinit` indicateur.

Le résultat final est que les variables locales ne peuvent pas être initialisées à zéro par le JIT, qui est dans la plupart des cas non observables en C#.
En plus de ces `stackalloc` données ne seront pas initialisées à zéro. Cela est tout à fait observable, mais également le scénario le plus motivant.

Les cibles d'attribut autorisées et reconnues sont : `Method` , `Property` ,, `Module` `Class` , `Struct` , `Interface` , `Constructor` . Toutefois, le compilateur ne requiert pas que l'attribut soit défini avec les cibles listées, et il se chargera de déterminer l'assembly dans lequel l'attribut est défini.

Lorsque l'attribut est spécifié sur un conteneur (`class` , `module` , qui contient la méthode pour une méthode imbriquée,...), l'indicateur affecte toutes les méthodes contenues dans le conteneur.

Les méthodes synthétisées « héritent » de l'indicateur du conteneur/propriétaire logique.

L'indicateur affecte uniquement la stratégie CodeGen pour les corps de méthode réels. c'est-à-dire, l'indicateur n'a aucun effet sur les méthodes abstraites et n'est pas propagé aux méthodes de substitution/implémentation.

Il s'agit d'une ***fonctionnalité de compilateur*** qui n'est ***pas une fonctionnalité de langage***.
De la même façon que les commutateurs de ligne de commande du compilateur, la fonctionnalité contrôle les détails d'implémentation d'une stratégie CodeGen particulière et n'a pas besoin d'être requise par la spécification C#.

Inconvénients

- Les anciens/autres compilateurs ne peuvent pas honorer l'attribut. Ignorer l'attribut est un comportement compatible. Seul peut entraîner une légère baisse des performances.

- Le code sans `localinits` indicateur peut déclencher des échecs de vérification. Les utilisateurs qui demandent cette fonctionnalité sont généralement insoucieux de la vérifiabilité.

- L'application de l'attribut à des niveaux supérieurs par rapport à une méthode individuelle a un effet non local, qui est observable quand `stackalloc` est utilisé. Pourtant, il s'agit du scénario le plus demandé.

Autres solutions

- omettre `localinits` l'indicateur lorsque la méthode est déclarée en `unsafe` contexte. Cela peut entraîner un changement de comportement silencieux et dangereux de déterministe en non déterministe dans le cas de `stackalloc` .
- omettez `localinits` Flag Always. Pire encore que ci-dessus.
- omettez `localinits` Flag sauf si `stackalloc` est utilisé dans le corps de la méthode. Ne traite pas le scénario le plus demandé et peut désactiver le code non vérifiable sans option pour revenir en arrière.

Questions non résolues

- L'attribut doit-il être émis aux métadonnées ?

Réunions de conception

Aucun.

Annotations des paramètres de type sans contraintes

Article • 13/05/2021

Résumé

Autorisez les annotations Nullable pour les paramètres de type qui ne sont pas limités à des types valeur ou des types référence : `T?` .

C#

```
static T? FirstOrDefault<T>(this IEnumerable<T> collection) { ... }
```

Annotation `?`

En C# 8, les `?` Annotations ne pouvaient être appliquées qu'aux paramètres de type qui étaient explicitement limités à des types valeur ou des types référence. En C# 9, `?` les annotations peuvent être appliquées à n'importe quel paramètre de type, indépendamment des contraintes.

À moins qu'un paramètre de type ne soit explicitement restreint à des types valeur, les annotations ne peuvent être appliquées que dans un `#nullable enable` contexte.

Si un paramètre de type `T` est substitué par un type référence, `T?` représente une instance Nullable de ce type référence.

C#

```
var s1 = new string[0].FirstOrDefault();  // string? s1
var s2 = new string?[0].FirstOrDefault(); // string? s2
```

Si `T` est substitué par un type valeur, `T?` représente une instance de `T` .

C#

```
var i1 = new int[0].FirstOrDefault();  // int i1
var i2 = new int?[0].FirstOrDefault(); // int? i2
```

Si `T` est substitué par un type annoté `U?` , `T?` représente le type annoté `U?` plutôt que `U??` .

C#

```
var u1 = new U[0].FirstOrDefault();  // U? u1
var u2 = new U?[0].FirstOrDefault(); // U? u2
```

Si `T` est substitué par un type `U` , puis `T?` représenté `U?` , même dans un `#nullable disable` contexte.

C#

```
#nullable disable
var u3 = new U[0].FirstOrDefault();  // U? u3
```

Pour les valeurs de retour, équivaut `T?` à `[MaybeNull]T` ; pour les valeurs d'argument, `T?` est équivalent à `[AllowNull]T` . L'équivalence est importante lors de la substitution ou de l'implémentation d'interfaces à partir d'un assembly compilé avec C# 8.

C#

```
public abstract class A
{
    [return: MaybeNull] public abstract T F1<T>();
    public abstract void F2<T>([AllowNull] T t);
}

public class B : A
{
    public override T? F1<T>() where T : default { ... }       // matches
A.F1<T>()
    public override void F2<T>(T? t) where T : default { ... } // matches
A.F2<T>()
}
```

`default` restrictions

Pour la compatibilité avec le code existant où les méthodes génériques substituées et explicitement implémentées ne pouvaient pas inclure de clauses de contrainte explicites, `T?` dans une méthode substituée ou implémentée explicitement est traitée comme `Nullable<T>` où `T` est un type valeur.

Pour autoriser les annotations pour les paramètres de type limités aux types référence, C# 8 permettait des contraintes explicites `where T : class` et `where T : struct` sur la méthode substituée ou implémentée explicitement.

C#

```
class A1
{
    public virtual void F1<T>(T? t) where T : struct { }
    public virtual void F1<T>(T? t) where T : class { }
}

class B1 : A1
{
    public override void F1<T>(T? t) /*where T : struct*/ { }
    public override void F1<T>(T? t) where T : class { }
}
```

Pour autoriser les annotations pour les paramètres de type qui ne sont pas limités aux types référence ou aux types valeur, C# 9 autorise une nouvelle `where T : default` contrainte.

C#

```
class A2
{
    public virtual void F2<T>(T? t) where T : struct { }
    public virtual void F2<T>(T? t) { }
}

class B2 : A2
{
    public override void F2<T>(T? t) /*where T : struct*/ { }
    public override void F2<T>(T? t) where T : default { }
}
```

L'utilisation d'une `default` contrainte autre que sur un remplacement de méthode ou une implémentation explicite constitue une erreur. L'utilisation d'une contrainte est une erreur `default` quand le paramètre de type correspondant dans la méthode d'interface ou de substitution est imposé à un type référence ou un type valeur.

Réunions de conception

- https://github.com/dotnet/csharplang/blob/master/meetings/2019/LDM-2019-11-25.md

- https://github.com/dotnet/csharplang/blob/master/meetings/2020/LDM-2020-06-17.md#t

Record structs

Article • 06/23/2023

> **Note**
>
> This article is a feature specification. The specification represents the *proposed* feature specification. There may be some discrepancies between the feature specification and the completed implementation. Those differences are captured in the pertinent **language design meeting (LDM) notes**. Links to pertinent meetings are included at the bottom of the spec. You can learn more about the process for merging feature speclets into the C# language standard in the article on the **specifications**.

The syntax for a record struct is as follows:

antlr

```
record_struct_declaration
    : attributes? struct_modifier* 'partial'? 'record' 'struct' identifier
type_parameter_list?
      parameter_list? struct_interfaces? type_parameter_constraints_clause*
record_struct_body
    ;

record_struct_body
    : struct_body
    | ';'
    ;
```

Record struct types are value types, like other struct types. They implicitly inherit from the class `System.ValueType`. The modifiers and members of a record struct are subject to the same restrictions as those of structs (accessibility on type, modifiers on members, `base(...)` instance constructor initializers, definite assignment for `this` in constructor, destructors, ...). Record structs will also follow the same rules as structs for parameterless instance constructors and field initializers, but this document assumes that we will lift those restrictions for structs generally.

See §15.4.9 See parameterless struct constructors spec.

Record structs cannot use `ref` modifier.

At most one partial type declaration of a partial record struct may provide a `parameter_list`. The `parameter_list` may be empty.

Record struct parameters cannot use `ref`, `out` or `this` modifiers (but `in` and `params` are allowed).

Members of a record struct

In addition to the members declared in the record struct body, a record struct type has additional synthesized members. Members are synthesized unless a member with a "matching" signature is declared in the record struct body or an accessible concrete non-virtual member with a "matching" signature is inherited. Two members are considered matching if they have the same signature or would be considered "hiding" in an inheritance scenario. See Signatures and overloading §7.6. It is an error for a member of a record struct to be named "Clone".

It is an error for an instance field of a record struct to have an unsafe type.

A record struct is not permitted to declare a destructor.

The synthesized members are as follows:

Equality members

The synthesized equality members are similar as in a record class (`Equals` for this type, `Equals` for `object` type, `==` and `!=` operators for this type),
except for the lack of `EqualityContract`, null checks or inheritance.

The record struct implements `System.IEquatable<R>` and includes a synthesized strongly-typed overload of `Equals(R other)` where `R` is the record struct. The method is `public`. The method can be declared explicitly. It is an error if the explicit declaration does not match the expected signature or accessibility.

If `Equals(R other)` is user-defined (not synthesized) but `GetHashCode` is not, a warning is produced.

```
C#
```

```
public readonly bool Equals(R other);
```

The synthesized `Equals(R)` returns `true` if and only if for each instance field `fieldN` in the record struct the value of
`System.Collections.Generic.EqualityComparer<TN>.Default.Equals(fieldN, other.fieldN)` where `TN` is the field type is `true`.

The record struct includes synthesized `==` and `!=` operators equivalent to operators declared as follows:

```
C#
```

```
public static bool operator==(R r1, R r2)
    => r1.Equals(r2);
public static bool operator!=(R r1, R r2)
    => !(r1 == r2);
```

The `Equals` method called by the `==` operator is the `Equals(R other)` method specified above. The `!=` operator delegates to the `==` operator. It is an error if the operators are declared explicitly.

The record struct includes a synthesized override equivalent to a method declared as follows:

```
C#
```

```
public override readonly bool Equals(object? obj);
```

It is an error if the override is declared explicitly. The synthesized override returns `other is R temp && Equals(temp)` where `R` is the record struct.

The record struct includes a synthesized override equivalent to a method declared as follows:

```
C#
```

```
public override readonly int GetHashCode();
```

The method can be declared explicitly.

A warning is reported if one of `Equals(R)` and `GetHashCode()` is explicitly declared but the other method is not explicit.

The synthesized override of `GetHashCode()` returns an `int` result of combining the values of `System.Collections.Generic.EqualityComparer<TN>.Default.GetHashCode(fieldN)` for each instance field `fieldN` with `TN` being the type of `fieldN`.

For example, consider the following record struct:

```
C#
```

```
record struct R1(T1 P1, T2 P2);
```

For this record struct, the synthesized equality members would be something like:

C#

```
struct R1 : IEquatable<R1>
{
    public T1 P1 { get; set; }
    public T2 P2 { get; set; }
    public override bool Equals(object? obj) => obj is R1 temp &&
Equals(temp);
    public bool Equals(R1 other)
    {
        return
            EqualityComparer<T1>.Default.Equals(P1, other.P1) &&
            EqualityComparer<T2>.Default.Equals(P2, other.P2);
    }
    public static bool operator==(R1 r1, R1 r2)
        => r1.Equals(r2);
    public static bool operator!=(R1 r1, R1 r2)
        => !(r1 == r2);
    public override int GetHashCode()
    {
        return Combine(
            EqualityComparer<T1>.Default.GetHashCode(P1),
            EqualityComparer<T2>.Default.GetHashCode(P2));
    }
}
```

Printing members: PrintMembers and ToString methods

The record struct includes a synthesized method equivalent to a method declared as follows:

C#

```
private bool PrintMembers(System.Text.StringBuilder builder);
```

The method does the following:

1. for each of the record struct's printable members (non-static public field and readable property members), appends that member's name followed by " = " followed by the member's value separated with ", ",
2. return true if the record struct has printable members.

For a member that has a value type, we will convert its value to a string representation using the most efficient method available to the target platform. At present that means calling `ToString` before passing to `StringBuilder.Append`.

If the record's printable members do not include a readable property with a non-`readonly` `get` accessor, then the synthesized `PrintMembers` is `readonly`. There is no requirement for the record's fields to be `readonly` for the `PrintMembers` method to be `readonly`.

The `PrintMembers` method can be declared explicitly. It is an error if the explicit declaration does not match the expected signature or accessibility.

The record struct includes a synthesized method equivalent to a method declared as follows:

```
C#

public override string ToString();
```

If the record struct's `PrintMembers` method is `readonly`, then the synthesized `ToString()` method is `readonly`.

The method can be declared explicitly. It is an error if the explicit declaration does not match the expected signature or accessibility.

The synthesized method:

1. creates a `StringBuilder` instance,
2. appends the record struct name to the builder, followed by " { ",
3. invokes the record struct's `PrintMembers` method giving it the builder, followed by " " if it returned true,
4. appends "}",
5. returns the builder's contents with `builder.ToString()`.

For example, consider the following record struct:

```
C#

record struct R1(T1 P1, T2 P2);
```

For this record struct, the synthesized printing members would be something like:

```
C#
```

```
struct R1 : IEquatable<R1>
{
    public T1 P1 { get; set; }
    public T2 P2 { get; set; }

    private bool PrintMembers(StringBuilder builder)
    {
        builder.Append(nameof(P1));
        builder.Append(" = ");
        builder.Append(this.P1); // or builder.Append(this.P1.ToString());
if P1 has a value type
        builder.Append(", ");

        builder.Append(nameof(P2));
        builder.Append(" = ");
        builder.Append(this.P2); // or builder.Append(this.P2.ToString());
if P2 has a value type

        return true;
    }

    public override string ToString()
    {
        var builder = new StringBuilder();
        builder.Append(nameof(R1));
        builder.Append(" { ");

        if (PrintMembers(builder))
            builder.Append(" ");

        builder.Append("}");
        return builder.ToString();
    }
}
```

Positional record struct members

In addition to the above members, record structs with a parameter list ("positional records") synthesize additional members with the same conditions as the members above.

Primary Constructor

A record struct has a public constructor whose signature corresponds to the value parameters of the type declaration. This is called the primary constructor for the type. It is an error to have a primary constructor and a constructor with the same signature

already present in the struct. If the type declaration does not include a parameter list, no primary constructor is generated.

```C#
record struct R1
{
    public R1() { } // ok
}

record struct R2()
{
    public R2() { } // error: 'R2' already defines constructor with same
parameter types
}
```

Instance field declarations for a record struct are permitted to include variable initializers. If there is no primary constructor, the instance initializers execute as part of the parameterless constructor. Otherwise, at runtime the primary constructor executes the instance initializers appearing in the record-struct-body.

If a record struct has a primary constructor, any user-defined constructor must have an explicit `this` constructor initializer that calls the primary constructor or an explicitly declared constructor.

Parameters of the primary constructor as well as members of the record struct are in scope within initializers of instance fields or properties. Instance members would be an error in these locations (similar to how instance members are in scope in regular constructor initializers today, but an error to use), but the parameters of the primary constructor would be in scope and useable and would shadow members. Static members would also be useable.

A warning is produced if a parameter of the primary constructor is not read.

The definite assignment rules for struct instance constructors apply to the primary constructor of record structs. For instance, the following is an error:

```C#
record struct Pos(int X) // definite assignment error in primary constructor
{
    private int x;
    public int X { get { return x; } set { x = value; } } = X;
}
```

Properties

For each record struct parameter of a record struct declaration there is a corresponding public property member whose name and type are taken from the value parameter declaration.

For a record struct:

- A public `get` and `init` auto-property is created if the record struct has `readonly` modifier, `get` and `set` otherwise. Both kinds of set accessors (`set` and `init`) are considered "matching". So the user may declare an init-only property in place of a synthesized mutable one. An inherited `abstract` property with matching type is overridden. No auto-property is created if the record struct has an instance field with expected name and type. It is an error if the inherited property does not have `public` `get` and `set`/`init` accessors. It is an error if the inherited property or field is hidden.
  The auto-property is initialized to the value of the corresponding primary constructor parameter. Attributes can be applied to the synthesized auto-property and its backing field by using `property:` or `field:` targets for attributes syntactically applied to the corresponding record struct parameter.

Deconstruct

A positional record struct with at least one parameter synthesizes a public void-returning instance method called `Deconstruct` with an out parameter declaration for each parameter of the primary constructor declaration. Each parameter of the Deconstruct method has the same type as the corresponding parameter of the primary constructor declaration. The body of the method assigns each parameter of the Deconstruct method to the value from an instance member access to a member of the same name. If the instance members accessed in the body do not include a property with a non-`readonly` `get` accessor, then the synthesized `Deconstruct` method is `readonly`. The method can be declared explicitly. It is an error if the explicit declaration does not match the expected signature or accessibility, or is static.

Allow `with` expression on structs

It is now valid for the receiver in a `with` expression to have a struct type.

On the right hand side of the `with` expression is a `member_initializer_list` with a sequence of assignments to *identifier*, which must be an accessible instance field or

property of the receiver's type.

For a receiver with struct type, the receiver is first copied, then each `member_initializer` is processed the same way as an assignment to a field or property access of the result of the conversion. Assignments are processed in lexical order.

Improvements on records

Allow `record class`

The existing syntax for record types allows `record class` with the same meaning as `record`:

antlr

```
record_declaration
    : attributes? class_modifier* 'partial'? 'record' 'class'? identifier
type_parameter_list?
      parameter_list? record_base? type_parameter_constraints_clause*
record_body
    ;
```

Allow user-defined positional members to be fields

See https://github.com/dotnet/csharplang/blob/master/meetings/2020/LDM-2020-10-05.md#changing-the-member-type-of-a-primary-constructor-parameter

No auto-property is created if the record has or inherits an instance field with expected name and type.

Allow parameterless constructors and member initializers in structs

See parameterless struct constructors spec.

Open questions

- how to recognize record structs in metadata? (we don't have an unspeakable clone method to leverage...)

Answered

- confirm that we want to keep PrintMembers design (separate method returning `bool`) (answer: yes)
- confirm we won't allow `record ref struct` (issue with `IEquatable<RefStruct>` and ref fields) (answer: yes)
- confirm implementation of equality members. Alternative is that synthesized `bool Equals(R other)`, `bool Equals(object? other)` and operators all just delegate to `ValueType.Equals`. (answer: yes)
- confirm that we want to allow field initializers when there is a primary constructor. Do we also want to allow parameterless struct constructors while we're at it (the Activator issue was apparently fixed)? (answer: yes, updated spec should be reviewed in LDM)
- how much do we want to say about `Combine` method? (answer: as little as possible)
- should we disallow a user-defined constructor with a copy constructor signature? (answer: no, there is no notion of copy constructor in the record structs spec)
- confirm that we want to disallow members named "Clone". (answer: correct)
- double-check that synthesized `Equals` logic is functionally equivalent to runtime implementation (e.g. float.NaN) (answer: confirmed in LDM)
- could field- or property-targeting attributes be placed in the positional parameter list? (answer: yes, same as for record class)
- `with` on generics? (answer: out of scope for C# 10)
- should `GetHashCode` include a hash of the type itself, to get different values between `record struct S1;` and `record struct S2;`? (answer: no)

Parameterless struct constructors

Article • 06/23/2023

> **Note**
>
> This article is a feature specification. The specification represents the *proposed* feature specification. There may be some discrepancies between the feature specification and the completed implementation. Those differences are captured in the pertinent **language design meeting (LDM) notes**. Links to pertinent meetings are included at the bottom of the spec. You can learn more about the process for merging feature speclets into the C# language standard in the article on the **specifications**.

