First Edition — Dart 2.18

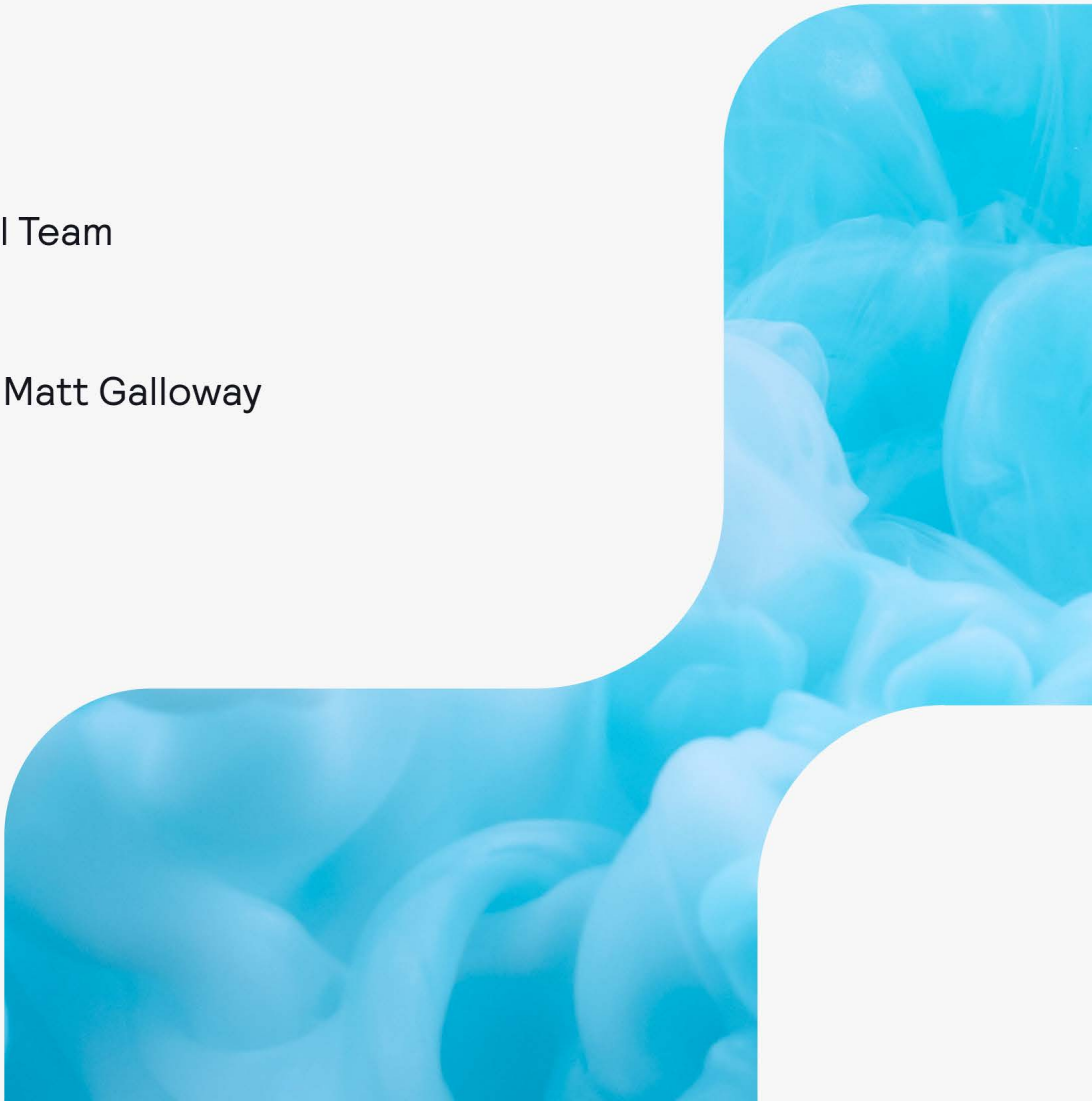
Dart Apprentice: Fundamentals

Modern Cross-Platform Programming With Dart

By the Kodeco Tutorial Team

Jonathan Sande

Based on materials by Matt Galloway



Kodeco

i What You Need

To follow along with this book, you'll need the following:

- **Computer**: Most any computer running a recent version of Windows, macOS or Linux.
- **Dart SDK**: A minimum version of 2.18.0 is required.
- **Visual Studio Code**: This book uses Visual Studio Code for the examples, but you can use another IDE if you prefer.

If you don't have access to a computer with the above requirements, it's also possible to run most of the example code in this book by visiting dartpad.dev in your smartphone's web browser.

ii Book Source Code & Forums

Where to Download the Materials

The materials for this book can be cloned or downloaded from the GitHub book materials repository:

- https://github.com/kodecocodes/daf-materials/tree/editions/1.0

Forums

We've also set up an official forum for the book at https://forums.kodeco.com/c/books/dart-apprentice-fundamentals. This is a great place to ask questions about the book or to submit any errors you may find.

iii Dedications

“To the greatest Coder of them all.”

— *Jonathan Sande*

iv About the Team

About the Authors

Jonathan Sande knows what it's like to bang his head against a wall because his app isn't working. He also understands the all-too-frequent feeling of still being completely lost even with twenty-seven browser tabs open. Once he finally does understand a topic, though, he enjoys writing the explanations and directions he wishes he had had when he started. Online he usually goes by the name Suragch, which is a Mongolian word meaning "student", a reminder to never stop learning. After recently deciding to follow Jesus for real rather than just pretending to, he's still trying to figure out if and how coding fits in.

About the Editors

John Bennedict (JB) Lorenzo is a tech editor of this book. He is a mobile expert currently based in Berlin, but was born in the Philippines, where he began his career in tech. In his free time, he does Latin dancing, calisthenics and traveling. He enjoys experiencing different cultures via food, language, stories and travel.

John Hagemann is an editor of this book. He is a government program and policy analyst, technical writer, and editor, and has worked as a journalist and a writing instructor.

Pablo Mateo is the final pass editor for this book. He is Head of the Onboarding & Mobile Center of Excellence at one of the biggest banks in the world and was also the founder and CTO of a technology development company in Madrid. His expertise is focused on web and mobile app development, although he first started as a creative arts director. He was for many years the main professor of the iOS and Android Mobile Development Masters Degree at a well-known technology school in Madrid (CICE). He has a master's degree in Artificial Intelligence & Machine-Learning and a Certificate in Quantum Computing at MIT.

v Acknowledgments

We would like to thank **Matt Galloway** for his work on *Swift Apprentice*, which is the Kodeco book that several of the early chapters of this book are based on. Once you learn one programming language, you can readily apply many concepts to the next language. That's certainly true for Swift and Dart, and having access to Matt's prior work was a big help in speeding up the writing process for those chapters.

The predecessor of this book was named *Dart Apprentice*, which we later split and expanded into *Dart Apprentice: Fundamentals* and *Dart Apprentice: Beyond the Basics*. The original tech editors for *Dart Apprentice* were **Brian Kayfitz** and **John Bennedict (JB) Lorenzo**. Their comments and suggestions greatly improved the content quality. Some of Brian's recommendations didn't make it into the original *Dart Apprentice* but were influential in designing the structure of the current two-book series. **Chris Belanger** was an editor for the first edition of *Dart Apprentice*, and **Joseph Howard** created a video course that influenced the structure and content of that book.

Finally, we would like to thank **Michael Thomsen** on the Dart Team at Google for reviewing *Dart Apprentice* and giving recommendations for updated content to include in this edition.

vi Introduction

Dart is a modern and powerful programming language. Google intentionally designed it to be unsurprising. In many ways, it's a boring language, and that's a good thing! It means Dart is fast and easy to learn. While Dart does have some unique characteristics, if you have any experience with other object-oriented or C-style languages, you'll immediately feel at home with Dart. Even if you come here as a complete beginner to programming, Dart is a good place to start. The concepts that you'll learn in this book will give you a solid foundation for your coding career.

There's a good chance you picked up this book because you want to make a Flutter app, and you heard you needed to learn Dart. It was no accident that Flutter chose Dart as its language. The Dart virtual machine allows lightning-fast development-time rebuilds, and its ahead-of-time compiler creates native applications for every major platform. As one of the most versatile languages on the market today, you can use Dart to write anything from command-line apps and backend servers to native applications for Android, iOS, web, Mac, Windows, Linux and even embedded devices.

It's no wonder then that developers across the world have taken notice. Rather than completely rewriting the same application in different languages for multiple platforms, developers save countless hours by using a single language and a shared codebase. This translates to a win for companies as well because they save money without sacrificing speed.

So, welcome!

About This Book Series

In its original form, this book started as a single, 10-chapter volume called *Dart Apprentice*. While writing the second edition, we broke the overly-long chapters into more manageable sub-topics, rearranged the teaching order, expanded the explanations and examples, and added completely new chapters. The original 10 chapters grew to almost 30. We didn't want to overwhelm readers with a massive tome but to provide them with a learning path that they could complete in measurable steps. For this reason, we split *Dart Apprentice* into two volumes:

1. ***Dart Apprentice: Fundamentals***, the book you have here, is the first of the two-part series. It covers basic programming concepts like expressions, data types, control flow, loops, functions, classes and collections. When you complete this book, you'll have reached the upper-beginner level.

2. The second book, ***Dart Apprentice: Beyond the Basics***, will build on the concepts you learn here and introduce new topics like string manipulation, anonymous functions, inheritance, interfaces, generics, error handling and asynchronous programming. If you complete that book, you can consider yourself a solid intermediate-level programmer in Dart.

Book Sample Projects

The book comes with supplemental material that's available as an online GitHub repository. In each chapter folder, you'll find a folder called **starter** that contains a starter project with an empty `main` function. You can either open this empty project in your editor by going to **File ▸ Open** in the menu, or just create a new project in the way you'll learn in Chapter 1.

In addition to the starter project, chapters will also have **final** and **challenge** folders. You can refer to the **final** folder if you get lost during the lesson. It'll contain the code from that lesson. Likewise, the **challenge** folder will contain the answers to the exercises and challenges from that chapter. You'll learn the most if you don't copy and paste this code but type it yourself.

Exercises

You'll sometimes find exercises in the middle of a chapter after learning about some topic. These are optional but generally easy to complete. Like the challenges, they'll help you solidify what you're learning.

Challenges

Challenges are an important part of *Dart Apprentice: Fundamentals*. At the end of each chapter, the book will give you one or more tasks to accomplish that make use of the knowledge you learned in the chapter. Completing them will not only help you reinforce that knowledge but will also show that you've mastered it.

How to Read This Book

Each chapter of this book builds on the ones that precede it, so you'll find it easiest to understand if you progress through the chapters in order.

Dart Apprentice: Fundamentals was written with the beginner in mind. If that's you, you'll learn the most by following along and trying each of the code examples, exercises and challenges as you come to them. The way to learn to code is by writing code and experimenting with it. That can't be emphasized enough.

More advanced readers may want to skim the content of this book to get up and running quickly. If that's you, try the challenges at the end of every chapter. If they're easy, move on to the next chapter. If they're not, go back and read the relevant parts of the chapter and check the challenge solutions.

Finally, for all readers, kodeco.com is committed to providing quality, up-to-date learning materials. We'd love to have your feedback. What parts of the book gave you one of those aha learning moments? Was some topic confusing? Did you spot a typo or an error? Let us know at forums.kodeco.com and look for the particular forum category for this book. We'll make an effort to take your comments into account in the next update of the book.

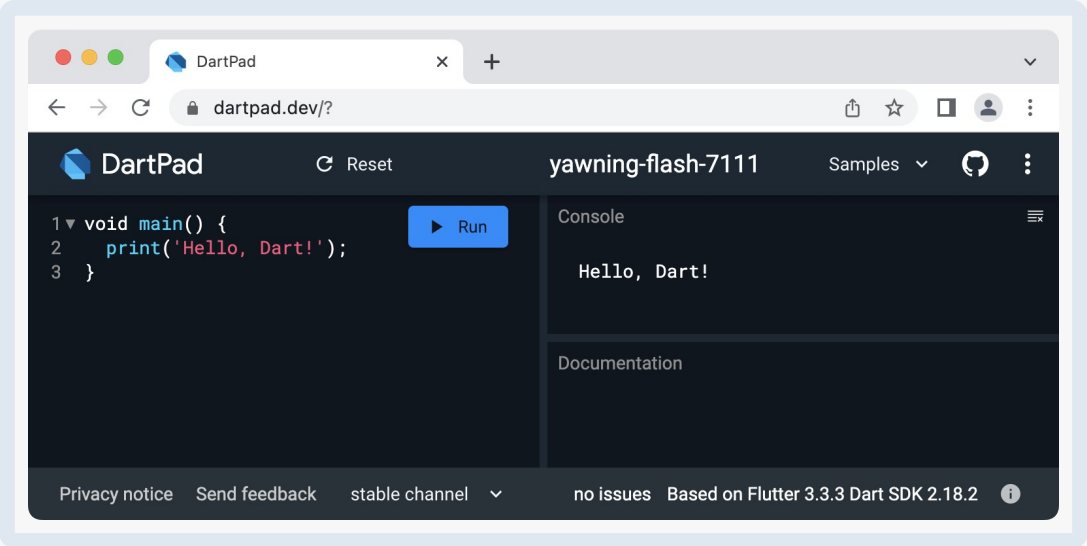
1 Hello, Dart!

Written by Jonathan Sande

This first chapter is designed to help you set up your development environment so that you can get the most out of the following chapters.

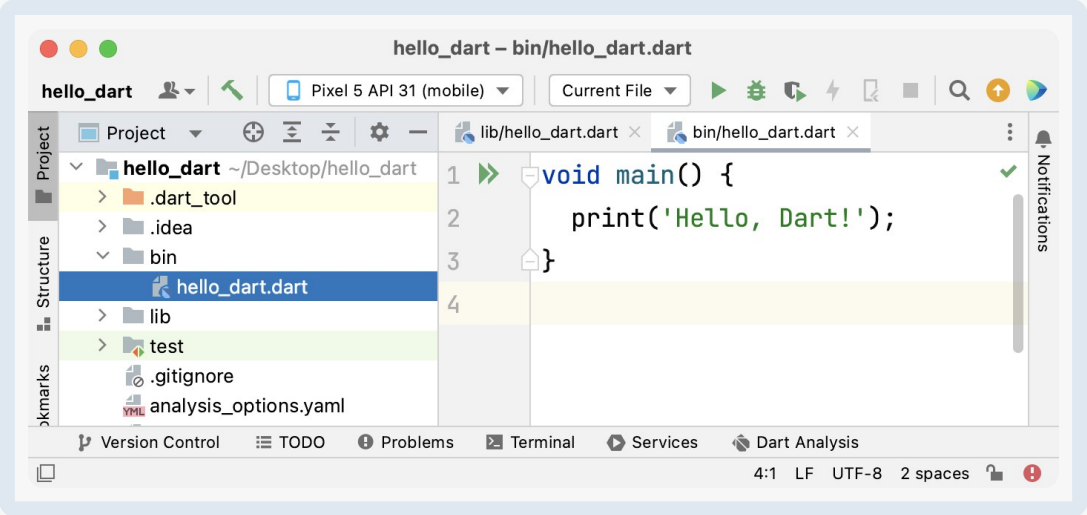
There are several different tools that Dart developers use when building apps:

- **DartPad**: This is a simple browser-based tool for writing and executing Dart code. It's available at dartpad.dev.



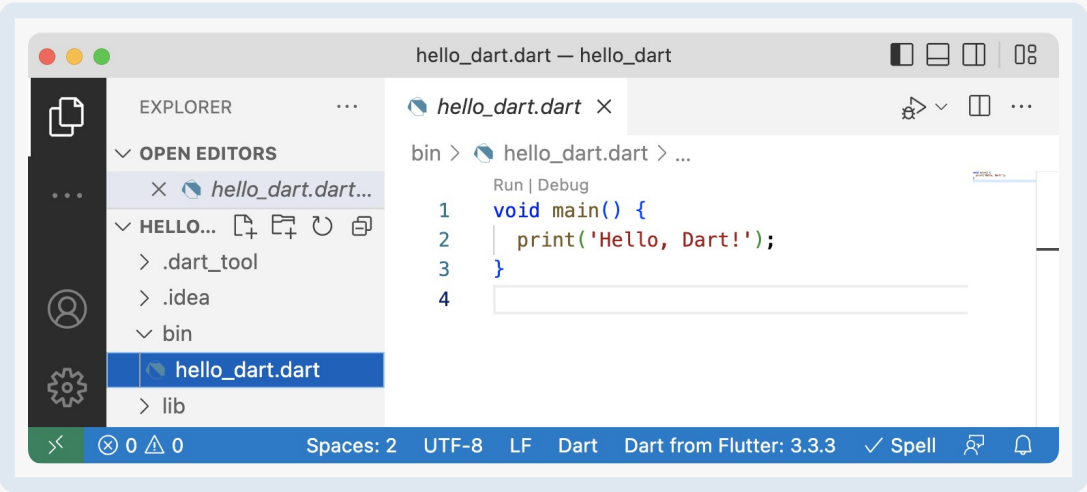
DartPad

- **IntelliJ IDEA**: IntelliJ is a powerful **Integrated Development Environment**, or **IDE**, that supports Dart development through a Dart plugin. Although Android Studio, a popular IDE for Flutter development, is built on IntelliJ, this book recommends that you use plain IntelliJ for pure Dart projects. The IntelliJ Dart plugin makes this an easier task than writing Dart code in Android Studio.



IntelliJ IDEA

- **Visual Studio Code**: Also known as **VS Code**, this is a lightweight IDE with a clean and simple interface. It fully supports Dart development with its Dart extension.



Visual Studio Code

- **Other editors**: There are also Dart plugins from the community for **Eclipse**, **Emacs** and **Vim**. You can even create your own Dart plugin for other editors and IDEs that support the Language Server Protocol (LSP). If you know how to do that, though, you can skip this chapter. Actually, you can probably skip the whole book.

This book uses Visual Studio Code for all of the examples contained within, but if you have another IDE you prefer, then by all means, continue using that one for your Dart development. If you don't have a preference, though, you'll find using VS Code an enjoyable experience. VS Code also supports Flutter development through an extension, so you won't be limiting yourself for future Flutter development if you choose to go the VS Code route now.

Installing Visual Studio Code

Visual Studio Code is a cross-platform, open-source IDE from Microsoft. It runs on Windows, MacOS and Linux, so unless the only device you've got at your disposal is a mobile phone, then you're covered!

> **Note**: If you *do* only have a mobile phone, don't despair! You can run the majority of the code examples in this book on dartpad.dev, which should work fine in any modern mobile browser.

Download Visual Studio Code at code.visualstudio.com, and follow the directions provided on the site to install it.

You'll also need the **Dart SDK**, which you'll install in the next section.

Installing the Dart SDK

The Dart **Software Development Kit**, or **SDK**, is a collection of command-line tools that make it possible to develop Dart applications.

Go to dart.dev/get-dart and follow the directions on that site to download and install the Dart SDK on your platform. If you get an error, try copying the error message and searching for it in Google. Chances are you're not the first person have that problem!

> **Note**: Flutter comes with a copy of the Dart SDK, so if you've already installed a recent version of Flutter, then you're good to go. At the time of this writing, the current stable release of Flutter was 3.3, which includes Dart 2.18.