Summary

Support parameterless constructors and instance field initializers for struct types.

Motivation

Explicit parameterless constructors would give more control over minimally constructed instances of the struct type. Instance field initializers would allow simplified initialization across multiple constructors. Together these would close an obvious gap between `struct` and `class` declarations.

Support for field initializers would also allow initialization of fields in `record struct` declarations without explicitly implementing the primary constructor.

C#

```
record struct Person(string Name)
{
    public object Id { get; init; } = GetNextId();
}
```

If struct field initializers are supported for constructors with parameters, it seems natural to extend that to parameterless constructors as well.

C#

```
record struct Person()
{
    public string Name { get; init; }
    public object Id { get; init; } = GetNextId();
}
```

Proposal

Instance field initializers

Instance field declarations for a struct may include initializers.

As with class field initializers §14.5.6.3:

> A variable initializer for an instance field cannot reference the instance being created.

An error is reported if a struct has field initializers and no declared instance constructors since the field initializers will not be run.

C#

```
struct S { int F = 42; } // error: 'struct' with field initializers must include an explicitly declared constructor
```

Constructors

A struct may declare a parameterless instance constructor.

A parameterless instance constructor is valid for all struct kinds including `struct`, `readonly struct`, `ref struct`, and `record struct`.

If no parameterless instance constructor is declared, the struct (see §15.4.9) ...

> implicitly has a parameterless instance constructor which always returns the value that results from setting all value type fields to their default value and all reference type fields to null.

Modifiers

A parameterless instance struct constructor must be declared `public`.

C#

```
struct S0 { }                   // ok
struct S1 { public S1() { } }   // ok
struct S2 { internal S2() { } } // error: parameterless constructor must be
'public'
```

Non-public constructors are ignored when importing types from metadata.

Constructors can be declared `extern` or `unsafe`. Constructors cannot be `partial`.

Executing field initializers

Instance variable initializers (§14.11.3) is **modified** as follows:

> When **a class** instance constructor has no constructor initializer, or it has a constructor initializer of the form `base(...)`, that constructor implicitly performs the initializations specified by the *variable_initializer*s of the instance fields declared in its class. This corresponds to a sequence of assignments that are executed immediately upon entry to the constructor and before the implicit invocation of the direct base class constructor.
>
> **When a struct instance constructor has no constructor initializer, that constructor implicitly performs the initializations specified by the *variable_initializer*s of the instance fields declared in its struct. This corresponds to a sequence of assignments that are executed immediately upon entry to the constructor.**
>
> **When a struct instance constructor has a `this()` constructor initializer that represents the *default parameterless constructor*, the declared constructor implicitly clears all instance fields and performs the initializations specified by the *variable_initializer*s of the instance fields declared in its struct. Immediately upon entry to the constructor, all value type fields are set to their default value and all reference type fields are set to `null`. Immediately after that, a sequence of assignments corresponding to the *variable_initializer*s are executed.**

Definite assignment

Instance fields (other than `fixed` fields) must be definitely assigned in struct instance constructors that do not have a `this()` initializer (see §15.4.9).

C#

```
struct S0 // ok
{
    int x;
    object y;
}

struct S1 // error: 'struct' with field initializers must include an
explicitly declared constructor
{
    int x = 1;
    object y;
}

struct S2
{
    int x = 1;
    object y;
    public S2() { } // error: field 'y' must be assigned
}

struct S3 // ok
{
    int x = 1;
    object y;
    public S3() { y = 2; }
}
```

No `base()` initializer

A `base()` initializer is disallowed in struct constructors.

The compiler will not emit a call to the base `System.ValueType` constructor from struct instance constructors.

`record struct`

An error is reported if a `record struct` has field initializers and does not contain a primary constructor nor any instance constructors since the field initializers will not be run.

C#

```
record struct R0;                   // ok
record struct R1 { int F = 42; }    // error: 'struct' with field
initializers must include an explicitly declared constructor
record struct R2() { int F = 42; } // ok
record struct R3(int F);            // ok
```

A `record struct` with an empty parameter list will have a parameterless primary constructor.

```
C#

record struct R3();                // primary .ctor: public R3() { }
record struct R4() { int F = 42; } // primary .ctor: public R4() { F = 42; }
```

An explicit parameterless constructor in a `record struct` must have a `this` initializer that calls the primary constructor or an explicitly declared constructor.

```
C#

record struct R5(int F)
{
    public R5() { }                  // error: must have 'this' initializer
that calls explicit .ctor
    public R5(object o) : this() { } // ok
    public int F =  F;
}
```

Fields

The implicitly-defined parameterless constructor will zero fields rather than calling any parameterless constructors for the field types. No warnings are reported that field constructors are ignored. *No change from C#9.*

```
C#

struct S0
{
    public S0() { }
}

struct S1
{
    S0 F; // S0 constructor ignored
}

struct S<T> where T : struct
{
    T F; // constructor (if any) ignored
}
```

`default` expression

`default` ignores the parameterless constructor and generates a zeroed instance. *No change from C#9.*

```
C#

// struct S { public S() { } }

_ = default(S); // constructor ignored, no warning
```

`new()`

Object creation invokes the parameterless constructor if public; otherwise the instance is zeroed. *No change from C#9.*

```
C#

// public struct PublicConstructor { public PublicConstructor() { } }
// public struct PrivateConstructor { private PrivateConstructor() { } }

_ = new PublicConstructor();  // call PublicConstructor::.ctor()
_ = new PrivateConstructor(); // initobj PrivateConstructor
```

A warning wave may report a warning for use of `new()` with a struct type that has constructors but no parameterless constructor. No warning will be reported when using substituting such a struct type for a type parameter with a `new()` or `struct` constraint.

```
C#

struct S { public S(int i) { } }
static T CreateNew<T>() where T : new() => new T();

_ = new S();        // warning: no constructor called
_ = CreateNew<S>(); // ok
```

Uninitialized values

A local or field of a struct type that is not explicitly initialized is zeroed. The compiler reports a definite assignment error for an uninitialized struct that is not empty. *No change from C#9.*

```
C#

NoConstructor s1;
PublicConstructor s2;
```

```
s1.ToString(); // error: use of unassigned local (unless type is empty)
s2.ToString(); // error: use of unassigned local (unless type is empty)
```

Array allocation

Array allocation ignores any parameterless constructor and generates zeroed elements. *No change from C#9.*

C#

```
// struct S { public S() { } }

var a = new S[1]; // constructor ignored, no warning
```

Parameter default value `new()`

A parameter default value of `new()` binds to the parameterless constructor if public (and reports an error that the value is not constant); otherwise the instance is zeroed. *No change from C#9.*

C#

```
// public struct PublicConstructor { public PublicConstructor() { } }
// public struct PrivateConstructor { private PrivateConstructor() { } }

static void F1(PublicConstructor s1 = new()) { }  // error: default value
must be constant
static void F2(PrivateConstructor s2 = new()) { } // ok: initobj
```

Type parameter constraints: `new()` and `struct`

The `new()` and `struct` type parameter constraints require the parameterless constructor to be `public` if defined (see Satisfying constraints - §8.4.5).

The compiler assumes all structs satisfy `new()` and `struct` constraints. *No change from C#9.*

C#

```
// public struct PublicConstructor { public PublicConstructor() { } }
// public struct InternalConstructor { internal InternalConstructor() { } }

static T CreateNew<T>() where T : new() => new T();
static T CreateStruct<T>() where T : struct => new T();
```

```
_ = CreateNew<PublicConstructor>();      // ok
_ = CreateStruct<PublicConstructor>();   // ok

_ = CreateNew<InternalConstructor>();    // compiles; may fail at runtime
_ = CreateStruct<InternalConstructor>(); // compiles; may fail at runtime
```

`new T()` is emitted as a call to `System.Activator.CreateInstance<T>()`, and the compiler assumes the implementation of `CreateInstance<T>()` invokes the `public` parameterless constructor if defined.

With .NET Framework, `Activator.CreateInstance<T>()` invokes the parameterless constructor if the constraint is `where T : new()` but appears to ignore the parameterless constructor if the constraint is `where T : struct`.

Optional parameters

Constructors with optional parameters are not considered parameterless constructors. *No change from C#9.*

C#

```
struct S1 { public S1(string s = "") { } }
struct S2 { public S2(params object[] args) { } }

_ = new S1(); // ok: ignores constructor
_ = new S2(); // ok: ignores constructor
```

Metadata

Explicit parameterless struct instance constructors will be emitted to metadata.

Public parameterless struct instance constructors will be imported from metadata; non-public struct instance constructors will be ignored. *No change from C#9.*

See also

- https://github.com/dotnet/roslyn/issues/1029

Design meetings

- https://github.com/dotnet/csharplang/blob/main/meetings/2021/LDM-2021-04-28.md#open-questions-in-record-and-parameterless-structs
- https://github.com/dotnet/csharplang/blob/main/meetings/2021/LDM-2021-03-10.md#parameterless-struct-constructors
- https://github.com/dotnet/csharplang/blob/main/meetings/2021/LDM-2021-01-27.md#field-initializers

Global Using Directive

Article • 03/24/2022

> **Note**
>
> This article is a feature specification. The specification represents the *proposed* feature specification. There may be some discrepancies between the feature specification and the completed implementation. Those differences are captured in the pertinent **language design meeting (LDM) notes**. Links to pertinent meetings are included at the bottom of the spec. You can learn more about the process for merging feature speclets into the C# language standard in the article on the **specifications**.

Syntax for a using directive is extended with an optional `global` keyword that can precede the `using` keyword:

antlr

```
compilation_unit
    : extern_alias_directive* global_using_directive* using_directive*
global_attributes? namespace_member_declaration*
    ;

global_using_directive
    : global_using_alias_directive
    | global_using_namespace_directive
    | global_using_static_directive
    ;

global_using_alias_directive
    : 'global' 'using' identifier '=' namespace_or_type_name ';'
    ;

global_using_namespace_directive
    : 'global' 'using' namespace_name ';'
    ;

global_using_static_directive
    : 'global' 'using' 'static' type_name ';'
    ;
```

- The *global_using_directive*s are allowed only on the Compilation Unit level (cannot be used inside a *namespace_declaration*).
- The *global_using_directive*s, if any, must precede any *using_directive*s.

- The scope of a *global_using_directive*s extends over the *namespace_member_declaration*s of all compilation units within the program. The scope of a *global_using_directive* specifically does not include other *global_using_directive*s. Thus, peer *global_using_directive*s or those from a different compilation unit do not affect each other, and the order in which they are written is insignificant. The scope of a *global_using_directive* specifically does not include *using_directive*s immediately contained in any compilation unit of the program.

The effect of adding a *global_using_directive* to a program can be thought of as the effect of adding a similar *using_directive* that resolves to the same target namespace or type to every compilation unit of the program. However, the target of a *global_using_directive* is resolved in context of the compilation unit that contains it.

§7.7 Scopes

These are the relevant bullet points with proposed additions (which are **in bold**):

- The scope of name defined by an *extern_alias_directive* extends over the ***global_using_directives*, ***using_directive*s, *global_attributes* and *namespace_member_declaration*s of its immediately containing compilation unit or namespace body. An *extern_alias_directive* does not contribute any new members to the underlying declaration space. In other words, an *extern_alias_directive* is not transitive, but, rather, affects only the compilation unit or namespace body in which it occurs.
- **The scope of a name defined or imported by a *global_using_directive* extends over the *global_attributes* and *namespace_member_declaration*s of all the *compilation_unit*s in the program.**

§7.8 Namespace and type names

Changes are made to the algorithm determining the meaning of a *namespace_or_type_name* as follows.

This is the relevant bullet point with proposed additions (which are **in bold**):

- If the *namespace_or_type_name* is of the form `I` or of the form `I<A1, ..., Ak>`:
  - If `K` is zero and the *namespace_or_type_name* appears within a generic method declaration (§14.6) and if that declaration includes a type parameter (§14.2.3) with name `I`, then the *namespace_or_type_name* refers to that type parameter.

- Otherwise, if the *namespace_or_type_name* appears within a type declaration, then for each instance type `T` (§14.3.2), starting with the instance type of that type declaration and continuing with the instance type of each enclosing class or struct declaration (if any):
  - If `K` is zero and the declaration of `T` includes a type parameter with name `I`, then the *namespace_or_type_name* refers to that type parameter.
  - Otherwise, if the *namespace_or_type_name* appears within the body of the type declaration, and `T` or any of its base types contain a nested accessible type having name `I` and `K` type parameters, then the *namespace_or_type_name* refers to that type constructed with the given type arguments. If there is more than one such type, the type declared within the more derived type is selected. Note that non-type members (constants, fields, methods, properties, indexers, operators, instance constructors, destructors, and static constructors) and type members with a different number of type parameters are ignored when determining the meaning of the *namespace_or_type_name*.
- If the previous steps were unsuccessful then, for each namespace `N`, starting with the namespace in which the *namespace_or_type_name* occurs, continuing with each enclosing namespace (if any), and ending with the global namespace, the following steps are evaluated until an entity is located:
  - If `K` is zero and `I` is the name of a namespace in `N`, then:
    - If the location where the *namespace_or_type_name* occurs is enclosed by a namespace declaration for `N` and the namespace declaration contains an *extern_alias_directive* or *using_alias_directive* that associates the name `I` with a namespace or type, **or any namespace declaration for `N` in the program contains a *global_using_alias_directive* that associates the name `I` with a namespace or type,** then the *namespace_or_type_name* is ambiguous and a compile-time error occurs.
    - Otherwise, the *namespace_or_type_name* refers to the namespace named `I` in `N`.
  - Otherwise, if `N` contains an accessible type having name `I` and `K` type parameters, then:
    - If `K` is zero and the location where the *namespace_or_type_name* occurs is enclosed by a namespace declaration for `N` and the namespace declaration contains an *extern_alias_directive* or *using_alias_directive* that associates the name `I` with a namespace or type, **or any namespace declaration for `N` in the program contains a *global_using_alias_directive* that associates the name `I` with a namespace or type,** then the *namespace_or_type_name* is ambiguous and a compile-time error occurs.

        - Otherwise, the *namespace_or_type_name* refers to the type constructed with the given type arguments.
    - Otherwise, if the location where the *namespace_or_type_name* occurs is enclosed by a namespace declaration for `N`:
        - If `K` is zero and the namespace declaration contains an *extern_alias_directive* or *using_alias_directive* that associates the name `I` with an imported namespace or type, **or any namespace declaration for `N` in the program contains a *global_using_alias_directive* that associates the name `I` with an imported namespace or type,** then the *namespace_or_type_name* refers to that namespace or type.
        - Otherwise, if the namespaces and type declarations imported by the *using_namespace_directive*s and *using_alias_directive*s of the namespace declaration **and the namespaces and type declarations imported by the *global_using_namespace_directive*s and *global_using_static_directive*s of any namespace declaration for `N` in the program** contain exactly one accessible type having name `I` and `K` type parameters, then the *namespace_or_type_name* refers to that type constructed with the given type arguments.
        - Otherwise, if the namespaces and type declarations imported by the *using_namespace_directive*s and *using_alias_directive*s of the namespace declaration **and the namespaces and type declarations imported by the *global_using_namespace_directive*s and *global_using_static_directive*s of any namespace declaration for `N` in the program** contain more than one accessible type having name `I` and `K` type parameters, then the *namespace_or_type_name* is ambiguous and an error occurs.
    - Otherwise, the *namespace_or_type_name* is undefined and a compile-time error occurs.

Simple names §11.7.4

Changes are made to the *simple_name* evaluation rules as follows.

This is the relevant bullet point with proposed additions (which are **in bold**):

- Otherwise, for each namespace `N`, starting with the namespace in which the *simple_name* occurs, continuing with each enclosing namespace (if any), and ending with the global namespace, the following steps are evaluated until an entity is located:
    - If `K` is zero and `I` is the name of a namespace in `N`, then:

    - If the location where the *simple_name* occurs is enclosed by a namespace declaration for `N` and the namespace declaration contains an *extern_alias_directive* or *using_alias_directive* that associates the name `I` with a namespace or type, **or any namespace declaration for `N` in the program contains a *global_using_alias_directive* that associates the name `I` with a namespace or type,** then the *simple_name* is ambiguous and a compile-time error occurs.
    - Otherwise, the *simple_name* refers to the namespace named `I` in `N`.
- Otherwise, if `N` contains an accessible type having name `I` and `K` type parameters, then:
    - If `K` is zero and the location where the *simple_name* occurs is enclosed by a namespace declaration for `N` and the namespace declaration contains an *extern_alias_directive* or *using_alias_directive* that associates the name `I` with a namespace or type, **or any namespace declaration for `N` in the program contains a *global_using_alias_directive* that associates the name `I` with a namespace or type,** then the *simple_name* is ambiguous and a compile-time error occurs.
    - Otherwise, the *namespace_or_type_name* refers to the type constructed with the given type arguments.
- Otherwise, if the location where the *simple_name* occurs is enclosed by a namespace declaration for `N`:
    - If `K` is zero and the namespace declaration contains an *extern_alias_directive* or *using_alias_directive* that associates the name `I` with an imported namespace or type, **or any namespace declaration for `N` in the program contains a *global_using_alias_directive* that associates the name `I` with an imported namespace or type,** then the *simple_name* refers to that namespace or type.
    - Otherwise, if the namespaces and type declarations imported by the *using_namespace_directive*s and *using_static_directive*s of the namespace declaration **and the namespaces and type declarations imported by the *global_using_namespace_directive*s and *global_using_static_directive*s of any namespace declaration for `N` in the program** contain exactly one accessible type or non-extension static member having name `I` and `K` type parameters, then the *simple_name* refers to that type or member constructed with the given type arguments.
    - Otherwise, if the namespaces and types imported by the *using_namespace_directive*s of the namespace declaration **and the namespaces and type declarations imported by the *global_using_namespace_directive*s and *global_using_static_directive*s of**

any namespace declaration for `N` in the program contain more than one accessible type or non-extension-method static member having name `I` and `K` type parameters, then the *simple_name* is ambiguous and an error occurs.

Extension method invocations §11.7.8.3

Changes are made to the algorithm to find the best *type_name* `C` as follows. This is the relevant bullet point with proposed additions (which are **in bold**):

- Starting with the closest enclosing namespace declaration, continuing with each enclosing namespace declaration, and ending with the containing compilation unit, successive attempts are made to find a candidate set of extension methods:
  - If the given namespace or compilation unit directly contains non-generic type declarations `Ci` with eligible extension methods `Mj`, then the set of those extension methods is the candidate set.
  - If types `Ci` imported by *using_static_declarations* and directly declared in namespaces imported by *using_namespace_directive*s in the given namespace or compilation unit **and, if containing compilation unit is reached, imported by *global_using_static_declarations* and directly declared in namespaces imported by *global_using_namespace_directive*s in the program** directly contain eligible extension methods `Mj`, then the set of those extension methods is the candidate set.

Compilation units §13.2

A *compilation_unit* defines the overall structure of a source file. A compilation unit consists of **zero or more *global_using_directive*s followed by** zero or more *using_directive*s followed by zero or more *global_attributes* followed by zero or more *namespace_member_declaration*s.

antlr

```
compilation_unit
    : extern_alias_directive* global_using_directive* using_directive*
global_attributes? namespace_member_declaration*
    ;
```

A C# program consists of one or more compilation units, each contained in a separate source file. When a C# program is compiled, all of the compilation units are processed together. Thus, compilation units can depend on each other, possibly in a circular fashion.