Verifying the Dart SDK Installation

After you've installed Dart, run the following command in a terminal to ensure that it's working:

```
dart --version
```

You should see the current Dart version displayed.

```
Dart SDK version: 2.18.2 (stable)
```

If yours is less than 2.18, you should upgrade to the latest version. Some examples in this book won't work with older versions of Dart.

For those using the Dart SDK packaged with Flutter, you can upgrade like so:

```
flutter upgrade
```

Contents of the SDK

Now check out what the Dart SDK offers you by entering the following command in the terminal:

```
dart help
```

You'll see a list of tools that make up the SDK. Although you won't directly interact with most of them in this book, it's helpful to know what they do:

- **analyze**: Your IDE uses this tool to tell you when you've made a mistake in your code. The sooner you know, the sooner you can fix it!
- **compile**: This tool compiles Dart code into an optimized native executable program for Windows, Linux or macOS. This is known as ahead-of-time, or **AOT**, compilation. Alongside native executables, web technologies are another major focus for Dart, so you can also use the `compile` tool to convert Dart code to JavaScript.
- **create**: This is for creating new Dart projects, which you'll do yourself in just a minute.
- **devtools**: These are a set of tools to help you with tasks like debugging or profiling the CPU and memory usage of a running app.
- **doc**: If your code has documentation comments, which you'll learn about in the next chapter, this tool will generate the HTML needed to display the comments as a web page.
- **fix**: One of Dart's goals is to continue evolving as a language without becoming bloated by obsolete, or **deprecated**, code. The `fix` tool is there to help developers update their old projects to use the shiniest new Dart syntax.
- **format**: It's easy for the indentation in your code to get messed up. This nice little tool will automatically fix it for you.
- **migrate**: Version 2.12 was a major update to the Dart language with the addition of sound null safety, which you'll learn about in Chapter 11, "Nullability". This tool helps migrate old projects to use null safety. Since you're starting fresh, though, you won't need to migrate anything. Lucky you!
- **pub**: Pub is the name of the **package manager** for Dart, and `pub` is the tool that handles the job. A **package** is a collection of third-party code that you can use in your Dart project. This can save you an incredible amount of time since you don't have to write that code yourself. You can browse the packages available to you on Pub by visiting pub.dev.

- **run**: This runs your Dart program in the Dart **Virtual Machine**, or **VM**. You'll use the Dart VM to compile your code right before it's needed. In contrast to AOT, this is known as just-in-time, or **JIT**, compilation, which will let you make small changes to your code and rerun it almost instantly. This is especially useful for applications like Flutter where you need to make lots of little changes as you refine the UI.
- **test**: Dart fully supports unit testing, and this tool will help you get that done.

Dart on the Command Line

Now that you have the Dart SDK installed, you're going to use the Dart VM to run a few lines of code, first in a single file and then as a full project.

Running a Single Dart File

Find or create a convenient folder on your computer where you can save the Dart projects you create in this book. Create a new text file in that folder and name it **hello.dart**.

Writing the Code

Next, add the following Dart code to that empty file:

```
void main() {
  print('Hello, Dart!');
}
```

`main` is the name of the function where all Dart programs start. Inside that function, you call another function, `print`, which displays the text `Hello, Dart!` on the screen.

Running the Code

Save the file, and then run the following terminal command in the same folder as **hello.dart**:

```
dart run hello.dart
```

The `run` keyword is the `run` tool from the Dart SDK that you learned about earlier. It runs the code in **hello.dart** in the Dart VM.

You should now see the following output in the console:

```
Hello, Dart!
```

Congratulations! You've written and run your first Dart program.

Setting Up a Full Dart Project

It's nice to be able to run a single file, but as you build bigger projects, you'll want to divide your code into manageable pieces and also include configuration and asset files. To do that, you need to create a full Dart project. Remember that `create` tool? The time has come.

Creating the Project

Go to the location where you want to create your project folder, and then run the following command in the terminal:

```
dart create hello_dart
```

This creates a simple Dart project with some default code.

Running the Project

Enter the new folder you just created like so:

```
cd hello_dart
```

Now run the project with the following command:

```
dart run bin/hello_dart.dart
```

You'll see the text `Hello world: 42!`, which is the output of the code in the default project that the `create` tool generated.

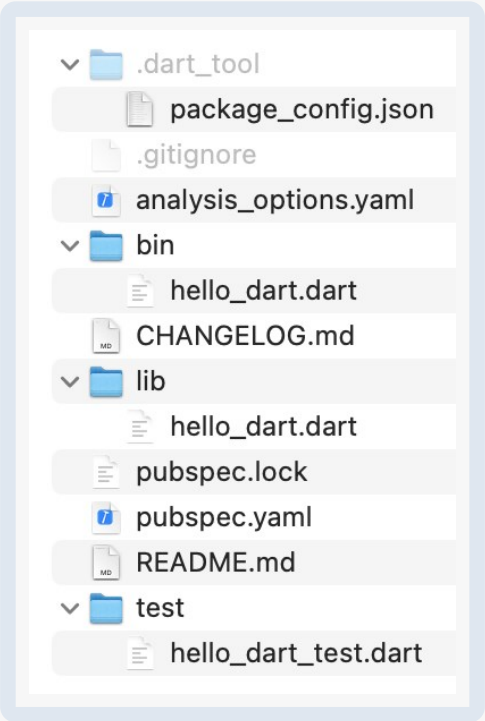
The `run` keyword is optional. Run the project again without it:

```
dart bin/hello_dart.dart
```

Again, `Hello world: 42!` is the result.

The Structure of a Dart Project

Take a look at the structure and contents of the **hello_dart** folder:



The purposes of the most important items in that folder are as follows:

- **.gitignore**: Tells Git which Dart-related files it doesn't need to bother saving to GitHub. Or whatever other Git repository you're using.
- **analysis_options.yaml**: Holds special rules that will help you detect issues with your code, a process known as **linting**.
- **bin**: Contains the executable Dart code.
- **hello_dart.dart**: Named the same as the project folder, the `create` tool generated this file for you to put your Dart code in.
- **CHANGELOG.md**: Holds a manually-curated, Markdown-formatted list of the latest updates to your project. Whenever you release a new version of a Dart project, you should let other developers know what you've changed.
- **lib**: Stands for "library". In larger projects, you'll have many **.dart** files that you'll organize under the **lib** folder. Similar to how a normal library holds a collection of books that you can borrow, a library in the Dart world holds a collection of code that you can use elsewhere.
- **pubspec.yaml**: Contains a list of the third-party Pub dependencies you want to use in your project. The name "pubspec" stands for "Pub specifications". You also set the version number of your project in this file.
- **pubspec.lock**: Specifies the precise version of each dependency that the project is using. This is helpful for teams to ensure that everyone is using the same dependencies.
- **README.md**: Describes what your project does and how to use it. Other developers will appreciate this greatly.
- **test**: Stores your test files. Good developers write code to verify that their programs behave as expected.

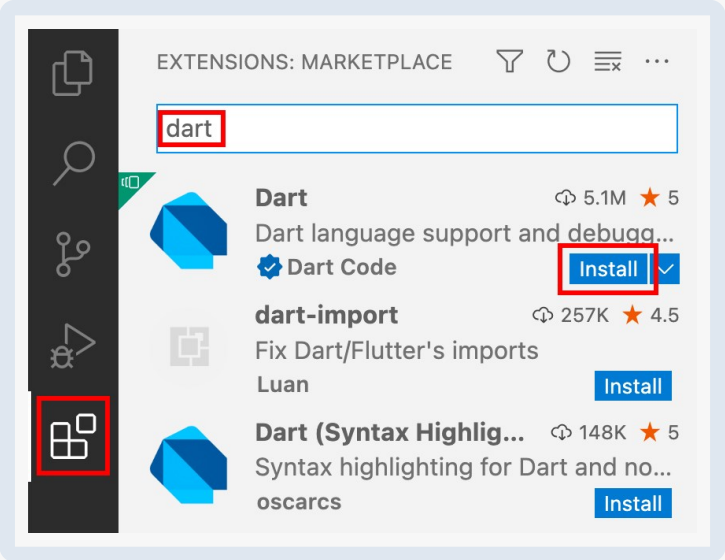
Note: **YAML** stands for "YAML Ain't Markup Language", one of those recursive acronyms that computer programmers like to amuse themselves with. YAML is a clean and readable way to format configuration files, and you'll come across this file type often in your Dart career.

Using VS Code for Dart Development

You've created and run a project from the command line, but it's also possible to do the same thing from within VS Code. This section will walk you through that process.

Installing the Dart Extension

Open Visual Studio Code, and on the left-hand side you'll see a vertical toolbar called the **Activity Bar**. Click the **Extensions icon**, which looks like four boxes. Then type **dart** in the search area. When the Dart extension appears, click the **Install button** to install it.



Now your VS Code installation supports Dart. Next, you'll learn how to create a Dart project in VS Code.

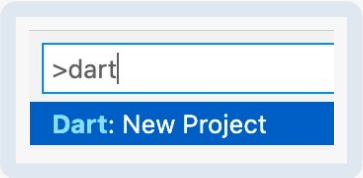
Creating a New Dart Project

The Dart extension in VS Code makes it easy to create a new Dart project. To see how this works, you'll recreate the same project that you previously created from the command line.

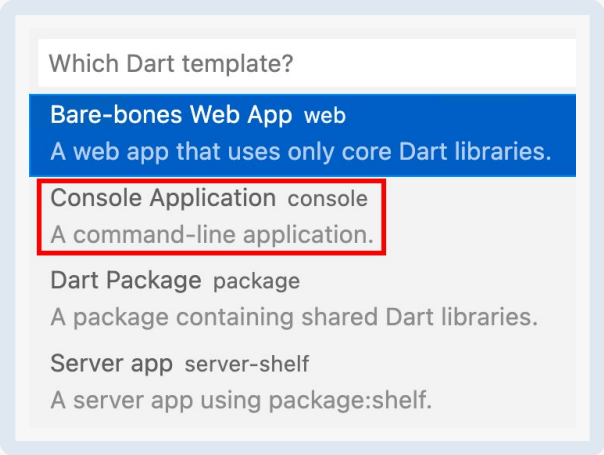
To start, delete your old **hello_dart** folder and its contents.

You can create a new project from the **Command Palette**. To access the Command Palette, either go to **View ▸ Command Palette...** in the menu, or press the shortcut **Command+Shift+P** on a Mac or **Control+Shift+P** on a PC.

Start typing **dart** to filter the list of commands. Then choose **Dart: New Project**.

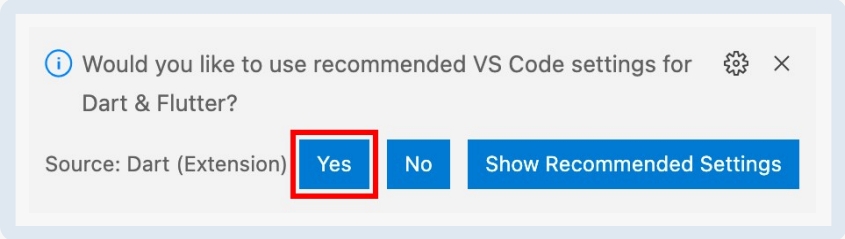


Next, choose **Console Application** from the list.



Choose a location to save the project folder that VS Code will create, and name the project **hello_dart**.

If you see a dialog asking whether to use the recommended VS Code settings for Dart and Flutter, choose **Yes**.



Browsing the Generated Code

Open the file **hello_dart.dart** in the **bin** directory, and you'll see it contains the following code:

```
import 'package:hello_dart/hello_dart.dart' as hello_dart;

void main(List<String> arguments) {
  print('Hello world: ${hello_dart.calculate()}!');
}
```

You don't need to know how all of that works yet, but here are a few notes:

- The line beginning with `import` is grabbing the `calculate` function from the **hello_dart.dart** file in the **lib** folder. You'll learn about importing and functions later in the book.
- The `List<String> arguments` portion is only necessary when creating command-line apps that take arguments. You won't be creating command line apps in this book.

Simplifying the Project

The generated code is great for showing how Dart works, but this book is going to start even simpler. That means you can cut some things out.

Replace the code in **bin/hello_dart.dart** with the following three lines:

```
void main() {
  print('Hello, Dart!');
}
```

Then delete the **lib** and **test** folders. You'll find an option to delete if you right-click the folder names.

These folders serve important functions, but they aren't necessary when you're just getting started.

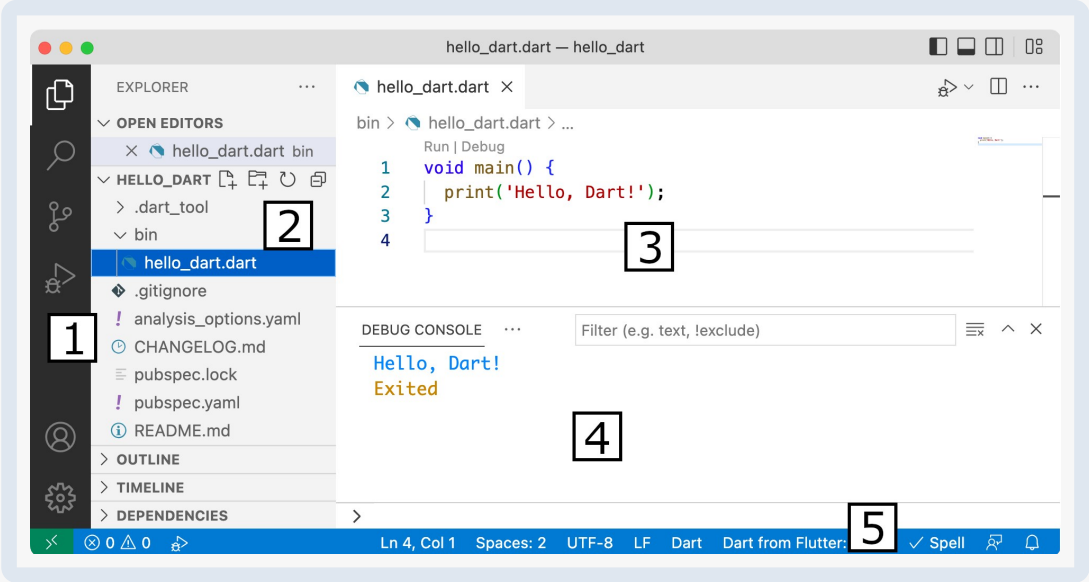
Running Dart in VS Code

To run your code, make sure you have the **bin/hello_dart.dart** file open. Then click the word **Run** that appears directly over the `main` function.

You'll see `Hello, Dart!` appear in the debug console.

Exploring the VS Code UI

This is a good opportunity to explore the various parts of the Visual Studio Code user interface.



The numbers below correspond to the various parts of the IDE:

1. **Activity Bar**: Choose which content to show in the side bar.
2. **Side Bar**: The Explorer is displaying the current project and file.
3. **Editor**: Write your Dart code here.
4. **Panels**: Show program output, run terminal commands and more.
5. **Status Bar**: Display information about the current project.

More Ways to Run Your Project

You ran your project earlier by pressing the **Run** label over the `main` function. Here are three more ways that you can run your project:

1. Choose **Run ▸ Start Debugging** from the menu.
2. Press **F5**.
3. Click the **Start Debugging button** in the top right corner.

All of these do the same thing. This time use **F5** to run the program, and you'll see `Hello, Dart!` appear in the debug console again.

Excellent! You're all set to explore Dart further in the rest of this book.

Key Points

- Visual Studio Code is an Integrated Development Environment (IDE) that you can use to write Dart code when you have the Dart extension installed.
- The Dart SDK provides the underlying tools needed to compile and run Dart apps.
- Dart code run from the command line or in VS Code uses the Dart Virtual Machine.
- The VS Code window is divided into the Activity Bar, Side Bar, Editor, Panel, and Status Bar.
- Pub is the package manager that Dart uses to add third-party source code to your projects.

Where to Go From Here?

If you're new to Visual Studio Code, there's a lot more to learn about it. You can find many instructional resources by going to the **Help** menu.

Most of the time, you'll write Dart code using VS Code on a computer, but if you find yourself waiting in a long line at the supermarket, you can pass the time by writing Dart code at dartpad.dev, which will run on your mobile browser. Try out the sample Dart and Flutter projects there. If your device screen is too small to edit the code easily, go to your browser settings and view the page as a desktop site.

Now that you have your programming environment all set up, you'll go on to start writing real Dart code in the next chapter. See you there!

2 Expressions, Variables & Constants

Written by Jonathan Sande

Now that you've set up your Dart development environment, it's time to start writing some Dart code!

Follow along with the coding examples in this chapter. Simply create a project for your own use, and type in the code as you work through the examples.

Commenting Code

Dart, like most other programming languages, allows you to document your code through the use of comments. These allow you to write text directly alongside your code, and the compiler ignores it.

The depth and complexity of the code you write can obscure the big-picture details of why you wrote your code a certain way, or even what problem your code is solving. To prevent this, it's a good idea to document what you wrote so that the next human who passes through your code will be able to make sense of your work. That next human, after all, may be a future you!

The first way to write a comment in Dart is by starting a line with two forward slashes:

```
// This is a comment. It is not executed.
```

This is a **single-line comment**.

You can stack up single-line comments to allow you to write **multi-line comments** as shown below:

```
// This is also a comment,
// over multiple lines.
```

You may also create **comment blocks** by putting your comment text between `/*` and `*/`:

```
/* This is also a comment. Over many...
many...
many lines. */
```

The start is denoted by `/*` and the end is denoted by `*/`.

Dart also allows you to nest comments, like so:

```
/* This is a comment.
/* And inside it is
another comment. */
Back to the first. */
```

In addition to these two ways of writing comments, Dart also has a third type called **documentation comments**. Single-line documentation comments begin with `///`, while block documentation comments are enclosed between `/**` and `*/`. Here's an example of each:

```
/// I am a documentation comment
/// at your service.

/**
 * Me, too!
 */
```

Documentation comments are super useful because you can use them to generate...you guessed it...documentation! You'll want to add documentation comments to your code whenever you have a public API so that the users, and your future self, will know how your API works. Although this book won't go into depth on this, documentation comments even support **Markdown** formatting so that you can add elements like code examples or links to your comments.

> **Note**: An **API**, or **application programming interface**, is code that you share with other people or programs. You'll learn how to make your own API in *Dart Apprentice: Beyond the Basics*.

The Flutter and Dart documentation is well-known for its detailed comments. And since the code for Flutter and Dart is open source, simply browsing through it is an excellent way to learn how great documentation comments are written.

In fact, you can try browsing that documentation right now. Take the Hello Dart program from Chapter 1, "Hello, Dart!":

```
void main() {
  print('Hello, Dart!');
}
```

In VS Code, **Command+Click** on a Mac, or **Control+Click** on a PC, the `print` keyword. VS Code will take you to the source code for that keyword and you'll see the documentation comments for `print`:

```
/// Prints a string representation of the object to the console.
void print(Object? object) {
```

Speaking of `print`, that's another useful tool when writing Dart code.

Printing Output

`print` will output whatever you want to the debug console.

For example, consider the following code:

```
print('Hello, Dart Apprentice reader!');
```

Run this code and it'll output a nice message to the debug console, like so:

Adding `print` statements into your code is an easy way to monitor what's happening at a particular point in your code. Later, when you're ready to take your debugging to the next level, you can check out some of the more detailed logging packages on pub.dev.

You can print any expression in Dart. To learn what an expression is, though, keep reading.