The *global_using_directives* of a compilation unit affect the *global_attributes* and *namespace_member_declarations* of all compilation units in the program.

Extern aliases §13.4

The scope of an *extern_alias_directive* extends over the ***global_using_directives*, *using_directives*, *global_attributes*** and *namespace_member_declarations* of its immediately containing compilation unit or namespace body.

Using alias directives §13.5.2

The order in which *using_alias_directives* are written has no significance, and resolution of the *namespace_or_type_name* referenced by a *using_alias_directive* is not affected by the *using_alias_directive* itself or by other *using_directives* in the immediately containing compilation unit or namespace body, **and, if the *using_alias_directive* is immediately contained in a compilation unit, is not affected by the *global_using_directives* in the program**. In other words, the *namespace_or_type_name* of a *using_alias_directive* is resolved as if the immediately containing compilation unit or namespace body had no *using_directives* **and, if the *using_alias_directive* is immediately contained in a compilation unit, the program had no *global_using_directives***. A *using_alias_directive* may however be affected by *extern_alias_directives* in the immediately containing compilation unit or namespace body.

Global Using alias directives

A *global_using_alias_directive* introduces an identifier that serves as an alias for a namespace or type within the program.

antlr

```
global_using_alias_directive
    : 'global' 'using' identifier '=' namespace_or_type_name ';'
    ;
```

Within member declarations in any compilation unit of a program that contains a *global_using_alias_directive*, the identifier introduced by the *global_using_alias_directive* can be used to reference the given namespace or type.

The *identifier* of a *global_using_alias_directive* must be unique within the declaration space of any compilation unit of a program that contains the *global_using_alias_directive*.

Just like regular members, names introduced by *global_using_alias_directive*s are hidden by similarly named members in nested scopes.

The order in which *global_using_alias_directive*s are written has no significance, and resolution of the *namespace_or_type_name* referenced by a *global_using_alias_directive* is not affected by the *global_using_alias_directive* itself or by other *global_using_directive*s or *using_directive*s in the program. In other words, the *namespace_or_type_name* of a *global_using_alias_directive* is resolved as if the immediately containing compilation unit had no *using_directive*s and the entire containing program had no *global_using_directive*s. A *global_using_alias_directive* may however be affected by *extern_alias_directive*s in the immediately containing compilation unit.

A *global_using_alias_directive* can create an alias for any namespace or type.

Accessing a namespace or type through an alias yields exactly the same result as accessing that namespace or type through its declared name.

Using aliases can name a closed constructed type, but cannot name an unbound generic type declaration without supplying type arguments.

Global Using namespace directives

A *global_using_namespace_directive* imports the types contained in a namespace into the program, enabling the identifier of each type to be used without qualification.

antlr

```
global_using_namespace_directive
    : 'global' 'using' namespace_name ';'
    ;
```

Within member declarations in a program that contains a *global_using_namespace_directive*, the types contained in the given namespace can be referenced directly.

A *global_using_namespace_directive* imports the types contained in the given namespace, but specifically does not import nested namespaces.

Unlike a *global_using_alias_directive*, a *global_using_namespace_directive* may import types whose identifiers are already defined within a compilation unit of the program. In effect, in a given compilation unit, names imported by any

global_using_namespace_directive in the program are hidden by similarly named members in the compilation unit.

When more than one namespace or type imported by *global_using_namespace_directive*s or *global_using_static_directive*s in the same program contain types by the same name, references to that name as a *type_name* are considered ambiguous.

Furthermore, when more than one namespace or type imported by *global_using_namespace_directive*s or *global_using_static_directive*s in the same program contain types or members by the same name, references to that name as a *simple_name* are considered ambiguous.

The *namespace_name* referenced by a *global_using_namespace_directive* is resolved in the same way as the *namespace_or_type_name* referenced by a *global_using_alias_directive*. Thus, *global_using_namespace_directive*s in the same program do not affect each other and can be written in any order.

Global Using static directives

A *global_using_static_directive* imports the nested types and static members contained directly in a type declaration into the containing program, enabling the identifier of each member and type to be used without qualification.

antlr

```
global_using_static_directive
    : 'global' 'using' 'static' type_name ';'
    ;
```

Within member declarations in a program that contains a *global_using_static_directive*, the accessible nested types and static members (except extension methods) contained directly in the declaration of the given type can be referenced directly.

A *global_using_static_directive* specifically does not import extension methods directly as static methods, but makes them available for extension method invocation.

A *global_using_static_directive* only imports members and types declared directly in the given type, not members and types declared in base classes.

Ambiguities between multiple *global_using_namespace_directive*s and *global_using_static_directives* are discussed in the section for *global_using_namespace_directive*s (above).

Qualified alias member §13.8

Changes are made to the algorithm determining the meaning of a *qualified_alias_member* as follows.

This is the relevant bullet point with proposed additions (which are **in bold**):

- Otherwise, starting with the namespace declaration (§13.3) immediately containing the *qualified_alias_member* (if any), continuing with each enclosing namespace declaration (if any), and ending with the compilation unit containing the *qualified_alias_member*, the following steps are evaluated until an entity is located:
  - If the namespace declaration or compilation unit contains a *using_alias_directive* that associates `N` with a type, **or, when a compilation unit is reached, the program contains a *global_using_alias_directive* that associates `N` with a type,** then the *qualified_alias_member* is undefined and a compile-time error occurs.
  - Otherwise, if the namespace declaration or compilation unit contains an *extern_alias_directive* or *using_alias_directive* that associates `N` with a namespace, ***or, when a compilation unit is reached, the program contains a *global_using_alias_directive* that associates `N` with a namespace,** then:
    - If the namespace associated with `N` contains a namespace named `I` and `K` is zero, then the *qualified_alias_member* refers to that namespace.
    - Otherwise, if the namespace associated with `N` contains a non-generic type named `I` and `K` is zero, then the *qualified_alias_member* refers to that type.
    - Otherwise, if the namespace associated with `N` contains a type named `I` that has `K` type parameters, then the *qualified_alias_member* refers to that type constructed with the given type arguments.
    - Otherwise, the *qualified_alias_member* is undefined and a compile-time error occurs.

File Scoped Namespaces

Article • 06/23/2023

> **Note**
>
> This article is a feature specification. The specification represents the *proposed* feature specification. There may be some discrepancies between the feature specification and the completed implementation. Those differences are captured in the pertinent **language design meeting (LDM) notes**. Links to pertinent meetings are included at the bottom of the spec. You can learn more about the process for merging feature speclets into the C# language standard in the article on the **specifications**.

Summary

File scoped namespaces use a less verbose format for the typical case of files containing only one namespace. The file scoped namespace format is `namespace X.Y.Z;` (note the semicolon and lack of braces). This allows for files like the following:

```
c#

namespace X.Y.Z;

using System;

class X
{
}
```

The semantics are that using the `namespace X.Y.Z;` form is equivalent to writing `namespace X.Y.Z { ... }` where the remainder of the file following the file-scoped namespace is in the `...` section of a standard namespace declaration.

Motivation

Analysis of the C# ecosystem shows that approximately 99.7% files are all of either one of these forms:

```
c#
```

```
namespace X.Y.Z
{
    // usings

    // types
}
```

or

```
c#
```

```
// usings

namespace X.Y.Z
{
    // types
}
```

However, both these forms force the user to indent the majority of their code and add a fair amount of ceremony for what is effectively a very basic concept. This affects clarity, uses horizontal and vertical space, and is often unsatisfying for users both used to C# and coming from other languages (which commonly have less ceremony here).

The primary goal of the feature therefore is to meet the needs of the majority of the ecosystem with less unnecessary boilerplate.

Detailed design

This proposal takes the form of a diff to the existing compilation units (§13.2) section of the specification.

Diff

~~A *compilation_unit* defines the overall structure of a source file. A compilation unit consists of zero or more *using_directives* followed by zero or more *global_attributes* followed by zero or more *namespace_member_declarations*.~~

A *compilation_unit* defines the overall structure of a source file. A compilation unit consists of zero or more *using_directive*s followed by zero or more *global_attributes* followed by a *compilation_unit_body*. A *compilation_unit_body* can either be a *file_scoped_namespace_declaration* or zero or more *statement*s and *namespace_member_declaration*s.

```
```

antlr

```
compilation_unit
~~    : extern_alias_directive* using_directive* global_attributes?
namespace_member_declaration*~~
    : extern_alias_directive* using_directive* global_attributes?
compilation_unit_body
    ;

compilation_unit_body
    : statement* namespace_member_declaration*
    | file_scoped_namespace_declaration
    ;
```

... unchanged ...

A *file_scoped_namespace_declaration* will contribute members corresponding to the *namespace_declaration* it is semantically equivalent to. See (Namespace Declarations) for more details.

Namespace declarations

A *namespace_declaration* consists of the keyword `namespace`, followed by a namespace name and body, optionally followed by a semicolon. A *file_scoped_namespace_declaration* consists of the keyword `namespace`, followed by a namespace name, a semicolon and an optional list of *extern_alias_directive*s, *using_directive*s and *type_declaration*s.

antlr

```
namespace_declaration
    : 'namespace' qualified_identifier namespace_body ';'?
    ;

file_scoped_namespace_declaration
    : 'namespace' qualified_identifier ';' extern_alias_directive*
using_directive* type_declaration*
    ;

... unchanged ...
```

... unchanged ...

the two namespace declarations above contribute to the same declaration space, in this case declaring two classes with the fully qualified names `N1.N2.A` and `N1.N2.B`. Because

the two declarations contribute to the same declaration space, it would have been an error if each contained a declaration of a member with the same name.

A *file_scoped_namespace_declaration* permits a namespace declaration to be written without the `{ ... }` block. For example:

```
C#
```

```
extern alias A;
namespace Name;
using B;
class C
{
}
```

is semantically equivalent to

```
C#
```

```
extern alias A;
namespace Name
{
    using B;
    class C
    {
    }
}
```

Specifically, a *file_scoped_namespace_declaration* is treated the same as a *namespace_declaration* at the same location in the *compilation_unit* with the same *qualified_identifier*. The *extern_alias_directive*s, *using_directive*s and *type_declaration*s of that *file_scoped_namespace_declaration* act as if they were declared in the same order inside the *namespace_body* of that *namespace_declaration*.

A source file cannot contain both a *file_scoped_namespace_declaration* and a *namespace_declaration*. A source file cannot contain multiple *file_scoped_namespace_declaration*s. A *compilation_unit* cannot contain both a *file_scoped_namespace_declaration* and any top level *statement*s. *type_declaration*s cannot precede a *file_scoped_namespace_declaration*.

Extern aliases

... unchanged ...

Extended property patterns

Article • 06/23/2023

> **Note**
>
> This article is a feature specification. The specification represents the *proposed* feature specification. There may be some discrepancies between the feature specification and the completed implementation. Those differences are captured in the pertinent **language design meeting (LDM) notes**. Links to pertinent meetings are included at the bottom of the spec. You can learn more about the process for merging feature speclets into the C# language standard in the article on the **specifications**.

Summary

Allow property subpatterns to reference nested members, for instance:

```C#
if (e is MethodCallExpression { Method.Name: "MethodName" })
```

Instead of:

```C#
if (e is MethodCallExpression { Method: { Name: "MethodName" } })
```

Motivation

When you want to match a child property, nesting another recursive pattern adds too much noise which will hurt readability with no real advantage.

Detailed design

The *property_pattern* syntax is modified as follow:

```diff
```

```
property_pattern
  : type? property_pattern_clause simple_designation?
  ;

property_pattern_clause
  : '{' (subpattern (',' subpattern)* ','?)? '}'
  ;

subpattern
- : identifier ':' pattern
+ : subpattern_name ':' pattern
  ;

+subpattern_name
+ : identifier
+ | subpattern_name '.' identifier
+ ;
```

The receiver for each name lookup is the type of the previous member *T0*, starting from the *input type* of the *property_pattern*. if *T* is a nullable type, *T0* is its underlying type, otherwise *T0* is equal to *T*.

For example, a pattern of the form `{ Prop1.Prop2: pattern }` is exactly equivalent to `{ Prop1: { Prop2: pattern } }`.

Note that this will include the null check when *T* is a nullable value type or a reference type. This null check means that the nested properties available will be the properties of *T0*, not of *T*.

Repeated member paths are allowed. The compilation of pattern matching can take advantage of common parts of patterns.

Improved Interpolated Strings

Article • 03/21/2022

> **Note**
>
> This article is a feature specification. The specification represents the *proposed* feature specification. There may be some discrepancies between the feature specification and the completed implementation. Those differences are captured in the pertinent **language design meeting (LDM) notes**. Links to pertinent meetings are included at the bottom of the spec. You can learn more about the process for merging feature speclets into the C# language standard in the article on the **specifications**.

Summary

We introduce a new pattern for creating and using interpolated string expressions to allow for efficient formatting and use in both general `string` scenarios and more specialized scenarios such as logging frameworks, without incurring unnecessary allocations from formatting the string in the framework.

Motivation

Today, string interpolation mainly lowers down to a call to `string.Format`. This, while general purpose, can be inefficient for a number of reasons:

1. It boxes any struct arguments, unless the runtime has happened to introduce an overload of `string.Format` that takes exactly the correct types of arguments in exactly the correct order.
    - This ordering is why the runtime is hesitant to introduce generic versions of the method, as it would lead to combinatoric explosion of generic instantiations of a very common method.
2. It has to allocate an array for the arguments in most cases.
3. There is no opportunity to avoid instantiating the instance if it's not needed. Logging frameworks, for example, will recommend avoiding string interpolation because it will cause a string to be realized that may not be needed, depending on the current log-level of the application.

4. It can never use `Span` or other ref struct types today, because ref structs are not allowed as generic type parameters, meaning that if a user wants to avoid copying to intermediate locations they have to manually format strings.

Internally, the runtime has a type called `ValueStringBuilder` to help deal with the first 2 of these scenarios. They pass a stackalloc'd buffer to the builder, repeatedly call `AppendFormat` with every part, and then get a final string out. If the resulting string goes past the bounds of the stack buffer, they can then move to an array on the heap. However, this type is dangerous to expose directly, as incorrect usage could lead to a rented array to be double-disposed, which then will cause all sorts of undefined behavior in the program as two locations think they have sole access to the rented array. This proposal creates a way to use this type safely from native C# code by just writing an interpolated string literal, leaving written code unchanged while improving every interpolated string that a user writes. It also extends this pattern to allow for interpolated strings passed as arguments to other methods to use a handler pattern, defined by receiver of the method, that will allow things like logging frameworks to avoid allocating strings that will never be needed, and giving C# users familiar, convenient interpolation syntax.

Detailed Design

The handler pattern

We introduce a new handler pattern that can represent an interpolated string passed as an argument to a method. The simple English of the pattern is as follows:

When an *interpolated_string_expression* is passed as an argument to a method, we look at the type of the parameter. If the parameter type has a constructor that can be invoked with 2 int parameters, `literalLength` and `formattedCount`, optionally takes additional parameters specified by an attribute on the original parameter, optionally has an out boolean trailing parameter, and the type of the original parameter has instance `AppendLiteral` and `AppendFormatted` methods that can be invoked for every part of the interpolated string, then we lower the interpolation using that, instead of into a traditional call to `string.Format(formatStr, args)`. A more concrete example is helpful for picturing this:

C#

```
// The handler that will actually "build" the interpolated string"
[InterpolatedStringHandler]
public ref struct TraceLoggerParamsInterpolatedStringHandler
```

```
{
    // Storage for the built-up string

    private bool _logLevelEnabled;

    public TraceLoggerParamsInterpolatedStringHandler(int literalLength, int
formattedCount, Logger logger, out bool handlerIsValid)
    {
        if (!logger._logLevelEnabled)
        {
            handlerIsValid = false;
            return;
        }

        handlerIsValid = true;
        _logLevelEnabled = logger.EnabledLevel;
    }

    public void AppendLiteral(string s)
    {
        // Store and format part as required
    }

    public void AppendFormatted<T>(T t)
    {
        // Store and format part as required
    }
}

// The logger class. The user has an instance of this, accesses it via
static state, or some other access
// mechanism
public class Logger
{
    // Initialization code omitted
    public LogLevel EnabledLevel;

    public void
LogTrace([InterpolatedStringHandlerArguments("")]TraceLoggerParamsInterpolat
edStringHandler handler)
    {
        // Impl of logging
    }
}

Logger logger = GetLogger(LogLevel.Info);

// Given the above definitions, usage looks like this:
var name = "Fred Silberberg";
logger.LogTrace($"{name} will never be printed because info is < trace!");

// This is converted to:
var name = "Fred Silberberg";
var receiverTemp = logger;
var handler = new TraceLoggerParamsInterpolatedStringHandler(literalLength:
```

```
47, formattedCount: 1, receiverTemp, out var handlerIsValid);
if (handlerIsValid)
{
    handler.AppendFormatted(name);
    handler.AppendLiteral(" will never be printed because info is <
trace!");
}
receiverTemp.LogTrace(handler);
```

Here, because `TraceLoggerParamsInterpolatedStringHandler` has a constructor with the correct parameters, we say that the interpolated string has an implicit handler conversion to that parameter, and it lowers to the pattern shown above. The specese needed for this is a bit complicated, and is expanded below.

The rest of this proposal will use `Append...` to refer to either of `AppendLiteral` or `AppendFormatted` in cases when both are applicable.

New attributes

The compiler recognizes the `System.Runtime.CompilerServices.InterpolatedStringHandlerAttribute`:

C#

```
using System;
namespace System.Runtime.CompilerServices
{
    [AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct,
AllowMultiple = false, Inherited = false)]
    public sealed class InterpolatedStringHandlerAttribute : Attribute
    {
        public InterpolatedStringHandlerAttribute()
        {
        }
    }
}
```

This attribute is used by the compiler to determine if a type is a valid interpolated string handler type.

The compiler also recognizes the `System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute`:

C#

```
namespace System.Runtime.CompilerServices
{
```

```
    [AttributeUsage(AttributeTargets.Parameter, AllowMultiple = false,
Inherited = false)]
    public sealed class InterpolatedStringHandlerArgumentAttribute :
Attribute
    {
        public InterpolatedHandlerArgumentAttribute(string argument);
        public InterpolatedHandlerArgumentAttribute(params string[]
arguments);

        public string[] Arguments { get; }
    }
}
```

This attribute is used on parameters, to inform the compiler how to lower an interpolated string handler pattern used in a parameter position.

Interpolated string handler conversion

Type `T` is said to be an *applicable_interpolated_string_handler_type* if it is attributed with `System.Runtime.CompilerServices.InterpolatedStringHandlerAttribute`. There exists an implicit *interpolated_string_handler_conversion* to `T` from an *interpolated_string_expression*, or an *additive_expression* composed entirely of _interpolated_string_expression_s and using only `+` operators.

For simplicity in the rest of this speclet, *interpolated_string_expression* refers to both a simple *interpolated_string_expression*, and to an *additive_expression* composed entirely of _interpolated_string_expression_s and using only `+` operators.

Note that this conversion always exists, regardless of whether there will be later errors when actually attempting to lower the interpolation using the handler pattern. This is done to help ensure that there are predictable and useful errors and that runtime behavior doesn't change based on the content of an interpolated string.