Statements and Expressions

Two important words that you'll often hear thrown about in programming language documentation are **statement** and **expression**. It's helpful to understand the difference between the two.

Statements

A **statement** is a command, something you tell the computer to do. In Dart, all simple statements end with a semicolon. You've already seen that with the `print` statement:

```
print('Hello, Dart Apprentice reader!');
```

The semicolon on the right finishes the statement.

People coming from languages that don't require semicolons may think they're unnecessary. However, due to Dart's special syntax and features, semicolons give the compiler the context it needs to properly understand the code.

In addition to simple statements, Dart also has **complex statements** and code blocks that use curly braces, but there's no need to add semicolons after the braces.

One example of a complex statement is the `if` statement:

```
if (someCondition) {
  // code block
}
```

No semicolons are needed on the lines with the opening or closing curly braces. You'll learn more about `if` statements and other control flow statements in Chapter 5, "Control Flow".

Expressions

Unlike a statement, an **expression** doesn't *do* something; it *is* something. That is, an expression is a value, or is something that can be calculated as a value.

Here are a few examples of expressions in Dart:

```
42
3 + 2
'Hello, Dart Apprentice reader!'
x
```

The values can be numbers, text, or some other type. They can even be variables such as `x`, whose value isn't known until runtime.

Coming up next, you'll see many more examples of expressions.

Arithmetic Operations

In this section, you'll learn about the various **arithmetic operations** that Dart has to offer by seeing how they apply to numbers. In later chapters, you'll learn about operations for types beyond simple numbers.

Simple Operations

Each operation in Dart uses a symbol known as the **operator** to denote the type of operation it performs. Consider the four arithmetic operations you learned in your early school days: addition, subtraction, multiplication and division. For these simple operations, Dart uses the following operators:

- Add: `+`
- Subtract: `-`
- Multiply: `*`
- Divide: `/`

These operators are used like so:

```
2 + 6
10 - 2
2 * 4
24 / 3
```

Each of these lines is an expression because each can be calculated down to a value. In these cases, all four expressions have the same value: `8`. Notice how the code looks similar to how you would write the operations out with pen and paper.

Check the answers yourself in VS Code using a `print` statement:

```
print(2 + 6);
```

Note: If you print the result the expression `24 / 3` in VS Code, you'll see `8.0` rather than `8`. This is because the output of the division operator is a `double`. You'll learn more about `double` later in this chapter and in Chapter 3, "Types & Operations".

On the other hand, if you print `24 / 3` on dartpad.dev, you'll only see `8`. That's because on the web, Dart compiles your code to JavaScript, and JavaScript doesn't differentiate between `8` and `8.0`.

Dart ignores whitespace, so you can remove the spaces surrounding the operator:

```
2+6
```

However, it's often easier to read expressions when you have white space on either side. In fact, the **dart format** tool in the Dart SDK will format your code according to the standard whitespace formatting rules. In VS Code, you can apply `dart format` with the keyboard shortcut **Shift+Option+F** on a Mac or **Shift+Alt+F** on a PC. Depending on your VS Code settings, saving the file may also trigger an auto-format.

> **Note**: This book won't always explicitly tell you to `print(X)`, where `X` is some Dart expression that you're learning about, but you should proactively do this yourself to check the value.

Decimal Numbers

All of the operations above use whole numbers, more formally known as **integers**. However, as you know, not every number is whole.

For example, consider the following:

```
22 / 7
```

If you're used to another language that uses integer division by default, you might expect the result to be `3`. However, Dart gives you the standard decimal answer:

```
3.142857142857143
```

If you actually did want to perform integer division, then you could use the `~/` operator:

```
22 ~/ 7
```

This produces a result of `3`.

The `~/` operator is officially called the **truncating division operator** when applied to numbers. If you squint, the tilde kind of looks like an elephant trunk, so that might help you remember what it does. Or not.

The Euclidean Modulo Operation

Dart also has more complex operations you can use. All of them are standard mathematical operations, just not as common as the others. You'll take a look at them now.

The first of these is the **Euclidean modulo operation**. That's a complex name for an easy task. In division, the denominator goes into the numerator a whole number of times, plus a remainder. This remainder is what the Euclidean modulo operation calculates. For example, 10 modulo 3 equals 1, because 3 goes into 10 three times, with a remainder of 1.

In Dart, the Euclidean modulo operator is the `%` symbol. You use it like so:

```
28 % 10
```

In this case, the result is 8, because 10 goes into 28 twice with a remainder of 8.

Order of Operations

Of course, it's likely that when you calculate a value, you'll want to use multiple operators. Here's an example of how to do this in Dart:

```
((8000 / (5 * 10)) - 32) ~/ (29 % 5)
```

Note the use of parentheses, which in Dart serve two purposes: to make it clear to anyone reading the code — including yourself — what you meant, and to disambiguate the intended order of operations.

For example, consider the following:

```
350 / 5 + 2
```

Does this equal `72` (350 divided by 5, plus 2) or `50` (350 divided by 7)? Those of you who paid attention in school are shouting, "72"! And you're right.

Dart uses the same reasoning and achieves this through what's known as **operator precedence**. The division operator (`/`) has higher precedence than the addition operator (`+`), so in this example, the code executes the division operation first.

If you wanted Dart to perform the addition first — that is, so the expression will return `50` instead of `72` — then you could use parentheses, like so:

```
350 / (5 + 2)
```

The precedence rules are the same as you learned in school. Multiplication and division have equal precedence. Addition and subtraction are equal in precedence to each other, but are lower in precedence than multiplication and division.

The `~/` and `%` operators have the same precedence as multiplication and division. If you're ever uncertain about what precedence an operator has, you can always use parentheses to be sure the expression evaluates as you want it to.

Math Functions

Dart also has a large range of math functions. You never know when you'll need to flex those trigonometry muscles, especially when you're a pro at Dart and writing complex animations!

To use these math functions, add the following import to the top of your file:

```
import 'dart:math';
```

`dart:math` is one of Dart's core libraries. Adding the `import` statement tells the compiler that you want to use something from that library.

Now you can write the following:

```
sin(45 * pi / 180)
// 0.7071067811865475

cos(135 * pi / 180)
// -0.7071067811865475
```

These convert an angle from degrees to radians, and then compute the sine and cosine respectively. Notice how both make use of `pi`, which is a constant Dart provides you. Nice!

> **Note**: Remember that if you want to see the values of these mathematical expressions, you need to put them inside a `print` statement like this:
>
> `print(sin(45 * pi / 180));`

Then there's this:

```
sqrt(2)
// 1.4142135623730951
```

This computes the square root of `2`.

Not mentioning these would be a shame:

```
max(5, 10)
// 10

min(-5, -10)
// -10
```

These compute the maximum and minimum of two numbers respectively.

If you're particularly adventurous you can even combine these functions like so:

```
max(sqrt(2), pi / 2)
// 1.5707963267948966
```

Exercise

Print the value of `1` over the square root of `2`. Confirm that it equals the sine of `45°`.

This is your first exercise. You can find the answers in the **challenge** folder in the supplemental materials that come with this book.

Naming Data

At its simplest, computer programming is all about manipulating data because everything you see on your screen can be reduced to numbers. Sometimes you represent and work with data as various types of numbers, but other times, the data comes in more complex forms such as text, images and collections.

In your Dart code, you can give each piece of data a name you can use to refer to that piece of data later. The name carries with it an associated **type** that denotes what sort of data the name refers to, such as text, numbers, or a date. You'll learn about some of the basic types in this chapter, and you'll encounter many other types throughout this book.

Variables

Take a look at the following:

```
int number = 10;
```

This statement declares a **variable** called `number` of type `int`. It then sets the value of the variable to the number `10`. The `int` part of the statement is known as a **type annotation**, which tells Dart explicitly what the type is.

> **Note**: Thinking back to operators, here's another one. The equals sign, `=`, is known as the **assignment operator** because it assigns a value to a variable. This is different than the equals sign you are familiar with from math. That equals sign is more like the `==` **equality operator** in Dart, which you'll learn about in Chapter 5, "Control Flow".

A variable is called a variable because its value can change. If you want to change the value of a variable, then you can just give it a different value of the same type:

```
int number = 10;
number = 15;
```

`number` started as `10` but then changed to `15`.

The type `int` can store integers. The way you store decimal numbers is like so:

```
double apple = 3.14159;
```

This is similar to the `int` example. This time, though, the variable is a `double`, a type that can store decimals with high precision.

For readers who are familiar with object-oriented programming, you'll be interested to learn that `10`, `3.14159` and every other value that you can assign to a variable are objects in Dart. In fact, Dart doesn't have the primitive variable types that exist in some languages. Everything is an object. Although `int` and `double` look like primitives, they're subclasses of `num`, which is a subclass of `Object`.

With numbers as objects, this lets you do some interesting things:

```
10.isEven
// true

3.14159.round()
// 3
```

> **Note**: Don't worry if you're not familiar with object-oriented programming. You'll learn all about it as you progress through this book series.

Type Safety

Dart is a type-safe language. That means once you tell Dart what a variable's type is, you can't change it later. Here's an example:

```
int myInteger = 10;
myInteger = 3.14159; // No, no, no. That's not allowed :]
```

`3.14159` is a `double`, but you already defined `myInteger` as an `int`. No changes allowed!

Dart's type safety will save you countless hours when coding since the compiler will tell you immediately whenever you try to give a variable the wrong type. This prevents you from having to chase down bugs after you experience runtime crashes.

Of course, sometimes it's useful to be able to assign related types to the same variable. That's still possible. For example, you could solve the problem above, where you want `myNumber` to store both an integer and floating-point value, like so:

```
num myNumber;
myNumber = 10;       // OK
myNumber = 3.14159; // OK
myNumber = 'ten';    // No, no, no.
```

The `num` type can be either an `int` or a `double`, so the first two assignments work. However, the string value `'ten'` is of a different type, so the compiler complains.

Now, if you like living dangerously, you can throw safety to the wind and use the `dynamic` type. This lets you assign any data type you like to your variable, and the compiler won't warn you about *anything*.

```
dynamic myVariable;
myVariable = 10;       // OK
myVariable = 3.14159; // OK
myVariable = 'ten';    // OK
```

But, honestly, don't do that. Friends don't let friends use `dynamic`. Your life is too valuable for that.

In Chapter 3, "Types & Operations", you'll learn more about types.

Type Inference

Dart is smart in a lot of ways. You don't even have to tell it the type of a variable, and Dart can still figure out what you mean. The `var` keyword says to Dart, "Use whatever type is appropriate."

```
var someNumber = 10;
```

There's no need to tell Dart that `10` is an integer. Dart *infers* the type and makes `someNumber` an `int`. Type safety still applies, though:

```
var someNumber = 10;
someNumber = 15;      // OK
someNumber = 3.14159; // No, no, no.
```

Trying to set `someNumber` to a `double` will result in an error. Your program won't even compile. Quick catch; time saved. Thanks, Dart!

Constants

Dart has two different types of "variables" whose values never change. They are declared with the `const` and `final` keywords, and the following sections will show the difference between the two.

Const Constants

Variables whose value you can change are known as **mutable data**. Mutables certainly have their place in programs, but can also present problems. It's easy to lose track of all the places in your code that can change the value of a particular variable. For that reason, you should use **constants** rather than variables whenever possible. These unchangeable variables are known as **immutable data**.

To create a constant in Dart, use the `const` keyword:

```
const myConstant = 10;
```

Just as with `var`, Dart uses type inference to determine that `myConstant` is an `int`.

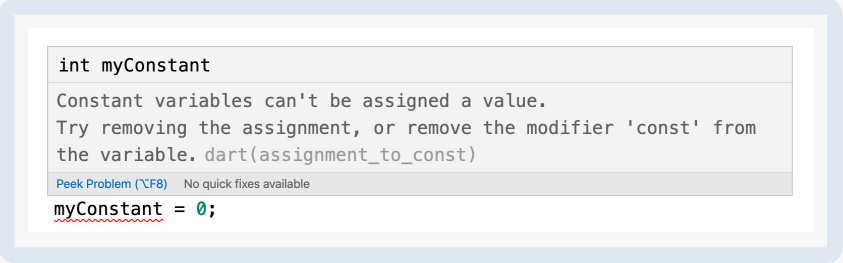
Once you've declared a constant, you can't change its data. For example, consider the following example:

```
const myConstant = 10;
myConstant = 0; // Not allowed.
```

The previous code produces an error:

```
Constant variables can't be assigned a value.
```

In VS Code, you would see the error represented this way:



If you think "constant variable" sounds a little oxymoronic, just remember that it's in good company: virtual reality, advanced BASIC, readable Perl and internet security.

Final Constants

Often, you know you'll want a constant in your program, but you don't know what its value is at compile time. You'll only know the value after the program starts running. This kind of constant is known as a **runtime constant**.

In Dart, `const` is only used for **compile-time constants**; that is, for values that can be determined by the compiler before the program ever starts running.

If you can't create a `const` variable because you don't know its value at compile time, then you must use the `final` keyword to make it a runtime constant. There are many reasons you might not know a value until after your program is running. For example, you might need to fetch a value from the server, or query the device settings, or ask a user to input their age.

Here is another example of a runtime value:

```
final hoursSinceMidnight = DateTime.now().hour;
```

`DateTime.now()` is a Dart function that tells you the current date and time when the code is run. Adding `hour` to that tells you the number of hours that have passed since the beginning of the day. Since that code will produce different results depending on the time of day, this is most definitely a runtime value. So to make `hoursSinceMidnight` a constant, you must use the `final` keyword instead of `const`.

If you try to change the `final` constant after it's already been set:

```
hoursSinceMidnight = 0;
```

this will produce the following error:

```
The final variable 'hoursSinceMidnight' can only be set once.
```

You don't actually need to worry too much about the difference between `const` and `final` constants. As a general rule, try `const` first, and if the compiler complains, then make it `final`. The benefit of using `const` is it gives the compiler the freedom to make internal optimizations to the code before compiling it.

No matter what kind of variable you have, though, you should give special consideration to what you *call* it.

Using Meaningful Names

Always try to choose meaningful names for your variables and constants. Good names act as documentation and make your code easy to read.

A good name specifically describes the role of a variable or constant. Here are some examples of good names:

- `personAge`
- `numberOfPeople`
- `gradePointAverage`

Often a bad name is simply not descriptive enough. Here are some examples of bad names:

- `a`
- `temp`
- `average`

The key is to ensure that you'll understand what the variable or constant refers to when you read it again later. Don't make the mistake of thinking you have an infallible memory! It's common in computer programming to look back at your own code as early as a day or two later and have forgotten what it does. Make it easier for yourself by giving your variables and constants intuitive, precise names.

Also, note how the names above are written. In Dart, it's standard to use `lowerCamelCase` for variable and constant names. Follow these rules to properly case your names:

- Start with a lowercase letter.
- If the name is made up of multiple words, join them together and start every word after the first one with an uppercase letter.
- Treat abbreviations like words, for example, `sourceUrl` and `urlDescription`.

Exercises

If you haven't been following along with these exercises in VS Code, now's the time to create a new project and try some exercises to test yourself!

1. Declare a constant of type `int` called `myAge` and set it to your age.
2. Declare a variable of type `double` called `averageAge`. Initially, set the variable to your own age. Then, set it to the average of your age and your best friend's age.
3. Create a constant called `testNumber` and initialize it with whatever integer you'd like. Next, create another constant called `evenOdd` and set it equal to `testNumber` modulo `2`. Now change `testNumber` to various numbers. What do you notice about `evenOdd`?

Increment and Decrement

A common operation that you'll need is to be able to **increment** or **decrement** a variable. In Dart, this is achieved like so:

```
var counter = 0;

counter += 1;
// counter = 1;

counter -= 1;
// counter = 0;
```

The counter variable begins as `0`. The increment sets its value to `1`, and then the decrement sets its value back to `0`.

The `+=` and `-=` operators are similar to the assignment operator (`=`), except they also perform an addition or subtraction. They take the current value of the variable, add or subtract the given value, and assign the result back to the variable.

In other words, the code above is shorthand for the following:

```
var counter = 0;
counter = counter + 1;
counter = counter - 1;
```

If you only need to increment or decrement by `1`, then you can use the `++` or `--` operators:

```
var counter = 0;
counter++; // 1
counter--; // 0
```

The `*=` and `/=` operators perform similar operations for multiplication and division, respectively:

```
double myValue = 10;

myValue *= 3;  // same as myValue = myValue * 3;
// myValue = 30.0;

myValue /= 2;  // same as myValue = myValue / 2;
// myValue = 15.0;
```

Division in Dart produces a result with a type of `double`, so `myValue` could not be an `int`.

Challenges

Before moving on, here are some challenges to test your knowledge of variables and constants. It's best if you try to solve them yourself, but solutions are available with the supplementary materials for this book if you get stuck.

Challenge 1: Variable Dogs

Declare an `int` variable called `dogs` and set that equal to the number of dogs you own. Then imagine you bought a new puppy and increment the `dogs` variable by one.

Challenge 2: Make It Compile

Given the following code:

```
age = 16;
print(age);
age = 30;
print(age);
```

Modify the first line so that the code compiles. Did you use `var`, `int`, `final` or `const`?

Challenge 3: Compute the Answer

Consider the following code:

```
const x = 46;
const y = 10;
```

Work out what each answer equals when you add the following lines of code to the code above:

```
const answer1 = (x * 100) + y;
const answer2 = (x * 100) + (y * 100);
const answer3 = (x * 100) + (y / 10);
```

Challenge 4: Average Rating

Declare three constants called `rating1`, `rating2` and `rating3` of type `double` and assign each a value. Calculate the average of the three and store the result in a constant named `averageRating`.

Challenge 5: Quadratic Equations

A quadratic equation is something of the form

$a \cdot x^2 + b \cdot x + c = 0$.

The values of `x` which satisfy this can be solved by using the equation

$x = (-b \pm \sqrt{b^2 - 4 \cdot a \cdot c}) / (2 \cdot a)$.

Declare three constants named `a`, `b` and `c` of type `double`. Then calculate the two values for `x` using the equation above (noting that the `±` means plus or minus, so one value of `x` for each). Store the results in constants called `root1` and `root2` of type `double`.

Key Points

- Code comments are denoted by a line starting with `//`, or by multiple lines bookended with `/*` and `*/`.
- Documentation comments are denoted by a line starting with `///` or multiple lines bookended with `/**` and `*/`.
- You can use `print` to write to the debug console.
- The arithmetic operators are:

```
Addition: +
Subtraction: -
Multiplication: *
Division: /
Integer division: ~/
Modulo (remainder): %
```

- Dart has many functions, including `min`, `max`, `sqrt`, `sin` and `cos`.
- Constants and variables give names to data.
- Once you've declared a constant, you can't change its data, but you can change a variable's data at any time.
- If a variable's type can be inferred, you can replace the type with the `var` keyword.
- The `const` keyword is used for compile-time constants while `final` is used for runtime constants.
- Always give variables and constants meaningful names to save yourself and your colleagues headaches later.
- The following operators perform arithmetic and then assign back to the variable:

```
Add and assign: +=
Subtract and assign: -=
Multiply and assign: *=
Divide and assign: /=
Increment by 1: ++
Decrement by 1: --
```

Where to Go From Here?

In this chapter, you learned that documentation comments support Markdown formatting. If you aren't familiar with Markdown, it would be well worth your time to learn it. Dart supports CommonMark, so commonmark.org is a good place to start.