Applicable function member adjustments

We adjust the wording of the applicable function member algorithm (§11.6.4.2) as follows (a new sub-bullet is added to each section, in bold):

A function member is said to be an ***applicable function member*** with respect to an argument list `A` when all of the following are true:

- Each argument in `A` corresponds to a parameter in the function member declaration as described in Corresponding parameters (§11.6.2.2), and any parameter to which no argument corresponds is an optional parameter.

- For each argument in `A`, the parameter passing mode of the argument (i.e., value, `ref`, or `out`) is identical to the parameter passing mode of the corresponding parameter, and
  - for a value parameter or a parameter array, an implicit conversion (§10.2) exists from the argument to the type of the corresponding parameter, or
  - **for a `ref` parameter whose type is a struct type, an implicit *interpolated_string_handler_conversion* exists from the argument to the type of the corresponding parameter, or**
  - for a `ref` or `out` parameter, the type of the argument is identical to the type of the corresponding parameter. After all, a `ref` or `out` parameter is an alias for the argument passed.

For a function member that includes a parameter array, if the function member is applicable by the above rules, it is said to be applicable in its ***normal form***. If a function member that includes a parameter array is not applicable in its normal form, the function member may instead be applicable in its ***expanded form***:

- The expanded form is constructed by replacing the parameter array in the function member declaration with zero or more value parameters of the element type of the parameter array such that the number of arguments in the argument list `A` matches the total number of parameters. If `A` has fewer arguments than the number of fixed parameters in the function member declaration, the expanded form of the function member cannot be constructed and is thus not applicable.
- Otherwise, the expanded form is applicable if for each argument in `A` the parameter passing mode of the argument is identical to the parameter passing mode of the corresponding parameter, and
  - for a fixed value parameter or a value parameter created by the expansion, an implicit conversion (§10.2) exists from the type of the argument to the type of the corresponding parameter, or
  - **for a `ref` parameter whose type is a struct type, an implicit *interpolated_string_handler_conversion* exists from the argument to the type of the corresponding parameter, or**
  - for a `ref` or `out` parameter, the type of the argument is identical to the type of the corresponding parameter.

Important note: this means that if there are 2 otherwise equivalent overloads, that only differ by the type of the *applicable_interpolated_string_handler_type*, these overloads will be considered ambiguous. Further, because we do not see through explicit casts, it is possible that there could arise an unresolvable scenario where both applicable overloads use `InterpolatedStringHandlerArguments` and are totally uncallable without manually performing the handler lowering pattern. We could potentially make changes

to the better function member algorithm to resolve this if we so choose, but this scenario unlikely to occur and isn't a priority to address.

Better conversion from expression adjustments

We change the better conversion from expression (§11.6.4.4) section to the following:

Given an implicit conversion `C1` that converts from an expression `E` to a type `T1`, and an implicit conversion `C2` that converts from an expression `E` to a type `T2`, `C1` is a ***better conversion*** than `C2` if:

1. `E` is a non-constant *interpolated_string_expression*, `C1` is an *implicit_string_handler_conversion*, `T1` is an *applicable_interpolated_string_handler_type*, and `C2` is not an *implicit_string_handler_conversion*, or
2. `E` does not exactly match `T2` and at least one of the following holds:
    - `E` exactly matches `T1` (§11.6.4.4)
    - `T1` is a better conversion target than `T2` (§11.6.4.6)

This does mean that there are some potentially non-obvious overload resolution rules, depending on whether the interpolated string in question is a constant-expression or not. For example:

C#

```
void Log(string s) { ... }
void Log(TraceLoggerParamsInterpolatedStringHandler p) { ... }

Log($""); // Calls Log(string s), because $"" is a constant expression
Log($"{"test"}"); // Calls Log(string s), because $"{"test"}" is a constant
expression
Log($"{1}"); // Calls Log(TraceLoggerParamsInterpolatedStringHandler p),
because $"{1}" is not a constant expression
```

This is introduced so that things that can simply be emitted as constants do so, and don't incur any overhead, while things that cannot be constant use the handler pattern.

InterpolatedStringHandler and Usage

We introduce a new type in `System.Runtime.CompilerServices`: `DefaultInterpolatedStringHandler`. This is a ref struct with many of the same semantics

as `ValueStringBuilder`, intended for direct use by the C# compiler. This struct would look approximately like this:

C#

```
// API Proposal issue: https://github.com/dotnet/runtime/issues/50601
namespace System.Runtime.CompilerServices
{
    [InterpolatedStringHandler]
    public ref struct DefaultInterpolatedStringHandler
    {
        public DefaultInterpolatedStringHandler(int literalLength, int
formattedCount);
        public string ToStringAndClear();

        public void AppendLiteral(string value);

        public void AppendFormatted<T>(T value);
        public void AppendFormatted<T>(T value, string? format);
        public void AppendFormatted<T>(T value, int alignment);
        public void AppendFormatted<T>(T value, int alignment, string?
format);

        public void AppendFormatted(ReadOnlySpan<char> value);
        public void AppendFormatted(ReadOnlySpan<char> value, int alignment
= 0, string? format = null);

        public void AppendFormatted(string? value);
        public void AppendFormatted(string? value, int alignment = 0,
string? format = null);

        public void AppendFormatted(object? value, int alignment = 0,
string? format = null);
    }
}
```

We make a slight change to the rules for the meaning of an *interpolated_string_expression* (§11.7.3 ↗):

If the type of an interpolated string is `string` and the type `System.Runtime.CompilerServices.DefaultInterpolatedStringHandler` exists, and the current context supports using that type, the string is lowered using the handler pattern. The final `string` value is then obtained by calling `ToStringAndClear()` on the handler type. Otherwise, if the type of an interpolated string is `System.IFormattable` or `System.FormattableString` [the rest is unchanged]

The "and the current context supports using that type" rule is intentionally vague to give the compiler leeway in optimizing usage of this pattern. The handler type is likely to be a ref struct type, and ref struct types are normally not permitted in async methods. For

this particular case, the compiler would be allowed to make use the handler if none of the interpolation holes contain an `await` expression, as we can statically determine that the handler type is safely used without additional complicated analysis because the handler will be dropped after the interpolated string expression is evaluated.

~~Open~~ **Question**:

Do we want to instead just make the compiler know about `DefaultInterpolatedStringHandler` and skip the `string.Format` call entirely? It would allow us to hide a method that we don't necessarily want to put in people's faces when they manually call `string.Format`.

Answer: Yes.

~~Open~~ **Question**:

Do we want to have handlers for `System.IFormattable` and `System.FormattableString` as well?

Answer: No.

Handler pattern codegen

In this section, method invocation resolution refers to the steps listed in §11.7.8.2.

Constructor resolution

Given an *applicable_interpolated_string_handler_type* `T` and an *interpolated_string_expression* `i`, method invocation resolution and validation for a valid constructor on `T` is performed as follows:

1. Member lookup for instance constructors is performed on `T`. The resulting method group is called `M`.
2. The argument list `A` is constructed as follows:
   a. The first two arguments are integer constants, representing the literal length of `i`, and the number of *interpolation* components in `i`, respectively.
   b. If `i` is used as an argument to some parameter `pi` in method `M1`, and parameter `pi` is attributed with `System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute`, then for every name `Argx` in the `Arguments` array of that attribute the compiler matches it to a parameter `px` that has the same name. The empty string is matched to the receiver of `M1`.

- If any `Argx` is not able to be matched to a parameter of `M1`, or an `Argx` requests the receiver of `M1` and `M1` is a static method, an error is produced and no further steps are taken.
- Otherwise, the type of every resolved `px` is added to the argument list, in the order specified by the `Arguments` array. Each `px` is passed with the same `ref` semantics as is specified in `M1`.

c. The final argument is a `bool`, passed as an `out` parameter.

3. Traditional method invocation resolution is performed with method group `M` and argument list `A`. For the purposes of method invocation final validation, the context of `M` is treated as a *member_access* through type `T`.

   - If a single-best constructor `F` was found, the result of overload resolution is `F`.
   - If no applicable constructors were found, step 3 is retried, removing the final `bool` parameter from `A`. If this retry also finds no applicable members, an error is produced and no further steps are taken.
   - If no single-best method was found, the result of overload resolution is ambiguous, an error is produced, and no further steps are taken.

4. Final validation on `F` is performed.

   - If any element of `A` occurred lexically after `i`, an error is produced and no further steps are taken.
   - If any `A` requests the receiver of `F`, and `F` is an indexer being used as an *initializer_target* in a *member_initializer*, then an error is reported and no further steps are taken.

Note: the resolution here intentionally do *not* use the actual expressions passed as other arguments for `Argx` elements. We only consider the types post-conversion. This makes sure that we don't have double-conversion issues, or unexpected cases where a lambda is bound to one delegate type when passed to `M1` and bound to a different delegate type when passed to `M`.

Note: We report an error for indexers uses as member initializers because of the order of evaluation for nested member initializers. Consider this code snippet:

```
C#
```

```
var x1 = new C1 { C2 = { [GetString()] = { A = 2, B = 4 } } };

/* Lowering:
```

```
__c1 = new C1();
string argTemp = GetString();
__c1.C2[argTemp][1] = 2;
__c1.C2[argTemp][3] = 4;

Prints:
GetString
get_C2
get_C2
*/

string GetString()
{
    Console.WriteLine("GetString");
    return "";
}

class C1
{
    private C2 c2 = new C2();
    public C2 C2 { get { Console.WriteLine("get_C2"); return c2; } set { } }
}

class C2
{
    public C3 this[string s]
    {
        get => new C3();
        set { }
    }
}

class C3
{
    public int A
    {
        get => 0;
        set { }
    }
    public int B
    {
        get => 0;
        set { }
    }
}
```

The arguments to `__c1.C2[]` are evaluated *before* the receiver of the indexer. While we could come up with a lowering that works for this scenario (either by creating a temp for `__c1.C2` and sharing it across both indexer invocations, or only using it for the first indexer invocation and sharing the argument across both invocations) we think that any lowering would be confusing for what we believe is a pathological scenario. Therefore, we forbid the scenario entirely.

~~Open Question~~:

If we use a constructor instead of `Create`, we'd improve runtime codegen, at the expense of narrowing the pattern a bit.

Answer: We will restrict to constructors for now. We can revisit adding a general `Create` method later if the scenario arises.

`Append...` method overload resolution

Given an *applicable_interpolated_string_handler_type* `T` and an *interpolated_string_expression* `i`, overload resolution for a set of valid `Append...` methods on `T` is performed as follows:

1. If there are any *interpolated_regular_string_character* components in `i`:
   a. Member lookup on `T` with the name `AppendLiteral` is performed. The resulting method group is called `Ml`.
   b. The argument list `Al` is constructed with one value parameter of type `string`.
   c. Traditional method invocation resolution is performed with method group `Ml` and argument list `Al`. For the purposes of method invocation final validation, the context of `Ml` is treated as a *member_access* through an instance of `T`.
      - If a single-best method `Fi` is found and no errors were produced, the result of method invocation resolution is `Fi`.
      - Otherwise, an error is reported.
2. For every *interpolation* `ix` component of `i`:
   a. Member lookup on `T` with the name `AppendFormatted` is performed. The resulting method group is called `Mf`.
   b. The argument list `Af` is constructed:
      i. The first parameter is the `expression` of `ix`, passed by value.
      ii. If `ix` directly contains a *constant_expression* component, then an integer value parameter is added, with the name `alignment` specified.
      iii. If `ix` is directly followed by an *interpolation_format*, then a string value parameter is added, with the name `format` specified.
   c. Traditional method invocation resolution is performed with method group `Mf` and argument list `Af`. For the purposes of method invocation final validation, the context of `Mf` is treated as a *member_access* through an instance of `T`.

    - If a single-best method `Fi` is found, the result of method invocation resolution is `Fi`.
    - Otherwise, an error is reported.

3. Finally, for every `Fi` discovered in steps 1 and 2, final validation is performed:

    - If any `Fi` does not return `bool` by value or `void`, an error is reported.
    - If all `Fi` do not return the same type, an error is reported.

Note that these rules do not permit extension methods for the `Append...` calls. We could consider enabling that if we choose, but this is analogous to the enumerator pattern, where we allow `GetEnumerator` to be an extension method, but not `Current` or `MoveNext()`.

These rules *do* permit default parameters for the `Append...` calls, which will work with things like `CallerLineNumber` or `CallerArgumentExpression` (when supported by the language).

We have separate overload lookup rules for base elements vs interpolation holes because some handlers will want to be able to understand the difference between the components that were interpolated and the components that were part of the base string.

~~Open~~ Question

Some scenarios, like structured logging, want to be able to provide names for interpolation elements. For example, today a logging call might look like `Log("{name} bought {itemCount} items", name, items.Count);`. The names inside the `{}` provide important structure information for loggers that help with ensuring output is consistent and uniform. Some cases might be able to reuse the `:format` component of an interpolation hole for this, but many loggers already understand format specifiers and have existing behavior for output formatting based on this info. Is there some syntax we can use to enable putting these named specifiers in?

Some cases may be able to get away with `CallerArgumentExpression`, provided that support does land in C# 10. But for cases that invoke a method/property, that may not be sufficient.

Answer:

While there are some interesting parts to templated strings we could explore in an orthogonal language feature, we don't think a specific syntax here has much benefit over solutions such as using a tuple: `$"{("StructuredCategory", myExpression)}"`.

Performing the conversion

Given an *applicable_interpolated_string_handler_type* `T` and an *interpolated_string_expression* `i` that had a valid constructor `Fc` and `Append...` methods `Fa` resolved, lowering for `i` is performed as follows:

1. Any arguments to `Fc` that occur lexically before `i` are evaluated and stored into temporary variables in lexical order. In order to preserve lexical ordering, if `i` occurred as part of a larger expression `e`, any components of `e` that occurred before `i` will be evaluated as well, again in lexical order.
2. `Fc` is called with the length of the interpolated string literal components, the number of *interpolation* holes, any previously evaluated arguments, and a `bool` out argument (if `Fc` was resolved with one as the last parameter). The result is stored into a temporary value `ib`.
   a. The length of the literal components is calculated after replacing any *open_brace_escape_sequence* with a single `{`, and any *close_brace_escape_sequence* with a single `}`.
3. If `Fc` ended with a `bool` out argument, a check on that `bool` value is generated. If true, the methods in `Fa` will be called. Otherwise, they will not be called.
4. For every `Fax` in `Fa`, `Fax` is called on `ib` with either the current literal component or *interpolation* expression, as appropriate. If `Fax` returns a `bool`, the result is logically anded with all preceding `Fax` calls.
   a. If `Fax` is a call to `AppendLiteral`, the literal component is unescaped by replacing any *open_brace_escape_sequence* with a single `{`, and any *close_brace_escape_sequence* with a single `}`.
5. The result of the conversion is `ib`.

Again, note that arguments passed to `Fc` and arguments passed to `e` are the same temp. Conversions may occur on top of the temp to convert to a form that `Fc` requires, but for example lambdas cannot be bound to a different delegate type between `Fc` and `e`.

~~Open~~ Question

This lowering means that subsequent parts of the interpolated string after a false-returning `Append...` call don't get evaluated. This could potentially be very confusing, particularly if the format hole is side-effecting. We could instead evaluate all format holes first, then repeatedly call `Append...` with the results, stopping if it returns false. This would ensure that all expressions get evaluated as one might expect, but we call as

few methods as we need to. While the partial evaluation might be desirable for some more advanced cases, it is perhaps non-intuitive for the general case.

Another alternative, if we want to always evaluate all format holes, is to remove the `Append...` version of the API and just do repeated `Format` calls. The handler can track whether it should just be dropping the argument and immediately returning for this version.

Answer: We will have conditional evaluation of the holes.

~~Open~~ Question

Do we need to dispose of disposable handler types, and wrap calls with try/finally to ensure that Dispose is called? For example, the interpolated string handler in the bcl might have a rented array inside it, and if one of the interpolation holes throws an exception during evaluation, that rented array could be leaked if it wasn't disposed.

Answer: No. handlers can be assigned to locals (such as `MyHandler handler = $"{MyCode()};`), and the lifetime of such handlers is unclear. Unlike foreach enumerators, where the lifetime is obvious and no user-defined local is created for the enumerator.

Impact on nullable reference types

To minimize complexity of the implementation, we have a few limitations on how we perform nullable analysis on interpolated string handler constructors used as arguments to a method or indexer. In particular, we do not flow information from the constructor back through to the original slots of parameters or arguments from the original context, and we do not use constructor parameter types to inform generic type inference for type parameters in the containing method. An example of where this can have an impact is:

C#

```
string s = "";
C c = new C();
c.M(s, $"", c.ToString(), s.ToString()); // No warnings on c.ToString() or
s.ToString(), as the `MaybeNull` does not flow back.

public class C
{
    public void M(string s1, [InterpolatedStringHandlerArgument("", "s1")]
CustomHandler c1, string s2, string s3) { }
}

[InterpolatedStringHandler]
public partial struct CustomHandler
```

```
{
    public CustomHandler(int literalLength, int formattedCount, [MaybeNull]
C c, [MaybeNull] string s) : this()
    {
    }
}
```

C#

```
string? s = null;
M(s, $""); // Infers `string` for `T` because of the `T?` parameter, not
`string?`, as flow analysis does not consider the unannotated `T` parameter
of the constructor

void M<T>(T? t, [InterpolatedStringHandlerArgument("s1")] CustomHandler<T>
c) { }

[InterpolatedStringHandler]
public partial struct CustomHandler<T>
{
    public CustomHandler(int literalLength, int formattedCount, T t) :
this()
    {
    }
}
```

Other considerations

Allow `string` types to be convertible to handlers as well

For type author simplicity, we could consider allowing expressions of type `string` to be implicitly-convertible to *applicable_interpolated_string_handler_types*. As proposed today, authors will likely need to overload on both that handler type and regular `string` types, so their users don't have to understand the difference. This may be an annoying and non-obvious overhead, as a `string` expression can be viewed as an interpolation with `expression.Length` prefilled length and 0 holes to be filled.

This would allow new APIs to only expose a handler, without also having to expose a `string`-accepting overload. However, it won't get around the need for changes to better conversion from expression, so while it would work it may be unnecessary overhead.

Answer:

We think that this could end up being confusing, and there's an easy workaround for custom handler types: add a user-defined conversion from string.

Incorporating spans for heap-less strings

`ValueStringBuilder` as it exists today has 2 constructors: one that takes a count, and allocates on the heap eagerly, and one that takes a `Span<char>`. That `Span<char>` is usually a fixed size in the runtime codebase, around 250 elements on average. To truly replace that type, we should consider an extension to this where we also recognize `GetInterpolatedString` methods that take a `Span<char>`, instead of just the count version. However, we see a few potential thorny cases to resolve here:

- We don't want to stackalloc repeatedly in a hot loop. If we were to do this extension to the feature, we'd likely want to share the stackalloc'd span between loop iterations. We know this is safe, as `Span<T>` is a ref struct that can't be stored on the heap, and users would have to be pretty devious to manage to extract a reference to that `Span` (such as creating a method that accepts such a handler then deliberately retrieving the `Span` from the handler and returning it to the caller). However, allocating ahead of time produces other questions:
  - Should we eagerly stackalloc? What if the loop is never entered, or exits before it needs the space?
  - If we don't eagerly stackalloc, does that mean we introduce a hidden branch on every loop? Most loops likely won't care about this, but it could affect some tight loops that don't want to pay the cost.
- Some strings can be quite big, and the appropriate amount to `stackalloc` is dependent on a number of factors, including runtime factors. We don't really want the C# compiler and specification to have to determine this ahead of time, so we'd want to resolve https://github.com/dotnet/runtime/issues/25423 and add an API for the compiler to call in these cases. It also adds more pros and cons to the points from the previous loop, where we don't want to potentially allocate large arrays on the heap many times or before one is needed.

Answer:

This is out of scope for C# 10. We can look at this in general when we look at the more general `params Span<T>` feature.

Non-try version of the API

For simplicity, this spec currently just proposes recognizing a `Append...` method, and things that always succeed (like `InterpolatedStringHandler`) would always return true

from the method. This was done to support partial formatting scenarios where the user wants to stop formatting if an error occurs or if it's unnecessary, such as the logging case, but could potentially introduce a bunch of unnecessary branches in standard interpolated string usage. We could consider an addendum where we use just `FormatX` methods if no `Append...` method is present, but it does present questions about what we do if there's a mix of both `Append...` and `FormatX` calls.

Answer:

We want the non-try version of the API. The proposal has been updated to reflect this.

Passing previous arguments to the handler

There is unfortunate lack of symmetry in the proposal at it currently exists: invoking an extension method in reduced form produces different semantics than invoking the extension method in normal form. This is different from most other locations in the language, where reduced form is just a sugar. We propose adding an attribute to the framework that we will recognize when binding a method, that informs the compiler that certain parameters should be passed to the constructor on the handler. Usage looks like this:

C#

```
namespace System.Runtime.CompilerServices
{
    [AttributeUsage(AttributeTargets.Parameter, AllowMultiple = false,
Inherited = false)]
    public sealed class InterpolatedStringHandlerArgumentAttribute :
Attribute
    {
        public InterpolatedStringHandlerArgumentAttribute(string argument);
        public InterpolatedStringHandlerArgumentAttribute(params string[]
arguments);

        public string[] Arguments { get; }
    }
}
```

Usage of this is then:

C#

```
namespace System
{
    public sealed class String
    {
```

```
        public static string Format(IFormatProvider? provider,
[InterpolatedStringHandlerArgument("provider")] ref
DefaultInterpolatedStringHandler handler);
        …
    }
}

namespace System.Runtime.CompilerServices
{
    public ref struct DefaultInterpolatedStringHandler
    {
        public DefaultInterpolatedStringHandler(int baseLength, int
holeCount, IFormatProvider? provider); // additional factory
        …
    }
}

var formatted = string.Format(CultureInfo.InvariantCulture, $"{X} = {Y}");

// Is lowered to

var tmp1 = CultureInfo.InvariantCulture;
var handler = new DefaultInterpolatedStringHandler(3, 2, tmp1);
handler.AppendFormatted(X);
handler.AppendLiteral(" = ");
handler.AppendFormatted(Y);
var formatted = string.Format(tmp1, handler);
```

The questions we need to answer:

1. Do we like this pattern in general?
2. Do we want to allow these arguments to come from after the handler parameter? Some existing patterns in the BCL, such as `Utf8Formatter`, put the value to be formatted *before* the thing needed to format into. To fit in best with these patterns, we'd likely want to allow this, but we need to decide if this out-of-order evaluate is ok.

Answer:

We want to support this. The spec has been updated to reflect this. Arguments will be required to be specified in lexical order at the call site, and if a needed argument to the create method is specified after the interpolated string literal, an error is produced.

`await` usage in interpolation holes

Because `$"{await A()}"` is a valid expression today, we need to rationalize how interpolation holes with await. We could solve this with a few rules:

1. If an interpolated string used as a `string`, `IFormattable`, or `FormattableString` has an `await` in an interpolation hole, fall back to old-style formatter.
2. If an interpolated string is subject to an *implicit_string_handler_conversion* and *applicable_interpolated_string_handler_type* is a `ref struct`, `await` is not allowed to be used in the format holes.

Fundamentally, this desugaring could use a ref struct in an async method as long as we guarantee that the `ref struct` will not need to be saved to the heap, which should be possible if we forbid `await`s in the interpolation holes.

Alternatively, we could simply make all handler types non-ref structs, including the framework handler for interpolated strings. This would, however, preclude us from someday recognizing a `Span` version that does not need to allocate any scratch space at all.

Answer:

We will treat interpolated string handlers the same as any other type: this means that if the handler type is a ref struct and the current context doesn't allow the usage of ref structs, it is illegal to use handler here. The spec around lowering of string literals used as strings is intentionally vague to allow the compiler to decide on what rules it deems appropriate, but for custom handler types they will have to follow the same rules as the rest of the language.

Handlers as ref parameters

Some handlers might want to be passed as ref parameters (either `in` or `ref`). Should we allow either? And if so, what will a `ref` handler look like? `ref $""` is confusing, as you're not actually passing the string by ref, you're passing the handler that is created from the ref by ref, and has similar potential issues with async methods.

Answer:

We want to support this. The spec has been updated to reflect this. The rules should reflect the same rules that apply to extension methods on value types.

Interpolated strings through binary expressions and conversions

Because this proposal makes interpolated strings context sensitive, we would like to allow the compiler to treat a binary expression composed entirely of interpolated

strings, or an interpolated string subjected to a cast, as an interpolated string literal for the purposes of overload resolution. For example, take the following scenario:

```
C#
```

```
struct Handler1
{
    public Handler1(int literalLength, int formattedCount, C c) => ...;
    // AppendX... methods as necessary
}
struct Handler2
{
    public Handler2(int literalLength, int formattedCount, C c) => ...;
    // AppendX... methods as necessary
}

class C
{
    void M(Handler1 handler) => ...;
    void M(Handler2 handler) => ...;
}

c.M($"{X}"); // Ambiguous between the M overloads
```

This would be ambiguous, necessitating a cast to either `Handler1` or `Handler2` in order to resolve. However, in making that cast, we would potentially throw away the information that there is context from the method receiver, meaning that the cast would fail because there is nothing to fill in the information of `c`. A similar issue arises with binary concatenation of strings: the user could want to format the literal across several lines to avoid line wrapping, but would not be able to because that would no longer be an interpolated string literal convertible to the handler type.

To resolve these cases, we make the following changes:

- An *additive_expression* composed entirely of *interpolated_string_expressions* and using only `+` operators is considered to be an *interpolated_string_literal* for the purposes of conversions and overload resolution. The final interpolated string is created by logically concatinating all individual *interpolated_string_expression* components, from left to right.
- A *cast_expression* or a *relational_expression* with operator `as` whose operand is an *interpolated_string_expressions* is considered an *interpolated_string_expressions* for the purposes of conversions and overload resolution.

Open Questions:

Do we want to do this? We don't do this for `System.FormattableString`, for example, but that can be broken out onto a different line, whereas this can be context-dependent and therefore not able to be broken out into a different line. There are also no overload resolution concerns with `FormattableString` and `IFormattable`.

Answer:

We think that this is a valid use case for additive expressions, but that the cast version is not compelling enough at this time. We can add it later if necessary. The spec has been updated to reflect this decision.

Other use cases

See https://github.com/dotnet/runtime/issues/50635 for examples of proposed handler APIs using this pattern.

Constant Interpolated Strings

Article • 06/23/2023

> **Note**
>
> This article is a feature specification. The specification represents the *proposed* feature specification. There may be some discrepancies between the feature specification and the completed implementation. Those differences are captured in the pertinent **language design meeting (LDM) notes**. Links to pertinent meetings are included at the bottom of the spec. You can learn more about the process for merging feature speclets into the C# language standard in the article on the **specifications**.

Summary

Enables constants to be generated from interpolated strings of type string constant.

Motivation

The following code is already legal:

```
public class C
{
    const string S1 = "Hello world";
    const string S2 = "Hello" + " " + "World";
    const string S3 = S1 + " Kevin, welcome to the team!";
}
```

However, there have been many community requests to make the following also legal:

```
public class C
{
    const string S1 = $"Hello world";
    const string S2 = $"Hello{" "}World";
    const string S3 = $"{S1} Kevin, welcome to the team!";
}
```

This proposal represents the next logical step for constant string generation, where existing string syntax that works in other situations is made to work for constants.

Detailed design

The following represent the updated specifications for constant expressions under this new proposal. Current specifications from which this was directly based on can be found in §11.20.

Constant Expressions

A *constant_expression* is an expression that can be fully evaluated at compile-time.

antlr

```
constant_expression
    : expression
    ;
```

A constant expression must be the `null` literal or a value with one of the following types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`, `object`, `string`, or any enumeration type. Only the following constructs are permitted in constant expressions:

- Literals (including the `null` literal).
- References to `const` members of class and struct types.
- References to members of enumeration types.
- References to `const` parameters or local variables
- Parenthesized sub-expressions, which are themselves constant expressions.
- Cast expressions, provided the target type is one of the types listed above.
- `checked` and `unchecked` expressions
- Default value expressions
- Nameof expressions
- The predefined `+`, `-`, `!`, and `~` unary operators.
- The predefined `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `|`, `^`, `&&`, `||`, `==`, `!=`, `<`, `>`, `<=`, and `>=` binary operators, provided each operand is of a type listed above.
- The `?:` conditional operator.
- *Interpolated strings `$"{}"`, provided that all components are constant expressions of type `string` and all interpolated components lack alignment and format specifiers.*

The following conversions are permitted in constant expressions:

- Identity conversions
- Numeric conversions
- Enumeration conversions
- Constant expression conversions
- Implicit and explicit reference conversions, provided that the source of the conversions is a constant expression that evaluates to the null value.

Other conversions including boxing, unboxing and implicit reference conversions of non-null values are not permitted in constant expressions. For example:

```
C#
```

```
class C
{
    const object i = 5;          // error: boxing conversion not permitted
    const object str = "hello"; // error: implicit reference conversion
}
```

the initialization of i is an error because a boxing conversion is required. The initialization of str is an error because an implicit reference conversion from a non-null value is required.

Whenever an expression fulfills the requirements listed above, the expression is evaluated at compile-time. This is true even if the expression is a sub-expression of a larger expression that contains non-constant constructs.

The compile-time evaluation of constant expressions uses the same rules as run-time evaluation of non-constant expressions, except that where run-time evaluation would have thrown an exception, compile-time evaluation causes a compile-time error to occur.

Unless a constant expression is explicitly placed in an `unchecked` context, overflows that occur in integral-type arithmetic operations and conversions during the compile-time evaluation of the expression always cause compile-time errors (§11.20).

Constant expressions occur in the contexts listed below. In these contexts, a compile-time error occurs if an expression cannot be fully evaluated at compile-time.

- Constant declarations (§14.4).
- Enumeration member declarations (§18.4).
- Default arguments of formal parameter lists (§14.6.2)
- `case` labels of a `switch` statement (§12.8.3).
- `goto case` statements (§12.10.4).

- Dimension lengths in an array creation expression (§11.7.15.5) that includes an initializer.
- Attributes (§21).

An implicit constant expression conversion (§10.2.11) permits a constant expression of type `int` to be converted to `sbyte`, `byte`, `short`, `ushort`, `uint`, or `ulong`, provided the value of the constant expression is within the range of the destination type.

Drawbacks

This proposal adds additional complexity to the compiler in exchange for broader applicability of interpolated strings. As these strings are fully evaluated at compile time, the valuable automatic formatting features of interpolated strings are less neccesary. Most use cases can be largely replicated through the alternatives below.

Alternatives

The current `+` operator for string concatenation can combine strings in a similar manner to the current proposal.

Unresolved questions

What parts of the design are still undecided?

Design meetings

Link to design notes that affect this proposal, and describe in one sentence for each what changes they led to.

Lambda improvements

Article • 06/23/2023

> **Note**
>
> This article is a feature specification. The specification represents the *proposed* feature specification. There may be some discrepancies between the feature specification and the completed implementation. Those differences are captured in the pertinent **language design meeting (LDM) notes**. Links to pertinent meetings are included at the bottom of the spec. You can learn more about the process for merging feature speclets into the C# language standard in the article on the **specifications**.

Summary

Proposed changes:

1. Allow lambdas with attributes
2. Allow lambdas with explicit return type
3. Infer a natural delegate type for lambdas and method groups

Motivation

Support for attributes on lambdas would provide parity with methods and local functions.

Support for explicit return types would provide symmetry with lambda parameters where explicit types can be specified. Allowing explicit return types would also provide control over compiler performance in nested lambdas where overload resolution must bind the lambda body currently to determine the signature.

A natural type for lambda expressions and method groups will allow more scenarios where lambdas and method groups may be used without an explicit delegate type, including as initializers in `var` declarations.

Requiring explicit delegate types for lambdas and method groups has been a friction point for customers, and has become an impediment to progress in ASP.NET with recent work on MapAction.

ASP.NET MapAction without proposed changes (`MapAction()` takes a `System.Delegate` argument):

C#

```
[HttpGet("/")] Todo GetTodo() => new(Id: 0, Name: "Name");
app.MapAction((Func<Todo>)GetTodo);

[HttpPost("/")] Todo PostTodo([FromBody] Todo todo) => todo;
app.MapAction((Func<Todo, Todo>)PostTodo);
```

ASP.NET MapAction with natural types for method groups:

C#

```
[HttpGet("/")] Todo GetTodo() => new(Id: 0, Name: "Name");
app.MapAction(GetTodo);

[HttpPost("/")] Todo PostTodo([FromBody] Todo todo) => todo;
app.MapAction(PostTodo);
```

ASP.NET MapAction with attributes and natural types for lambda expressions:

C#

```
app.MapAction([HttpGet("/")] () => new Todo(Id: 0, Name: "Name"));
app.MapAction([HttpPost("/")] ([FromBody] Todo todo) => todo);
```

Attributes

Attributes may be added to lambda expressions and lambda parameters. To avoid ambiguity between method attributes and parameter attributes, a lambda expression with attributes must use a parenthesized parameter list. Parameter types are not required.

C#

```
f = [A] () => { };        // [A] lambda
f = [return:A] x => x;    // syntax error at '=>'
f = [return:A] (x) => x;  // [A] lambda
f = [A] static x => x;    // syntax error at '=>'

f = ([A] x) => x;         // [A] x
f = ([A] ref int x) => x; // [A] x
```

Multiple attributes may be specified, either comma-separated within the same attribute list or as separate attribute lists.

```
C#

var f = [A1, A2][A3] () => { };    // ok
var g = ([A1][A2, A3] int x) => x; // ok
```

Attributes are not supported for *anonymous methods* declared with `delegate { }` syntax.

```
C#

f = [A] delegate { return 1; };         // syntax error at 'delegate'
f = delegate ([A] int x) { return x; }; // syntax error at '['
```

The parser will look ahead to differentiate a collection initializer with an element assignment from a collection initializer with a lambda expression.

```
C#

var y = new C { [A] = x };    // ok: y[A] = x
var z = new C { [A] x => x }; // ok: z[0] = [A] x => x
```

The parser will treat `?[` as the start of a conditional element access.

```
C#

x = b ? [A];               // ok
y = b ? [A] () => { } : z; // syntax error at '('
```

Attributes on the lambda expression or lambda parameters will be emitted to metadata on the method that maps to the lambda.

In general, customers should not depend on how lambda expressions and local functions map from source to metadata. How lambdas and local functions are emitted can, and has, changed between compiler versions.

The changes proposed here are targeted at the `Delegate` driven scenario. It should be valid to inspect the `MethodInfo` associated with a `Delegate` instance to determine the signature of the lambda expression or local function including any explicit attributes and additional metadata emitted by the compiler such as default parameters. This allows teams such as ASP.NET to make available the same behaviors for lambdas and local functions as ordinary methods.

Explicit return type

An explicit return type may be specified before the parenthesized parameter list.

```
C#
```

```
f = T () => default;                    // ok
f = short x => 1;                       // syntax error at '=>'
f = ref int (ref int x) => ref x;       // ok
f = static void (_) => { };             // ok
f = async async (async async) => async; // ok?
```

The parser will look ahead to differentiate a method call `T()` from a lambda expression `T () => e`.

Explicit return types are not supported for anonymous methods declared with `delegate { }` syntax.

```
C#
```

```
f = delegate int { return 1; };         // syntax error
f = delegate int (int x) { return x; }; // syntax error
```

Method type inference should make an exact inference from an explicit lambda return type.

```
C#
```

```
static void F<T>(Func<T, T> f) { ... }
F(int (i) => i); // Func<int, int>
```

Variance conversions are not allowed from lambda return type to delegate return type (matching similar behavior for parameter types).

```
C#
```

```
Func<object> f1 = string () => null; // error
Func<object?> f2 = object () => x;   // warning
```

The parser allows lambda expressions with `ref` return types within expressions without additional parentheses.

```
C#
```

```
d = ref int () => x; // d = (ref int () => x)
F(ref int () => x);  // F((ref int () => x))
```

`var` cannot be used as an explicit return type for lambda expressions.

C#

```
class var { }

d = var (var v) => v;              // error: contextual keyword 'var' cannot
be used as explicit lambda return type
d = @var (var v) => v;             // ok
d = ref var (ref var v) => ref v;  // error: contextual keyword 'var' cannot
be used as explicit lambda return type
d = ref @var (ref var v) => ref v; // ok
```

Natural (function) type

An *anonymous function* expression (§11.16) (a *lambda expression* or an *anonymous method*) has a natural type if the parameters types are explicit and the return type is either explicit or can be inferred (see §11.6.3.13).

A *method group* has a natural type if all candidate methods in the method group have a common signature. (If the method group may include extension methods, the candidates include the containing type and all extension method scopes.)

The natural type of an anonymous function expression or method group is a *function_type*. A *function_type* represents a method signature: the parameter types and ref kinds, and return type and ref kind. Anonymous function expressions or method groups with the same signature have the same *function_type*.

Function_types are used in a few specific contexts only:

- implicit and explicit conversions
- method type inference (§11.6.3) and best common type (§11.6.3.15)
- `var` initializers

A *function_type* exists at compile time only: *function_types* do not appear in source or metadata.

Conversions

From a *function_type* `F` there are implicit *function_type* conversions:

- To a *function_type* `G` if the parameters and return types of `F` are variance-convertible to the parameters and return type of `G`
- To `System.MulticastDelegate` or base classes or interfaces of `System.MulticastDelegate`
- To `System.Linq.Expressions.Expression` or `System.Linq.Expressions.LambdaExpression`

Anonymous function expressions and method groups already have *conversions from expression* to delegate types and expression tree types (see anonymous function conversions §10.7 and method group conversions §10.8). Those conversions are sufficient for converting to strongly-typed delegate types and expression tree types. The *function_type* conversions above add *conversions from type* to the base types only: `System.MulticastDelegate`, `System.Linq.Expressions.Expression`, etc.

There are no conversions to a *function_type* from a type other than a *function_type*. There are no explicit conversions for *function_types* since *function_types* cannot be referenced in source.

A conversion to `System.MulticastDelegate` or base type or interface realizes the anonymous function or method group as an instance of an appropriate delegate type. A conversion to `System.Linq.Expressions.Expression<TDelegate>` or base type realizes the lambda expression as an expression tree with an appropriate delegate type.

C#

```
Delegate d = delegate (object obj) { }; // Action<object>
Expression e = () => "";                // Expression<Func<string>>
object o = "".Clone;                    // Func<object>
```

Function_type conversions are not implicit or explicit standard conversions §10.4 and are not considered when determining whether a user-defined conversion operator is applicable to an anonymous function or method group. From evaluation of user defined conversions §10.5.3:

> For a conversion operator to be applicable, it must be possible to perform a standard conversion (§10.4) from the source type to the operand type of the operator, and it must be possible to perform a standard conversion from the result type of the operator to the target type.