Speaking of documentation, it's also worth your while to develop the habit of reading source code. Read it like you would a book, whether it's the Dart and Flutter source code, or a third-party library. This will often teach you much more than you can learn from other sources.

In the next chapter, you'll dive even deeper into Dart's type system. See you there!

3 Types & Operations

Written by Jonathan Sande

Life is full of variety, and that variety expresses itself in different ways. What type of toothpaste do you use? Spearmint? Cinnamon? What's your blood type? A? B? O+? What type of ice cream do you like? Vanilla? Strawberry? Praline pecan fudge swirl? Having names for all of these different things helps you talk intelligently about them. It also helps you to recognize when something is out of place. After all, no one brushes their teeth with praline pecan fudge swirl. Though it does sound kind of nice.

Programming types are just as useful as real-life types. They help you to categorize all the different kinds of data you use in your code.

In Chapter 2, "Expressions, Variables & Constants", you learned how to name data using variables and also got a brief introduction to Dart data types. In this chapter, you'll learn even more about types and what you can do with them.

Data Types in Dart

In Dart, a **type** is a way to tell the compiler how you plan to use some data. By this point in this book, you've already seen the following types:

- `int`
- `double`
- `num`
- `dynamic`
- `String`

The last one in that list, `String`, is the type used for text like `'Hello, Dart!'`.

Just as you don't brush your teeth with ice cream, Dart types keep you from trying to do silly things like dividing text or removing whitespace from a number.

Dart has even more built-in types than just the ones listed above. The basic ones, such as `int`, `double`, and `num` will serve you adequately in a great variety of programming scenarios, but when working on projects with specific needs, it'll be convenient to create custom types instead. A weather app, for example, may need a `Weather` type, while a social media app may need a `User` type. You'll learn how to create your own types in Chapter 5, "Control Flow", and Chapter 8, "Classes".

As you learned in Chapter 2, "Expressions, Variables & Constants", types like `int`, `double` and `num` are subclasses, or **subtypes**, of the `Object` type. `Object` defines a few core operations, such as testing for equality and describing itself. Every non-nullable type in Dart is a subtype of `Object`, and as a subtype, shares `Object`'s basic functionality.

Note: You'll learn about nullable types in Chapter 11, "Nullability".

Type Inference

In the previous chapter, you also got a sneak peek at type inference, which you'll look at in more depth now.

Annotating Variables Explicitly

It's fine to always explicitly add the **type annotation** when you declare a variable. This means writing the data type before the variable name.

```
int myInteger = 10;
double myDouble = 3.14;
```

You annotated the first variable with `int` and the second with `double`.

Creating Constant Variables

Declaring variables the way you did above makes them mutable. If you want to make them immutable, but still keep the type annotation, you can add `const` or `final` in front.

These forms of declaration are fine with `const`:

```
const int myInteger = 10;
const double myDouble = 3.14;
```

They're also fine with `final`:

```
final int myInteger = 10;
final double myDouble = 3.14;
```

Note: Mutable data is convenient to work with because you can change it any time you like. However, many experienced software engineers have come to appreciate the benefits of immutable data. When a value is immutable, that means you can trust that no one will change that value after you create it. Limiting your data in this way prevents many hard-to-find bugs from creeping in, and also makes the program easier to reason about and to test.

Letting the Compiler Infer the Type

While it's permissible to include the type annotation as in the example above, it's redundant. You're smart enough to know that `10` is an `int` and `3.14` is a `double`, and it turns out the Dart compiler can deduce this as well. The compiler doesn't need you to explicitly tell it the type every time — it can figure the type out on its own through a process called **type inference**. Not all programming languages have type inference, but Dart does — and it's a key component behind Dart's power as a language.

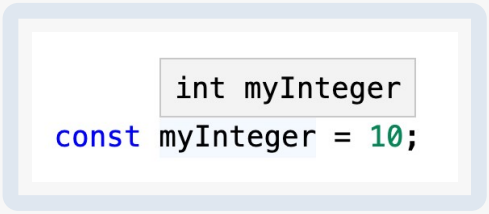
You can simply drop the type in most instances. For example, here are the constants from above without the type annotations:

```
const myInteger = 10;
const myDouble = 3.14;
```

Dart infers that `myInteger` is an `int` and `myDouble` is a `double`.

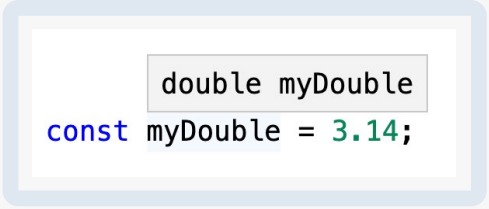
Checking the Inferred Type in VS Code

Sometimes, it can be useful to check the inferred type of a variable or constant. You can do this in VS Code by hovering your mouse pointer over the variable name. VS Code will display a popover like this:



VS Code shows you the inferred type. In this example, the type is `int`.

It works for other types, too. Hovering your mouse pointer over `myDouble` shows that it's a `double`:



Type inference isn't magic; Dart is simply doing what your own brain does very easily. Programming languages that don't use type inference often feel verbose, because you need to specify the (usually) obvious type each time you declare a variable or constant.

> **Note**: There are times when you'll want (or need) to explicitly include the type, either because Dart doesn't have enough information to figure it out, or because you want your intent to be clear to the reader. However, you'll see type inference used for most of the code examples in this book.

Checking the Type at Runtime

Your code can't hover a mouse pointer over a variable to check the type, but Dart does have a programmatic way of doing nearly the same thing: the `is` keyword.

```
num myNumber = 3.14;
print(myNumber is double);
print(myNumber is int);
```

Run this to see the following result:

```
true
false
```

Recall that both `double` and `int` are subtypes of `num`. That means `myNumber` could store either type. In this case, `3.14` is a `double`, and not an `int`, which is what the `is` keyword checks for and confirms by returning `true` and `false` respectively. You'll learn more about the type for `true` and `false` values in Chapter 5, "Control Flow".

Another option to see the type at runtime is to use the `runtimeType` property that is available to all types.

```
print(myNumber.runtimeType);
```

This prints `double` as expected.

Type Conversion

Sometimes, you'll have data in one type, but need to convert it to another. The naïve way to attempt this would be like so:

```
var integer = 100;
var decimal = 12.5;
integer = decimal;
```

Dart will complain if you try to do that:

```
A value of type 'double' can't be assigned to a variable of type 'int'.
```

Some programming languages aren't as strict and will perform conversions like this silently. Experience shows this kind of silent, implicit conversion is a frequent source of software bugs and often hurts code performance. Dart disallows you from assigning a value of one type to another and avoids these issues.

Remember, computers rely on programmers to tell them what to do. In Dart, that includes being explicit about type conversions. If you want the conversion to happen, you have to say so!

Instead of simply assigning and hoping for implicit conversion, you need to explicitly say that you want Dart to convert the type. You can convert this `double` to an `int` like so:

```
integer = decimal.toInt();
```

The assignment now tells Dart, unequivocally, that you want to convert from the original type, `double`, to the new type, `int`.

> **Note**: In this case, assigning the decimal value to the integer results in a loss of precision: The integer variable ends up with the value `12` instead of `12.5`. This is why it's important to be explicit. Dart wants to make sure you know what you're doing and that you may end up losing data by performing the type conversion.

Operators With Mixed Types

So far, you've only seen operators acting independently on integers or doubles. But what if you have an integer that you want to multiply with a double?

Take this example:

```
const hourlyRate = 19.5;
const hoursWorked = 10;
const totalCost = hourlyRate * hoursWorked;
```

`hourlyRate` is a `double` and `hoursWorked` is an `int`. What will the type of `totalCost` be? It turns out that Dart will make `totalCost` a `double`. This is the safest choice, since making it an `int` could cause a loss of precision.

If you actually do want an `int` as the result, then you need to perform the conversion explicitly:

```
const totalCost = (hourlyRate * hoursWorked).toInt();
```

The parentheses tell Dart to do the multiplication first, and after that, to take the result and convert it to an integer value. However, the compiler complains about this:

```
Const variables must be initialized with a constant value.
```

The problem is that `toInt` is a runtime method. This means that `totalCost` can't be determined at compile time, so making it `const` isn't valid. No problem; there's an easy fix. Just change `const` to `final`:

```
final totalCost = (hourlyRate * hoursWorked).toInt();
```

Now `totalCost` is an `int`.

Ensuring a Certain Type

Sometimes you want to define a constant or variable and ensure it remains a certain type, even though what you're assigning to it is of a different type. You saw earlier how you can convert from one type to another. For example, consider the following:

```
const wantADouble = 3;
```

Here, Dart infers the type of `wantADouble` as `int`. But what if you wanted the constant to store a `double` instead?

One thing you could do is the following:

```
final actuallyDouble = 3.toDouble();
```

This uses type conversion to convert `3` into a `double` before assignment, as you saw earlier in this chapter.

Another option would be to not use type inference at all, and to add the `double` annotation:

```
const double actuallyDouble = 3;
```

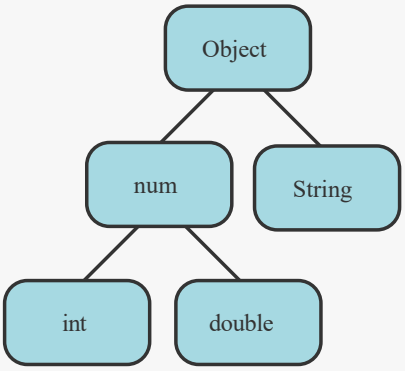
The number `3` is an integer, but literal number values that contain a decimal point cannot be integers, which means you could have avoided this entire discussion had you started with:

```
const wantADouble = 3.0;
```

Sorry! :]

Casting Down

The image below shows a tree of the types you've encountered so far. `Object` is a supertype of `num` and `String`, and `num` is a supertype of `int` and `double`. Conversely, `int` and `double` are subtypes of `num`, which is a subtype of `Object`.