C#

```
class C
{
    public static implicit operator C(Delegate d) { ... }
}

C c;
c = () => 1;      // error: cannot convert lambda expression to type 'C'
c = (C)(() => 2); // error: cannot convert lambda expression to type 'C'
```

A warning is reported for an implicit conversion of a method group to `object`, since the conversion is valid but perhaps unintentional.

C#

```
Random r = new Random();
object obj;
obj = r.NextDouble;         // warning: Converting method group to 'object'.
Did you intend to invoke the method?
obj = (object)r.NextDouble; // ok
```

Type inference

The existing rules for type inference are mostly unchanged (see §11.6.3). There are however a **couple of changes** below to specific phases of type inference.

First phase

The first phase (§11.6.3.2) allows an anonymous function to bind to `Ti` even if `Ti` is not a delegate or expression tree type (perhaps a type parameter constrained to `System.Delegate` for instance).

> For each of the method arguments `Ei`:
>
> - If `Ei` is an anonymous function **and `Ti` is a delegate type or expression tree type**, an *explicit parameter type inference* is made from `Ei` to `Ti` **and an *explicit return type inference* is made from `Ei` to `Ti`.**
> - Otherwise, if `Ei` has a type `U` and `xi` is a value parameter then a *lower-bound inference* is made *from* `U` *to* `Ti`.
> - Otherwise, if `Ei` has a type `U` and `xi` is a `ref` or `out` parameter then an *exact inference* is made *from* `U` *to* `Ti`.
> - Otherwise, no inference is made for this argument.

Explicit return type inference

An *explicit return type inference* is made *from* an expression `E` *to* a type `T` in the following way:

- **If `E` is an anonymous function with explicit return type `Ur` and `T` is a delegate type or expression tree type with return type `Vr` then an *exact inference* (§11.6.3.9) is made *from* `Ur` *to* `Vr`.**

Fixing

Fixing (§11.6.3.12) ensures other conversions are preferred over *function_type* conversions. (Lambda expressions and method group expressions only contribute to lower bounds so handling of *function_types* is needed for lower bounds only.)

An *unfixed* type variable `Xi` with a set of bounds is *fixed* as follows:

- The set of *candidate types* `Uj` starts out as the set of all types in the set of bounds for `Xi` **where function types are ignored in lower bounds if there any types that are not function types**.
- We then examine each bound for `Xi` in turn: For each exact bound `U` of `Xi` all types `Uj` which are not identical to `U` are removed from the candidate set. For each lower bound `U` of `Xi` all types `Uj` to which there is *not* an implicit conversion from `U` are removed from the candidate set. For each upper bound `U` of `Xi` all types `Uj` from which there is *not* an implicit conversion to `U` are removed from the candidate set.
- If among the remaining candidate types `Uj` there is a unique type `V` from which there is an implicit conversion to all the other candidate types, then `Xi` is fixed to `V`.
- Otherwise, type inference fails.

Best common type

Best common type (§11.6.3.15) is defined in terms of type inference so the type inference changes above apply to best common type as well.

C#

```
var fs = new[] { (string s) => s.Length; (string s) => int.Parse(s) } //
Func<string, int>[]
```

`var`

Anonymous functions and method groups with function types can be used as initializers in `var` declarations.

```
C#
```

```
var f1 = () => default;           // error: cannot infer type
var f2 = x => x;                  // error: cannot infer type
var f3 = () => 1;                 // System.Func<int>
var f4 = string () => null;       // System.Func<string>
var f5 = delegate (object o) { }; // System.Action<object>

static void F1() { }
static void F1<T>(this T t) { }
static void F2(this string s) { }

var f6 = F1;    // error: multiple methods
var f7 = "".F1; // System.Action
var f8 = F2;    // System.Action<string>
```

Function types are not used in assignments to discards.

```
C#
```

```
d = () => 0; // ok
_ = () => 1; // error
```

Delegate types

The delegate type for the anonymous function or method group with parameter types `P1, ..., Pn` and return type `R` is:

- if any parameter or return value is not by value, or there are more than 16 parameters, or any of the parameter types or return are not valid type arguments (say, `(int* p) => { }`), then the delegate is a synthesized `internal` anonymous delegate type with signature that matches the anonymous function or method group, and with parameter names `arg1, ..., argn` or `arg` if a single parameter;
- if `R` is `void`, then the delegate type is `System.Action<P1, ..., Pn>`;
- otherwise the delegate type is `System.Func<P1, ..., Pn, R>`.

The compiler may allow more signatures to bind to `System.Action<>` and `System.Func<>` types in the future (if `ref struct` types are allowed type arguments for instance).

`modopt()` or `modreq()` in the method group signature are ignored in the corresponding delegate type.

If two anonymous functions or method groups in the same compilation require synthesized delegate types with the same parameter types and modifiers and the same return type and modifiers, the compiler will use the same synthesized delegate type.

Overload resolution

Better function member (§11.6.4.3) is updated to prefer members where none of the conversions and none of the type arguments involved inferred types from lambda expressions or method groups.

> ### Better function member
>
> ... Given an argument list `A` with a set of argument expressions `{E1, E2, ..., En}` and two applicable function members `Mp` and `Mq` with parameter types `{P1, P2, ..., Pn}` and `{Q1, Q2, ..., Qn}`, `Mp` is defined to be a ***better function member*** than `Mq` if
>
> 1. **for each argument, the implicit conversion from `Ex` to `Px` is not a *function_type_conversion*, and**
>
>   - **`Mp` is a non-generic method or `Mp` is a generic method with type parameters `{X1, X2, ..., Xp}` and for each type parameter `Xi` the type argument is inferred from an expression or from a type other than a *function_type*, and**
>   - **for at least one argument, the implicit conversion from `Ex` to `Qx` is a *function_type_conversion*, or `Mq` is a generic method with type parameters `{Y1, Y2, ..., Yq}` and for at least one type parameter `Yi` the type argument is inferred from a *function_type*, or**
>
> 2. for each argument, the implicit conversion from `Ex` to `Qx` is not better than the implicit conversion from `Ex` to `Px`, and for at least one argument, the conversion from `Ex` to `Px` is better than the conversion from `Ex` to `Qx`.

Better conversion from expression (§11.6.4.4) is updated to prefer conversions that did not involve inferred types from lambda expressions or method groups.

> ### Better conversion from expression

> Given an implicit conversion `C1` that converts from an expression `E` to a type `T1`, and an implicit conversion `C2` that converts from an expression `E` to a type `T2`, `C1` is a ***better conversion*** than `C2` if:
>
> 1. **`C1` is not a *function_type_conversion* and `C2` is a *function_type_conversion*, or**
> 2. `E` is a non-constant *interpolated_string_expression*, `C1` is an *implicit_string_handler_conversion*, `T1` is an *applicable_interpolated_string_handler_type*, and `C2` is not an *implicit_string_handler_conversion*, or
> 3. `E` does not exactly match `T2` and at least one of the following holds:
>
>   - `E` exactly matches `T1` (§11.6.4.4)
>   - `T1` is a better conversion target than `T2` (§11.6.4.6)

Syntax

antlr

```
lambda_expression
  : modifier* identifier '=>' (block | expression)
  | attribute_list* modifier* type? lambda_parameters '=>' (block |
expression)
  ;

lambda_parameters
  : lambda_parameter
  | '(' (lambda_parameter (',' lambda_parameter)*)? ')'
  ;

lambda_parameter
  : identifier
  | attribute_list* modifier* type? identifier equals_value_clause?
  ;
```

Open issues

Should default values be supported for lambda expression parameters for completeness?

Should `System.Diagnostics.ConditionalAttribute` be disallowed on lambda expressions since there are few scenarios where a lambda expression could be used conditionally?

C#

```
([Conditional("DEBUG")] static (x, y) => Assert(x == y))(a, b); // ok?
```

Should the *function_type* be available from the compiler API, in addition to the resulting delegate type?

Currently, the inferred delegate type uses `System.Action<>` or `System.Func<>` when parameter and return types are valid type arguments *and* there are no more than 16 parameters, and if the expected `Action<>` or `Func<>` type is missing, an error is reported. Instead, should the compiler use `System.Action<>` or `System.Func<>` regardless of arity? And if the expected type is missing, synthesize a delegate type otherwise?

CallerArgumentExpression

Article • 06/23/2023

> **Note**
>
> This article is a feature specification. The specification represents the *proposed* feature specification. There may be some discrepancies between the feature specification and the completed implementation. Those differences are captured in the pertinent **language design meeting (LDM) notes**. Links to pertinent meetings are included at the bottom of the spec. You can learn more about the process for merging feature speclets into the C# language standard in the article on the **specifications**.

Summary

Allow developers to capture the expressions passed to a method, to enable better error messages in diagnostic/testing APIs and reduce keystrokes.

Motivation

When an assertion or argument validation fails, the developer wants to know as much as possible about where and why it failed. However, today's diagnostic APIs do not fully facilitate this. Consider the following method:

C#

```
T Single<T>(this T[] array)
{
    Debug.Assert(array != null);
    Debug.Assert(array.Length == 1);

    return array[0];
}
```

When one of the asserts fail, only the filename, line number, and method name will be provided in the stack trace. The developer will not be able to tell which assert failed from this information-- they will have to open the file and navigate to the provided line number to see what went wrong.

This is also the reason testing frameworks have to provide a variety of assert methods. With xUnit, `Assert.True` and `Assert.False` are not frequently used because they do not provide enough context about what failed.

While the situation is a bit better for argument validation because the names of invalid arguments are shown to the developer, the developer must pass these names to exceptions manually. If the above example were rewritten to use traditional argument validation instead of `Debug.Assert`, it would look like

C#

```
T Single<T>(this T[] array)
{
    if (array == null)
    {
        throw new ArgumentNullException(nameof(array));
    }

    if (array.Length != 1)
    {
        throw new ArgumentException("Array must contain a single element.",
nameof(array));
    }

    return array[0];
}
```

Notice that `nameof(array)` must be passed to each exception, although it's already clear from context which argument is invalid.

Detailed design

In the above examples, including the string `"array != null"` or `"array.Length == 1"` in the assert message would help the developer determine what failed. Enter `CallerArgumentExpression`: it's an attribute the framework can use to obtain the string associated with a particular method argument. We would add it to `Debug.Assert` like so

C#

```
public static class Debug
{
    public static void Assert(bool condition,
[CallerArgumentExpression("condition")] string message = null);
}
```

The source code in the above example would stay the same. However, the code the compiler actually emits would correspond to

C#

```
T Single<T>(this T[] array)
{
    Debug.Assert(array != null, "array != null");
    Debug.Assert(array.Length == 1, "array.Length == 1");

    return array[0];
}
```

The compiler specially recognizes the attribute on `Debug.Assert`. It passes the string associated with the argument referred to in the attribute's constructor (in this case, `condition`) at the call site. When either assert fails, the developer will be shown the condition that was false and will know which one failed.

For argument validation, the attribute cannot be used directly, but can be made use of through a helper class:

C#

```
public static class Verify
{
    public static void Argument(bool condition, string message,
[CallerArgumentExpression("condition")] string conditionExpression = null)
    {
        if (!condition) throw new ArgumentException(message: message,
paramName: conditionExpression);
    }

    public static void InRange(int argument, int low, int high,
        [CallerArgumentExpression("argument")] string argumentExpression =
null,
        [CallerArgumentExpression("low")] string lowExpression = null,
        [CallerArgumentExpression("high")] string highExpression = null)
    {
        if (argument < low)
        {
            throw new ArgumentOutOfRangeException(paramName:
argumentExpression,
                message: $"{argumentExpression} ({argument}) cannot be less
than {lowExpression} ({low}).");
        }

        if (argument > high)
        {
            throw new ArgumentOutOfRangeException(paramName:
argumentExpression,
```

```
                message: $"{argumentExpression} ({argument}) cannot be
greater than {highExpression} ({high}).");
        }
    }

    public static void NotNull<T>(T argument,
[CallerArgumentExpression("argument")] string argumentExpression = null)
        where T : class
    {
        if (argument == null) throw new ArgumentNullException(paramName:
argumentExpression);
    }
}

T Single<T>(this T[] array)
{
    Verify.NotNull(array); // paramName: "array"
    Verify.Argument(array.Length == 1, "Array must contain a single
element."); // paramName: "array.Length == 1"

    return array[0];
}

T ElementAt(this T[] array, int index)
{
    Verify.NotNull(array); // paramName: "array"
    // paramName: "index"
    // message: "index (-1) cannot be less than 0 (0).", or
    //          "index (6) cannot be greater than array.Length - 1 (5)."
    Verify.InRange(index, 0, array.Length - 1);

    return array[index];
}
```

A proposal to add such a helper class to the framework is underway at https://github.com/dotnet/corefx/issues/17068. If this language feature was implemented, the proposal could be updated to take advantage of this feature.

Extension methods

The `this` parameter in an extension method may be referenced by `CallerArgumentExpression`. For example:

C#

```
public static void ShouldBe<T>(this T @this, T expected,
[CallerArgumentExpression("this")] string thisExpression = null) {}

contestant.Points.ShouldBe(1337); // thisExpression: "contestant.Points"
```

`thisExpression` will receive the expression corresponding to the object before the dot. If it's called with static method syntax, e.g. `Ext.ShouldBe(contestant.Points, 1337)`, it will behave as if first parameter wasn't marked `this`.

There should always be an expression corresponding to the `this` parameter. Even if an instance of a class calls an extension method on itself, e.g. `this.Single()` from inside a collection type, the `this` is mandated by the compiler so `"this"` will get passed. If this rule is changed in the future, we can consider passing `null` or the empty string.

Extra details

- Like the other `Caller*` attributes, such as `CallerMemberName`, this attribute may only be used on parameters with default values.
- Multiple parameters marked with `CallerArgumentExpression` are permitted, as shown above.
- The attribute's namespace will be `System.Runtime.CompilerServices`.
- If `null` or a string that is not a parameter name (e.g. `"notAParameterName"`) is provided, the compiler will pass in an empty string.
- The type of the parameter `CallerArgumentExpressionAttribute` is applied to must have a standard conversion from `string`. This means no user-defined conversions from `string` are allowed, and in practice means the type of such a parameter must be `string`, `object`, or an interface implemented by `string`.

Drawbacks

- People who know how to use decompilers will be able to see some of the source code at call sites for methods marked with this attribute. This may be undesirable/unexpected for closed-source software.

- Although this is not a flaw in the feature itself, a source of concern may be that there exists a `Debug.Assert` API today that only takes a `bool`. Even if the overload taking a message had its second parameter marked with this attribute and made optional, the compiler would still pick the no-message one in overload resolution. Therefore, the no-message overload would have to be removed to take advantage of this feature, which would be a binary (although not source) breaking change.

Alternatives

- If being able to see source code at call sites for methods that use this attribute proves to be a problem, we can make the attribute's effects opt-in. Developers will enable it through an assembly-wide `[assembly: EnableCallerArgumentExpression]` attribute they put in `AssemblyInfo.cs`.
  - In the case the attribute's effects are not enabled, calling methods marked with the attribute would not be an error, to allow existing methods to use the attribute and maintain source compatibility. However, the attribute would be ignored and the method would be called with whatever default value was provided.

C#

```
// Assembly1

void Foo(string bar); // V1
void Foo(string bar, string barExpression = "not provided"); // V2
void Foo(string bar, [CallerArgumentExpression("bar")] string barExpression
= "not provided"); // V3

// Assembly2

Foo(a); // V1: Compiles to Foo(a), V2, V3: Compiles to Foo(a, "not
provided")
Foo(a, "provided"); // V2, V3: Compiles to Foo(a, "provided")

// Assembly3

[assembly: EnableCallerArgumentExpression]

Foo(a); // V1: Compiles to Foo(a), V2: Compiles to Foo(a, "not provided"),
V3: Compiles to Foo(a, "a")
Foo(a, "provided"); // V2, V3: Compiles to Foo(a, "provided")
```

- To prevent the binary compatibility problem from occurring every time we want to add new caller info to `Debug.Assert`, an alternative solution would be to add a `CallerInfo` struct to the framework that contains all the necessary information about the caller.

C#

```
struct CallerInfo
{
    public string MemberName { get; set; }
    public string TypeName { get; set; }
    public string Namespace { get; set; }
    public string FullTypeName { get; set; }
    public string FilePath { get; set; }
    public int LineNumber { get; set; }
```

```
    public int ColumnNumber { get; set; }
    public Type Type { get; set; }
    public MethodBase Method { get; set; }
    public string[] ArgumentExpressions { get; set; }
}

[Flags]
enum CallerInfoOptions
{
    MemberName = 1, TypeName = 2, ...
}

public static class Debug
{
    public static void Assert(bool condition,
        // If a flag is not set here, the corresponding CallerInfo member is
not populated by the caller, so it's
        // pay-for-play friendly.
        [CallerInfo(CallerInfoOptions.FilePath | CallerInfoOptions.Method |
CallerInfoOptions.ArgumentExpressions)] CallerInfo callerInfo =
default(CallerInfo))
    {
        string filePath = callerInfo.FilePath;
        MethodBase method = callerInfo.Method;
        string conditionExpression = callerInfo.ArgumentExpressions[0];
        //...
    }
}

class Bar
{
    void Foo()
    {
        Debug.Assert(false);

        // Translates to:

        var callerInfo = new CallerInfo();
        callerInfo.FilePath = @"C:\Bar.cs";
        callerInfo.Method = MethodBase.GetCurrentMethod();
        callerInfo.ArgumentExpressions = new string[] { "false" };
        Debug.Assert(false, callerInfo);
    }
}
```

This was originally proposed at https://github.com/dotnet/csharplang/issues/87 .

There are a few disadvantages of this approach:

- Despite being pay-for-play friendly by allowing you to specify which properties you need, it could still hurt perf significantly by allocating an array for the expressions/calling `MethodBase.GetCurrentMethod` even when the assert passes.

- Additionally, while passing a new flag to the `CallerInfo` attribute won't be a breaking change, `Debug.Assert` won't be guaranteed to actually receive that new parameter from call sites that compiled against an old version of the method.

Unresolved questions

TBD

Design meetings

N/A

Enhanced #line directives

Article • 01/24/2023

> **Note**
>
> This article is a feature specification. The specification represents the *proposed* feature specification. There may be some discrepancies between the feature specification and the completed implementation. Those differences are captured in the pertinent **language design meeting (LDM) notes**. Links to pertinent meetings are included at the bottom of the spec. You can learn more about the process for merging feature speclets into the C# language standard in the article on the **specifications**.

Summary

The compiler applies the mapping defined by `#line` directives to diagnostic locations and sequence points emitted to the PDB.

Currently only the line number and file path can be mapped while the starting character is inferred from the source code. The proposal is to allow specifying full span mapping.

Motivation

DSLs that generate C# source code (such as ASP.NET Razor) can't currently produce precise source mapping using `#line` directives. This results in degraded debugging experience in some cases as the sequence points emitted to the PDB can't map to the precise location in the original source code.

For example, the following Razor code

```
@page "/"
Time: @DateTime.Now
```

generates code like so (simplified):

```
C#
```

```
#line hidden
void Render()
{
   _builder.Add("Time:");
#line 2 "page.razor"
   _builder.Add(DateTime.Now);
#line hidden
}
```

The above directive would map the sequence point emitted by the compiler for the `_builder.Add(DateTime.Now);` statement to the line 2, but the column would be off (16 instead of 7).

The Razor source generator actually incorrectly generates the following code:

C#

```
#line hidden
void Render()
{
   _builder.Add("Time:");
   _builder.Add(
#line 2 "page.razor"
      DateTime.Now
#line hidden
);
}
```

The intent was to preserve the starting character and it works for diagnostic location mapping. However, this does not work for sequence points since `#line` directive only applies to the sequence points that follow it. There is no sequence point in the middle of the `_builder.Add(DateTime.Now);` statement (sequence points can only be emitted at IL instructions with empty evaluation stack). The `#line 2` directive in above code thus has no effect on the generated PDB and the debugger won't place a breakpoint or stop on the `@DateTime.Now` snippet in the Razor page.

Issues addressed by this proposal: https://github.com/dotnet/roslyn/issues/43432 https://github.com/dotnet/roslyn/issues/46526

Detailed design

We amend the syntax of `line_indicator` used in `pp_line` directive like so:

Current:

```
line_indicator
    : decimal_digit+ whitespace file_name
    | decimal_digit+
    | 'default'
    | 'hidden'
    ;
```

Proposed:

```
line_indicator
    : '(' decimal_digit+ ',' decimal_digit+ ')' '-' '(' decimal_digit+ ','
decimal_digit+ ')' whitespace decimal_digit+ whitespace file_name
    | '(' decimal_digit+ ',' decimal_digit+ ')' '-' '(' decimal_digit+ ','
decimal_digit+ ')' whitespace file_name
    | decimal_digit+ whitespace file_name
    | decimal_digit+
    | 'default'
    | 'hidden'
    ;
```

That is, the `#line` directive would accept either 5 decimal numbers (*start line*, *start character*, *end line*, *end character*, *character offset*), 4 decimal numbers (*start line*, *start character*, *end line*, *end character*), or a single one (*line*).

If *character offset* is not specified its default value is 0, otherwise it specifies the number of UTF-16 characters. The number must be non-negative and less then length of the line following the #line directive in the unmapped file.

(*start line*, *start character*)-(*end line*, *end character*) specifies a span in the mapped file. *start line* and *end line* are positive integers that specify line numbers. *start character*, *end character* are positive integers that specify UTF-16 character numbers. *start line*, *start character*, *end line*, *end character* are 1-based, meaning that the first line of the file and the first UTF-16 character on each line is assigned number 1.

> The implementation would constraint these numbers so that they specify a valid sequence point source span:
>
> - *start line* - 1 is within range [0, 0x20000000) and not equal to 0xfeefee.
> - *end line* - 1 is within range [0, 0x20000000) and not equal to 0xfeefee.
> - *start character* - 1 is within range [0, 0x10000)
> - *end character* - 1 is within range [0, 0x10000)
> - *end line* is greater or equal to *start line*.

> - *start line* is equal to *end line* then *end character* is greater than *start character*.

> Note that the numbers specified in the directive syntax are 1-based numbers but the actual spans in the PDB are zero-based. Hence the -1 adjustments above.

The mapped spans of sequence points and the locations of diagnostics that `#line` directive applies to are calculated as follows.

Let *d* be the zero-based number of the unmapped line containing the `#line` directive. Let span L = (start: (*start line* - 1, *start character* - 1), end: (*end line* - 1, *end character* - 1)) be zero-based span specified by the directive.

Function M that maps a position (line, character) within the scope of the `#line` directive in the source file containing the #line directive to a mapped position (mapped line, mapped character) is defined as follows:

M(*l*, *c*) =

l == *d* + 1 => (*L.start.line* + *l* – *d* – 1, *L.start.character* + max(*c* – *character offset*, 0))

l > *d* + 1 => (*L.start.line* + *l* – *d* – 1, *c*)

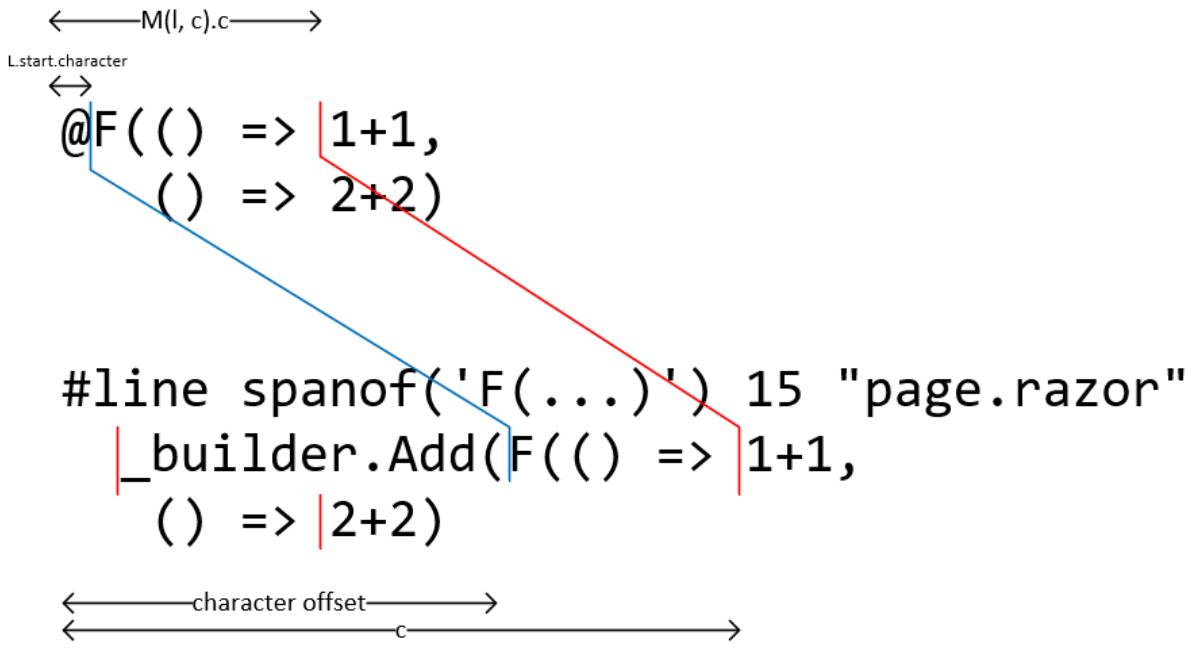
The syntax constructs that sequence points are associated with are determined by the compiler implementation and not covered by this specification. The compiler also decides for each sequence point its unmapped span. This span may partially or fully cover the associated syntax construct.

Once the unmapped spans are determined by the compiler the function *M* defined above is applied to their starting and ending positions, with the exception of the ending position of all sequence points within the scope of the #line directive whose unmapped location is at line *d* + 1 and character less than character offset. The end position of all these sequence points is *L.end*.

> Example [5.i] demonstrates why it is necessary to provide the ability to specify the end position of the first sequence point span.

> The above definition allows the generator of the unmapped source code to avoid intimate knowledge of which exact source constructs of the C# language produce sequence points. The mapped spans of the sequence points in the scope of the `#line` directive are derived from the relative position of the corresponding unmapped spans to the first unmapped span.

> Specifying the *character offset* allows the generator to insert any single-line prefix on the first line. This prefix is generated code that is not present in the mapped file. Such inserted prefix affects the value of the first unmapped sequence point span. Therefore the starting character of subsequent sequence point spans need to be offset by the length of the prefix (*character offset*). See example [2].



Examples

For clarity the examples use `spanof('...')` and `lineof('...')` pseudo-syntax to express the mapped span start position and line number, respectively, of the specified code snippet.

1. First and subsequent spans

Consider the following code with unmapped zero-based line numbers listed on the right:

```
#line (1,10)-(1,15) "a" // 3
  A();B(                // 4
);C();                  // 5
    D();                // 6
```

d = 3 L = (0, 9)..(0, 14)

There are 4 sequence point spans the directive applies to with following unmapped and mapped spans: (4, 2)..(4, 5) => (0, 9)..(0, 14) (4, 6)..(5, 1) => (0, 15)..(1, 1) (5, 2)..(5, 5) => (1, 2)..(1, 5) (6, 4)..(6, 7) => (2, 4)..(2, 7)

2. Character offset

Razor generates `_builder.Add(` prefix of length 15 (including two leading spaces).

Razor:

```
@page "/"
@F(() => 1+1,
   () => 2+2
)
```

Generated C#:

```C#
#line hidden
void Render()
{
#line spanof('F(...)') 15 "page.razor"  // 4
  _builder.Add(F(() => 1+1,            // 5
  () => 2+2                            // 6
));                                    // 7
#line hidden
}
);
}
```

d = 4 L = (1, 1)..(3,0) *character offset* = 15

Spans:

- `_builder.Add(F(…));` => `F(…)`: (5, 2)..(7, 2) => (1, 1)..(3, 0)
- `1+1` => `1+1`: (5, 23)..(5, 25) => (1, 9)..(1, 11)
- `2+2` => `2+2`: (6, 7)..(6, 9) => (2, 7)..(2, 9)

3. Razor: Single-line span

Razor:

```
@page "/"
Time: @DateTime.Now
```

Generated C#:

```C#
#line hidden
void Render()
{
  _builder.Add("Time:");
#line spanof('DateTime.Now') 15 "page.razor"
  _builder.Add(DateTime.Now);
#line hidden
);
}
```

4. Razor: Multi-line span

Razor:

```
@page "/"
@JsonToHtml(@"
{
  ""key1"": "value1",
  ""key2"": "value2"
}")
```

Generated C#:

```C#
#line hidden
void Render()
{
  _builder.Add("Time:");
#line spanof('JsonToHtml(@"...")') 15 "page.razor"
  _builder.Add(JsonToHtml(@"
{
  ""key1"": "value1",
  ""key2"": "value2"
}"));
#line hidden
}
);
}
```

5. Razor: block constructs

i. block containing expressions

In this example, the mapped span of the first sequence point that is associated with the IL instruction that is emitted for the `_builder.Add(Html.Helper(() =>` statement needs to cover the whole expression of `Html.Helper(...)` in the generated file `a.razor`. This is achieved by application of rule [1] to the end position of the sequence point.

```

```

```
@Html.Helper(() =>
{
    <p>Hello World</p>
    @DateTime.Now
})
```

C#

```
#line spanof('Html.Helper(() => { ... })') 13 "a.razor"
_builder.Add(Html.Helper(() =>
#line lineof('{') "a.razor"
{
#line spanof('DateTime.Now') 13 "a.razor"
_builder.Add(DateTime.Now);
#line lineof('}') "a.razor"
}
#line hidden
)
```

ii. block containing statements

Uses existing `#line line file` form since

a) Razor does not add any prefix, b) `{` is not present in the generated file and there can't be a sequence point placed on it, therefore the span of the first unmapped sequence point is unknown to Razor.

The starting character of `Console` in the generated file must be aligned with the Razor file.

```

```

```
@{Console.WriteLine(1);Console.WriteLine(2);}
```

C#

```
#line lineof('@{') "a.razor"
  Console.WriteLine(1);Console.WriteLine(2);
#line hidden
```

iii. block containing top-level code (@code, @functions)

Uses existing `#line line file` form since

a) Razor does not add any prefix, b) `{` is not present in the generated file and there can't be a sequence point placed on it, therefore the span of the first unmapped sequence point is unknown to Razor.

The starting character of `[Parameter]` in the generated file must be aligned with the Razor file.

```
@code {
    [Parameter]
    public int IncrementAmount { get; set; }
}
```

C#

```
#line lineof('[') "a.razor"
    [Parameter]
    public int IncrementAmount { get; set; }
#line hidden
```

6. Razor: `@for`, `@foreach`, `@while`, `@do`, `@if`, `@switch`, `@using`, `@try`, `@lock`

Uses existing `#line line file` form since a) Razor does not add any prefix. b) the span of the first unmapped sequence point may not be known to Razor (or shouldn't need to know).

The starting character of the keyword in the generated file must be aligned with the Razor file.

```
@for (var i = 0; i < 10; i++)
{
}
@if (condition)
{
}
else
{
}
```

C#

```
#line lineof('for') "a.razor"
 for (var i = 0; i < 10; i++)
{
}
#line lineof('if') "a.razor"
 if (condition)
{
}
else
{
}
#line hidden
```

Improved Definite Assignment Analysis

Article • 05/29/2022

> **Note**
>
> This article is a feature specification. The specification represents the *proposed* feature specification. There may be some discrepancies between the feature specification and the completed implementation. Those differences are captured in the pertinent **language design meeting (LDM) notes**. Links to pertinent meetings are included at the bottom of the spec. You can learn more about the process for merging feature speclets into the C# language standard in the article on the **specifications**.

Summary

Definite assignment §9.4 as specified has a few gaps which have caused users inconvenience. In particular, scenarios involving comparison to boolean constants, conditional-access, and null coalescing.

Related discussions and issues

csharplang discussion of this proposal:
https://github.com/dotnet/csharplang/discussions/4240

Probably a dozen or so user reports can be found via this or similar queries (i.e. search for "definite assignment" instead of "CS0165", or search in csharplang).
https://github.com/dotnet/roslyn/issues?q=is%3Aclosed+is%3Aissue+label%3A%22Resolution-By+Design%22+cs0165

I have included related issues in the scenarios below to give a sense of the relative impact of each scenario.

Scenarios

As a point of reference, let's start with a well-known "happy case" that does work in definite assignment and in nullable.

```
C#
```

```
#nullable enable

C c = new C();
if (c != null && c.M(out object obj0))
{
    obj0.ToString(); // ok
}

public class C
{
    public bool M(out object obj)
    {
        obj = new object();
        return true;
    }
}
```

Comparison to bool constant

- https://github.com/dotnet/csharplang/discussions/801
- https://github.com/dotnet/roslyn/issues/45582
  - Links to 4 other issues where people were affected by this.

C#

```
if ((c != null && c.M(out object obj1)) == true)
{
    obj1.ToString(); // undesired error
}

if ((c != null && c.M(out object obj2)) is true)
{
    obj2.ToString(); // undesired error
}
```

Comparison between a conditional access and a constant value

- https://github.com/dotnet/roslyn/issues/33559
- https://github.com/dotnet/csharplang/discussions/4214
- https://github.com/dotnet/csharplang/issues/3659
- https://github.com/dotnet/csharplang/issues/3485
- https://github.com/dotnet/csharplang/issues/3659

This scenario is probably the biggest one. We do support this in nullable but not in definite assignment.

```C#
if (c?.M(out object obj3) == true)
{
    obj3.ToString(); // undesired error
}
```

Conditional access coalesced to a bool constant

- https://github.com/dotnet/csharplang/discussions/916
- https://github.com/dotnet/csharplang/issues/3365

This scenario is very similar to the previous one. This is also supported in nullable but not in definite assignment.

```C#
if (c?.M(out object obj4) ?? false)
{
    obj4.ToString(); // undesired error
}
```

Conditional expressions where one arm is a bool constant

- https://github.com/dotnet/roslyn/issues/4272

It's worth pointing out that we already have special behavior for when the condition expression is constant (i.e. `true ? a : b`). We just unconditionally visit the arm indicated by the constant condition and ignore the other arm.

Also note that we haven't handled this scenario in nullable.

```C#
if (c != null ? c.M(out object obj4) : false)
{
    obj4.ToString(); // undesired error
}
```

Specification

?. (null-conditional operator) expressions

We introduce a new section **?. (null-conditional operator) expressions**. See the null-conditional operator specification (§11.7.7)and Precise rules for determining definite assignment §9.4.4 for context.

As in the definite assignment rules linked above, we refer to a given initially unassigned variable as *v*.

We introduce the concept of "directly contains". An expression *E* is said to "directly contain" a subexpression E_1 if it is not subject to a user-defined conversion §10.5 whose parameter is not of a non-nullable value type, and one of the following conditions holds:

- *E* is E_1. For example, `a?.b()` directly contains the expression `a?.b()`.
- If *E* is a parenthesized expression `(E2)`, and E_2 directly contains E_1.
- If *E* is a null-forgiving operator expression `E2!`, and E_2 directly contains E_1.
- If *E* is a cast expression `(T)E2`, and the cast does not subject E_2 to a non-lifted user-defined conversion whose parameter is not of a non-nullable value type, and E_2 directly contains E_1.

For an expression *E* of the form `primary_expression null_conditional_operations`, let E_0 be the expression obtained by textually removing the leading ? from each of the *null_conditional_operations* of *E* that have one, as in the linked specification above.

In subsequent sections we will refer to E_0 as the *non-conditional counterpart* to the null-conditional expression. Note that some expressions in subsequent sections are subject to additional rules that only apply when one of the operands directly contains a null-conditional expression.

- The definite assignment state of *v* at any point within *E* is the same as the definite assignment state at the corresponding point within *E0*.
- The definite assignment state of *v* after *E* is the same as the definite assignment state of *v* after *primary_expression*.

Remarks

We use the concept of "directly contains" to allow us to skip over relatively simple "wrapper" expressions when analyzing conditional accesses that are compared to other values. For example, `((a?.b(out x))!) == true` is expected to result in the same flow state as `a?.b == true` in general.

We also want to allow analysis to function in the presence of a number of possible conversions on a conditional access. Propagating out "state when not null" is not possible when the conversion is user-defined, though, since we can't count on user-defined conversions to honor the constraint that the output is non-null only if the input is non-null. The only exception to this is when the user-defined conversion's input is a non-nullable value type. For example:

```
C#

public struct S1 { }
public struct S2 { public static implicit operator S2?(S1 s1) => null; }
```

This also includes lifted conversions like the following:

```
C#

string x;

S1? s1 = null;
_ = s1?.M1(x = "a") ?? s1.Value.M2(x = "a");

x.ToString(); // ok

public struct S1
{
    public S1 M1(object obj) => this;
    public S2 M2(object obj) => new S2();
}
public struct S2
{
    public static implicit operator S2(S1 s1) => null;
}
```

When we consider whether a variable is assigned at a given point within a null-conditional expression, we simply assume that any preceding null-conditional operations within the same null-conditional expression succeeded.

For example, given a conditional expression `a?.b(out x)?.c(x)`, the non-conditional counterpart is `a.b(out x).c(x)`. If we want to know the definite assignment state of `x` before `?.c(x)`, for example, then we perform a "hypothetical" analysis of `a.b(out x)` and use the resulting state as an input to `?.c(x)`.

Boolean constant expressions

We introduce a new section "Boolean constant expressions":

For an expression *expr* where *expr* is a constant expression with a bool value:

- The definite assignment state of *v* after *expr* is determined by:
  - If *expr* is a constant expression with value *true*, and the state of *v* before *expr* is "not definitely assigned", then the state of *v* after *expr* is "definitely assigned when false".
  - If *expr* is a constant expression with value *false*, and the state of *v* before *expr* is "not definitely assigned", then the state of *v* after *expr* is "definitely assigned when true".

Remarks

We assume that if an expression has a constant value bool `false`, for example, it's impossible to reach any branch that requires the expression to return `true`. Therefore variables are assumed to be definitely assigned in such branches. This ends up combining nicely with the spec changes for expressions like `??` and `?:` and enabling a lot of useful scenarios.

It's also worth noting that we never expect to be in a conditional state *before* visiting a constant expression. That's why we do not account for scenarios such as "*expr* is a constant expression with value *true*, and the state of *v* before *expr* is "definitely assigned when true".

?? (null-coalescing expressions) augment

We augment section §9.4.4.29 as follows:

For an expression *expr* of the form `expr_first ?? expr_second`:

- ...
- The definite assignment state of *v* after *expr* is determined by:
  - ...
  - If *expr_first* directly contains a null-conditional expression *E*, and *v* is definitely assigned after the non-conditional counterpart E_0, then the definite assignment state of *v* after *expr* is the same as the definite assignment state of *v* after *expr_second*.

Remarks

The above rule formalizes that for an expression like `a?.M(out x) ?? (x = false)`, either the `a?.M(out x)` was fully evaluated and produced a non-null value, in which case `x` was

assigned, or the `x = false` was evaluated, in which case `x` was also assigned. Therefore `x` is always assigned after this expression.

This also handles the `dict?.TryGetValue(key, out var value) ?? false` scenario, by observing that *v* is definitely assigned after `dict.TryGetValue(key, out var value)`, and *v* is "definitely assigned when true" after `false`, and concluding that *v* must be "definitely assigned when true".

The more general formulation also allows us to handle some more unusual scenarios, such as:

- `if (x?.M(out y) ?? (b && z.M(out y))) y.ToString();`
- `if (x?.M(out y) ?? z?.M(out y) ?? false) y.ToString();`

?: (conditional) expressions

We augment section §9.4.4.30 as follows:

For an expression *expr* of the form `expr_cond ? expr_true : expr_false`:

- ...
- The definite assignment state of *v* after *expr* is determined by:
  - ...
  - If the state of *v* after *expr_true* is "definitely assigned when true", and the state of *v* after *expr_false* is "definitely assigned when true", then the state of *v* after *expr* is "definitely assigned when true".
  - If the state of *v* after *expr_true* is "definitely assigned when false", and the state of *v* after *expr_false* is "definitely assigned when false", then the state of *v* after *expr* is "definitely assigned when false".

Remarks

This makes it so when both arms of a conditional expression result in a conditional state, we join the corresponding conditional states and propagate it out instead of unsplitting the state and allowing the final state to be non-conditional. This enables scenarios like the following:

C#

```
bool b = true;
object x = null;
int y;
if (b ? x != null && Set(out y) : x != null && Set(out y))
```

```
{
  y.ToString();
}

bool Set(out int x) { x = 0; return true; }
```

This is an admittedly niche scenario, that compiles without error in the native compiler, but was broken in Roslyn in order to match the specification at the time. See internal issue.

==/!= (relational equality operator) expressions

We introduce a new section ==/!= **(relational equality operator) expressions**.

The general rules for expressions with embedded expressions §9.4.4.23 apply, except for the scenarios described below.

For an expression *expr* of the form `expr_first == expr_second`, where `==` is a[predefined comparison operator (§11.11) or a lifted operator (§11.4.8), the definite assignment state of *v* after *expr* is determined by:

- If *expr_first* directly contains a null-conditional expression *E* and *expr_second* is a constant expression with value *null*, and the state of *v* after the non-conditional counterpart E_0 is "definitely assigned", then the state of *v* after *expr* is "definitely assigned when false".
- If *expr_first* directly contains a null-conditional expression *E* and *expr_second* is an expression of a non-nullable value type, or a constant expression with a non-null value, and the state of *v* after the non-conditional counterpart E_0 is "definitely assigned", then the state of *v* after *expr* is "definitely assigned when true".
- If *expr_first* is of type *boolean*, and *expr_second* is a constant expression with value *true*, then the definite assignment state after *expr* is the same as the definite assignment state after *expr_first*.
- If *expr_first* is of type *boolean*, and *expr_second* is a constant expression with value *false*, then the definite assignment state after *expr* is the same as the definite assignment state of *v* after the logical negation expression `!expr_first`.

For an expression *expr* of the form `expr_first != expr_second`, where `!=` is a predefined comparison operator (§11.11) or a lifted operator ((§11.4.8)), the definite assignment state of *v* after *expr* is determined by:

- If *expr_first* directly contains a null-conditional expression *E* and *expr_second* is a constant expression with value *null*, and the state of *v* after the non-conditional

counterpart E_0 is "definitely assigned", then the state of *v* after *expr* is "definitely assigned when true".
- If *expr_first* directly contains a null-conditional expression *E* and *expr_second* is an expression of a non-nullable value type, or a constant expression with a non-null value, and the state of *v* after the non-conditional counterpart E_0 is "definitely assigned", then the state of *v* after *expr* is "definitely assigned when false".
- If *expr_first* is of type *boolean*, and *expr_second* is a constant expression with value *true*, then the definite assignment state after *expr* is the same as the definite assignment state of *v* after the logical negation expression `!expr_first`.
- If *expr_first* is of type *boolean*, and *expr_second* is a constant expression with value *false*, then the definite assignment state after *expr* is the same as the definite assignment state after *expr_first*.

All of the above rules in this section are commutative, meaning that if a rule applies when evaluated in the form `expr_second op expr_first`, it also applies in the form `expr_first op expr_second`.

Remarks

The general idea expressed by these rules is:

- if a conditional access is compared to `null`, then we know the operations definitely occurred if the result of the comparison is `false`
- if a conditional access is compared to a non-nullable value type or a non-null constant, then we know the operations definitely occurred if the result of the comparison is `true`.
- since we can't trust user-defined operators to provide reliable answers where initialization safety is concerned, the new rules only apply when a predefined `==`/`!=` operator is in use.

We may eventually want to refine these rules to thread through conditional state which is present at the end of a member access or call. Such scenarios don't really happen in definite assignment, but they do happen in nullable in the presence of `[NotNullWhen(true)]` and similar attributes. This would require special handling for `bool` constants in addition to just handling for `null`/non-null constants.

Some consequences of these rules:

- `if (a?.b(out var x) == true)) x() else x();` will error in the 'else' branch
- `if (a?.b(out var x) == 42)) x() else x();` will error in the 'else' branch
- `if (a?.b(out var x) == false)) x() else x();` will error in the 'else' branch

- `if (a?.b(out var x) == null)) x() else x();` will error in the 'then' branch
- `if (a?.b(out var x) != true)) x() else x();` will error in the 'then' branch
- `if (a?.b(out var x) != 42)) x() else x();` will error in the 'then' branch
- `if (a?.b(out var x) != false)) x() else x();` will error in the 'then' branch
- `if (a?.b(out var x) != null)) x() else x();` will error in the 'else' branch

`is` operator and `is` pattern expressions

We introduce a new section **`is` operator and `is` pattern expressions**.

For an expression *expr* of the form `E is T`, where *T* is any type or pattern

- The definite assignment state of *v* before *E* is the same as the definite assignment state of *v* before *expr*.
- The definite assignment state of *v* after *expr* is determined by:
  - If *E* directly contains a null-conditional expression, and the state of *v* after the non-conditional counterpart E_0 is "definitely assigned", and `T` is any type or a pattern that does not match a `null` input, then the state of *v* after *expr* is "definitely assigned when true".
  - If *E* directly contains a null-conditional expression, and the state of *v* after the non-conditional counterpart E_0 is "definitely assigned", and `T` is a pattern that matches a `null` input, then the state of *v* after *expr* is "definitely assigned when false".
  - If *E* is of type boolean and `T` is a pattern which only matches a `true` input, then the definite assignment state of *v* after *expr* is the same as the definite assignment state of *v* after E.
  - If *E* is of type boolean and `T` is a pattern which only matches a `false` input, then the definite assignment state of *v* after *expr* is the same as the definite assignment state of *v* after the logical negation expression `!expr`.
  - Otherwise, if the definite assignment state of *v* after E is "definitely assigned", then the definite assignment state of *v* after *expr* is "definitely assigned".

Remarks

This section is meant to address similar scenarios as in the `==`/`!=` section above. This specification does not address recursive patterns, e.g. `(a?.b(out x), c?.d(out y)) is (object, object)`. Such support may come later if time permits.

Additional scenarios

This specification doesn't currently address scenarios involving pattern switch expressions and switch statements. For example:

```
C#
```

```
_ = c?.M(out object obj4) switch
{
    not null => obj4.ToString() // undesired error
};
```

It seems like support for this could come later if time permits.

There have been several categories of bugs filed for nullable which require we essentially increase the sophistication of pattern analysis. It is likely that any ruling we make which improves definite assignment would also be carried over to nullable.

https://github.com/dotnet/roslyn/issues/49353
https://github.com/dotnet/roslyn/issues/46819
https://github.com/dotnet/roslyn/issues/44127

Drawbacks

It feels odd to have the analysis "reach down" and have special recognition of conditional accesses, when typically flow analysis state is supposed to propagate upward. We are concerned about how a solution like this could intersect painfully with possible future language features that do null checks.

Alternatives

Two alternatives to this proposal:

1. Introduce "state when null" and "state when not null" to the language and compiler. This has been judged to be too much effort for the scenarios we are trying to solve, but that we could potentially implement the above proposal and then move to a "state when null/not null" model later on without breaking people.
2. Do nothing.

Unresolved questions

There are impacts on switch expressions that should be specified:
https://github.com/dotnet/csharplang/discussions/4240#discussioncomment-343395

Design meetings

https://github.com/dotnet/csharplang/discussions/4243

AsyncMethodBuilder override

Article • 09/21/2021

> **Note**
>
> This article is a feature specification. The specification represents the *proposed* feature specification. There may be some discrepancies between the feature specification and the completed implementation. Those differences are captured in the pertinent **language design meeting (LDM) notes**. Links to pertinent meetings are included at the bottom of the spec. You can learn more about the process for merging feature speclets into the C# language standard in the article on the **specifications**.

Summary

Allow per-method override of the async method builder to use. For some async methods we want to customize the invocation of `Builder.Create()` to use a different *builder type*.

C#

```
[AsyncMethodBuilderAttribute(typeof(PoolingAsyncValueTaskMethodBuilder<>))]
// new usage of AsyncMethodBuilderAttribute type
static async ValueTask<int> ExampleAsync() { ... }
```

Motivation

Today, async method builders are tied to a given type used as a return type of an async method. For example, any method that's declared as `async Task` uses `AsyncTaskMethodBuilder`, and any method that's declared as `async ValueTask<T>` uses `AsyncValueTaskMethodBuilder<T>`. This is due to the `[AsyncMethodBuilder(Type)]` attribute on the type used as a return type, e.g. `ValueTask<T>` is attributed as `[AsyncMethodBuilder(typeof(AsyncValueTaskMethodBuilder<>))]`. This addresses the majority common case, but it leaves a few notable holes for advanced scenarios.

In .NET 5, an experimental feature was shipped that provides two modes in which `AsyncValueTaskMethodBuilder` and `AsyncValueTaskMethodBuilder<T>` operate. The on-by-default mode is the same as has been there since the functionality was introduced: when

the state machine needs to be lifted to the heap, an object is allocated to store the state, and the async method returns a `ValueTask{<T>}` backed by a `Task{<T>}`. However, if an environment variable is set, all builders in the process switch to a mode where, instead, the `ValueTask{<T>}` instances are backed by reusable `IValueTaskSource{<T>}` implementations that are pooled. Each async method has its own pool with a fixed maximum number of instances allowed to be pooled, and as long as no more than that number are ever returned to the pool to be pooled at the same time, `async ValueTask<{T}>` methods effectively become free of any GC allocation overhead.

There are several problems with this experimental mode, however, which is both why a) it's off by default and b) we're likely to remove it in a future release unless very compelling new information emerges (https://github.com/dotnet/runtime/issues/13633).

- It introduces a behavioral difference for consumers of the returned `ValueTask{<T>}` if that `ValueTask` isn't being consumed according to spec. When it's backed by a `Task`, you can do with the `ValueTask` things you can do with a `Task`, like await it multiple times, await it concurrently, block waiting for it to complete, etc. But when it's backed by an arbitrary `IValueTaskSource`, such operations are prohibited, and automatically switching from the former to the latter can lead to bugs. With the switch at the process level and affecting all `async ValueTask` methods in the process, whether you control them or not, it's too big a hammer.
- It's not necessarily a performance win, and could represent a regression in some situations. The implementation is trading the cost of pooling (accessing a pool isn't free) with the cost of GC, and in various situations the GC can win. Again, applying the pooling to all `async ValueTask` methods in the process rather than being selective about the ones it would most benefit is too big a hammer.
- It adds to the IL size of a trimmed application, even if the flag isn't set, and then to the resulting asm size. It's possible that can be worked around with improvements to the implementation to teach it that for a given deployment the environment variable will always be false, but as it stands today, every `async ValueTask` method saw for example an ~2K binary footprint increase in aot images due to this option, and, again, that applies to all `async ValueTask` methods in the whole application closure.
- Different methods may benefit from differing levels of control, e.g. the size of the pool employed because of knowledge of the method and how it's used, but the same setting is applied to all uses of the builder. One could imagine working around that by having the builder code use reflection at runtime to look for some attribute, but that adds significant run-time expense, and likely on the startup path.

On top of all of these issues with the existing pooling, it's also the case that developers are prevented from writing their own customized builders for types they don't own. If, for example, a developer wants to implement their own pooling support, they also have to introduce a brand new task-like type, rather than just being able to use `{Value}Task{<T>}`, because the attribute specifying the builder is only specifiable on the type declaration of the return type.

We need a way to have an individual async method opt-in to a specific builder.

Detailed design

Using AsyncMethodBuilderAttribute on methods

In `dotnet/runtime`, add `AttributeTargets.Method` to the targets for `System.Runtime.CompilerServices.AsyncMethodBuilderAttribute`:

C#

```
namespace System.Runtime.CompilerServices
{
    /// <summary>
    /// Indicates the type of the async method builder that should be used
by a language compiler:
    /// - to build the return type of an async method that is attributed,
    /// - to build the attributed type when used as the return type of an
async method.
    /// </summary>
    [AttributeUsage(AttributeTargets.Method | AttributeTargets.Class |
AttributeTargets.Struct | AttributeTargets.Interface |
AttributeTargets.Delegate | AttributeTargets.Enum, Inherited = false,
AllowMultiple = false)]
    public sealed class AsyncMethodBuilderAttribute : Attribute
    {
        /// <summary>Initializes the <see
cref="AsyncMethodBuilderAttribute"/>.</summary>
        /// <param name="builderType">The <see cref="Type"/> of the
associated builder.</param>
        public AsyncMethodBuilderAttribute(Type builderType) => BuilderType
= builderType;

        /// <summary>Gets the <see cref="Type"/> of the associated builder.
</summary>
        public Type BuilderType { get; }
    }
}
```

This allows the attribute to be applied on methods or local functions or lambdas.

Example of usage on a method:

```C#
[AsyncMethodBuilder(typeof(PoolingAsyncValueTaskMethodBuilder<>))] // new usage, referring to some custom builder type
static async ValueTask<int> ExampleAsync() { ... }
```

It is an error to apply the attribute multiple times on a given method.
It is an error to apply the attribute to a lambda with an implicit return type.

A developer who wants to use a specific custom builder for all of their methods can do so by putting the relevant attribute on each method.

Determining the builder type for an async method

When compiling an async method, the builder type is determined by:

1. using the builder type from the `AsyncMethodBuilder` attribute if one is present,
2. otherwise, falling back to the builder type determined by previous approach. (see spec for task-like types).

If an `AsyncMethodBuilder` attribute is present, we take the builder type specified by the attribute and construct it if necessary.
If the override type is an open generic type, take the single type argument of the async method's return type and substitute it into the override type.
If the override type is a bound generic type, then we produce an error.
If the async method's return type does not have a single type argument, then we produce an error.

We verify that the builder type is compatible with the return type of the async method:

1. look for the public `Create` method with no type parameters and no parameters on the constructed builder type.
   It is an error if the method is not found. It is an error if the method returns a type other than the constructed builder type.
2. look for the public `Task` property.
   It is an error if the property is not found.
3. consider the type of that `Task` property (a task-like type):
   It is an error if the task-like type does not matches the return type of the async method.

Note that it is not necessary for the return type of the method to be a task-like type.

Execution

The builder type determined above is used as part of the existing async method design.

For example, today if a method is defined as:

```
C#

public async ValueTask<T> ExampleAsync() { ... }
```

the compiler will generate code akin to:

```
C#

[AsyncStateMachine(typeof(<ExampleAsync>d__29))]
[CompilerGenerated]
static ValueTask<int> ExampleAsync()
{
    <ExampleAsync>d__29 stateMachine;
    stateMachine.<>t__builder = AsyncValueTaskMethodBuilder<int>.Create();
    stateMachine.<>1__state = -1;
    stateMachine.<>t__builder.Start(ref stateMachine);
    return stateMachine.<>t__builder.Task;
}
```

With this change, if the developer wrote:

```
C#

[AsyncMethodBuilder(typeof(PoolingAsyncValueTaskMethodBuilder<>))] // new
usage, referring to some custom builder type
static async ValueTask<int> ExampleAsync() { ... }
```

it would instead be compiled to:

```
C#

[AsyncStateMachine(typeof(<ExampleAsync>d__29))]
[CompilerGenerated]
[AsyncMethodBuilder(typeof(PoolingAsyncValueTaskMethodBuilder<>))] //
retained but not necessary anymore
static ValueTask<int> ExampleAsync()
{
    <ExampleAsync>d__29 stateMachine;
    stateMachine.<>t__builder =
PoolingAsyncValueTaskMethodBuilder<int>.Create(); // <>t__builder now a
different type
    stateMachine.<>1__state = -1;
    stateMachine.<>t__builder.Start(ref stateMachine);
```

```
    return stateMachine.<>t__builder.Task;
}
```

Just those small additions enable:

- Anyone to write their own builder that can be applied to async methods that return `Task<T>` and `ValueTask<T>`
- As "anyone", the runtime to ship the experimental builder support as new public builder types that can be opted into on a method-by-method basis; the existing support would be removed from the existing builders. Methods (including some we care about in the core libraries) can then be attributed on a case-by-case basis to use the pooling support, without impacting any other unattributed methods.

and with minimal surface area changes or feature work in the compiler.

Note that we need the emitted code to allow a different type being returned from `Create` method:

```

AsyncPooledBuilder _builder = AsyncPooledBuilderWithSize4.Create();
```

Note that this mechanism to change the the builder type cannot be used when the synthesized entry-point for top-level statements is async. An explicit entry-point should be used instead.

Drawbacks

- The syntax for applying such an attribute to a method is verbose. The impact of this is lessened if a developer can apply it to multiple methods en mass, e.g. at the type or module level.

Alternatives

- Implement a different task-like type and expose that difference to consumers. `ValueTask` was made extensible via the `IValueTaskSource` interface to avoid that need, however.
- Address just the ValueTask pooling part of the issue by enabling the experiment as the on-by-default-and-only implementation. That doesn't address other aspects, such as configuring the pooling, or enabling someone else to provide their own builder.

- Earlier versions of this document allowed for scoped override of builder types.

Unresolved questions

1. **Replace or also create.** All of the examples in this proposal are about replacing a buildable task-like's builder. Should the feature be scoped to just that? Or should you be able to use this attribute on a method with a return type that doesn't already have a builder (e.g. some common interface)? That could impact overload resolution.
2. **Private Builders**. Should the compiler support non-public async method builders? This is not spec'd today, but experimentally we only support public ones. That makes some sense when the attribute is applied to a type to control what builder is used with that type, since anyone writing an async method with that type as the return type would need access to the builder. However, with this new feature, when that attribute is applied to a method, it only impacts the implementation of that method, and thus could reasonably reference a non-public builder. Likely we will want to support library authors who have non-public ones they want to use.