Types at the top of the tree can only perform very general tasks. The further you go down the tree, the more specialized the types become and the more detailed their tasks.

At times, you may have a variable of some general supertype, but you need functionality that is only available in a subtype. If you're sure that the value of the variable actually is the subtype you need, then you can use the `as` keyword to change the type. This is known as **type casting**. When type casting from a supertype to a subtype, it's called **downcasting**.

Here's an example:

```
num someNumber = 3;
```

You have a number, and you want to check if it's even. You know that integers have an `isEven` property, so you attempt the following:

```
print(someNumber.isEven);
```

However, the compiler gives you an error:

```
The getter 'isEven' isn't defined for the type 'num'.
```

`num` is too general of a type to know anything about even or odd numbers. Only integers can be even or odd. The issue is that `num` could potentially be a `double` at runtime since `num` includes both `double` and `int`. In this case, though, you're sure that `3` is an integer, so you can cast `someNumber` to `int`.

```
final someInt = someNumber as int;
print(someInt.isEven);
```

The `as` keyword causes the compiler to recognize `someInt` as an `int`, so your code is now able to use the `isEven` property that belongs to the `int` type. Since `3` isn't even, Dart prints `false`.

You need to be careful with type casting, though. If you cast to the wrong type, you'll get a runtime error:

```
num someNumber = 3;
final someDouble = someNumber as double;
```

This will crash with the following message:

```
_CastError (type 'int' is not a subtype of type 'double' in type cast)
```

The runtime type of `someNumber` is `int`, not `double`. In Dart, you're not allowed to cast to a sibling type, such as `int` to `double`. You can only cast down to a subtype.

If you do need to convert an `int` to a `double` at runtime, use the `toDouble` method that you saw earlier:

```
final someDouble = someNumber.toDouble();
```

Exercises

1. Create a constant called `age1` and set it equal to `42`. Create another constant called `age2` and set it equal to `21`. Check that the type for both constants has been inferred correctly as `int` by hovering your mouse pointer over the variable names in VS Code.

2. Create a constant called `averageAge` and set it equal to the average of `age1` and `age2` using the operation `(age1 + age2) / 2`. Hover your mouse pointer over `averageAge` to check the type. Then check the result of `averageAge`. Why is it a `double` if the components are all `int`?

Object and dynamic Types

Dart grew out of the desire to solve some problems inherent in JavaScript. JavaScript is a **dynamically-typed** language. Dynamic means that something can change, and for JavaScript that means the types can change at runtime.

Here is an example in JavaScript:

```
var myVariable = 42;
myVariable = "hello";
```

In JavaScript, the first line is a `number` and the second line a `string`. Changing the types on the fly like this is completely valid in JavaScript. While this may be convenient at times, it makes it *really* easy to write buggy code. For example, you may be erroneously thinking that `myVariable` is still a number, so you write the following code:

```
var answer = myVariable * 3; // runtime error
```

Oops! That's an error because `myVariable` is actually a string, and the computer doesn't know what to do with "*hello*" times `3`. Not only is it an error, you won't even discover the error until you run the code.

You can totally prevent mistakes like that in Dart because it's an **optionally-typed** language. That means you can choose to use Dart as a dynamically typed language or as a **statically-typed** language. Static means that something *cannot* change; once you tell Dart what type a variable is, you're not allowed to change it anymore.

If you try to do the following in Dart:

```
var myVariable = 42;
myVariable = 'hello'; // compile-time error
```

The Dart compiler will immediately tell you that it's an error. That makes type errors trivial to detect.

As you saw in Chapter 2, "Expressions, Variables & Constants", the creators of Dart did include a `dynamic` type for those who wish to write their programs in a dynamically-typed way.

```
dynamic myVariable = 42;
myVariable = 'hello'; // OK
```

In fact, this is the default if you use `var` and don't initialize your variable:

```
var myVariable; // defaults to dynamic
myVariable = 42;      // OK
myVariable = 'hello'; // OK
```

While `dynamic` is built into the system, it's more of a concession rather than an encouragement to use it. **You should still embrace static typing** in your code as it will prevent you from making silly mistakes.

If you need to explicitly say that any type is allowed, you should consider using the `Object?` type.

```
Object? myVariable = 42;
myVariable = 'hello'; // OK
```

At runtime, `Object?` and `dynamic` behave nearly the same. However, when you explicitly declare a variable as `Object?`, you're telling everyone that you generalized your variable on purpose and that they'll need to check its type at runtime if they want to do anything specific with it. Using `dynamic`, on the other hand, is more like saying you don't know what the type is; you're telling people they can do what they like with this variable, but it's completely on them if their code crashes.

Note: You may be wondering what that question mark at the end of `Object?` is. That means that the type can include the `null` value. You'll learn more about nullability in Chapter 11, "Nullability".

Challenges

Before moving on, here are some challenges to test your knowledge of types and operations. It's best if you try to solve them yourself, but solutions are available with the supplementary materials for this book if you get stuck.

Challenge 1: Teacher's Grading

You're a teacher, and in your class, attendance is worth 20% of the grade, the homework is worth 30% and the exam is worth 50%. Your student got 90 points for her attendance, 80 points for her homework and 94 points on her exam. Calculate her grade as an integer percentage rounded down.

Challenge 2: What Type?

What's the type of `value`?

```
const value = 10 / 2;
```

Key Points

- Type conversion allows you to convert values of one type into another.
- When doing operations with basic arithmetic operators (`+`, `-`, `*`, `/`) and mixed types, the result will be a `double`.
- Type inference allows you to omit the type when Dart can figure it out.
- Dart is an optionally-typed language. While it's preferable to choose statically-typed variables, you may write Dart code in a dynamically-typed way by explicitly adding the `dynamic` type annotation in front of variables.

4 Strings

Written by Jonathan Sande

Numbers are essential in programming, but they aren't the only type of data you need to work with in your apps. Text is also a common data type, representing things such as people's names, their addresses, or even the complete text of a book. All of these are examples of text that an app might have to handle.

Most computer programming languages store text in a data type called a **string**. This chapter introduces you to strings, first by giving you background on the concept, and then by showing you how to use them in Dart.

How Computers Represent Strings

Computers think of strings as a collection of individual **characters**. Numbers are the language of CPUs, and all code, in every programming language, can be reduced to raw numbers. Strings are no different.

That may sound very strange. How can characters be numbers? At its base, a computer needs to be able to translate a character into the computer's own language, and it does so by assigning each character a different number. This two-way mapping between characters and numbers is called a **character set**.

When you press a character key on your keyboard, you're actually communicating the number of the character to the computer. Your computer converts that number into a picture of the character and finally, presents that picture to you.

Unicode

In isolation, a computer is free to choose whatever character set mapping it likes. If the computer wants the letter **a** to equal the number `10`, then so be it. But when computers start talking to each other, they need to use a common character set.

If two computers used different character sets, then when one computer transferred a string to another computer, they would end up thinking the strings contained different characters.

There have been several standards over the years, but the modern standard is **Unicode**. It defines the character set mapping that almost all computers use today.

> **Note**: You can read more about Unicode at its official website, unicode.org.

As an example, consider the word **cafe**. The Unicode standard tells us that the letters of this word should be mapped to numbers like so:

c	a	f	e
99	97	102	101

The number associated with each character is called a **code point**. So in the example above, **c** uses code point `99`, **a** uses code point `97`, and so on.

Of course, Unicode is not just for the simple Latin characters used in English, such as **c**, **a**, **f** and **e**. It also lets you map characters from languages around the world. The word **cafe**, as you're probably aware, is derived from French, in which it's written as **café**. Unicode maps these characters like so:

c	a	f	é
99	97	102	233

And here's an example using simplified Chinese characters that mean "I love you":

我	爱	你
25105	29233	20320

You're familiar with the small pictures called emojis that you can use when texting your friends. These pictures are, in fact, just normal characters and are also mapped by Unicode. For example:

🎯	😂
127919	128514

These are only two characters. The code points for them are very large numbers, but each is still only a single code point. The computer considers them no different than any other two characters.

> **Note**: The word “emoji” comes from the Japanese 絵文字, where “e” means picture and “moji” means character.

The numbers for each of the characters above were written in decimal notation, but you usually write Unicode code points in hexadecimal format. Here they are again in hex:

c	a	f	é
63	61	66	E9

我	爱	你
6211	7231	4F60

🎯	😂
1F3AF	1F602

Using base-16 makes the numbers more compact, easier to find in the Unicode character code charts and generally nicer to work with while programming.

Strings and Characters in Dart

Dart, like any good programming language, can work directly with strings. It does so through the `String` data type. In the remainder of this chapter, you’ll learn about this data type and how to work with it.

You’ve already seen a Dart string back in Chapter 1 where you printed one:

```
print('Hello, Dart!');
```

You can extract that same string as a named variable:

```
var greeting = 'Hello, Dart!';
print(greeting);
```

The right-hand side of this expression is known as a **string literal**. Due to type inference, Dart knows that `greeting` is of type `String`. Since you used the `var` keyword, you’re allowed to reassign the value of `greeting` as long as the new value is still a string.

```
var greeting = 'Hello, Dart!';
greeting = 'Hello, Flutter!';
```

Even though you changed the value of `greeting` here, you didn't modify the string itself. That's because strings are immutable in Dart. It's not like you replaced `Dart` in the first string with `Flutter`. No, you completely discarded the string `'Hello, Dart!'` and replaced it with a whole new string whose value was `'Hello, Flutter!'`.

> **Note**: The code examples that follow contain emoji characters that may be difficult to input on your keyboard. You can find all of them to conveniently copy-and-paste by opening **starter/bin/starter.dart** in the Chapter 4 supplemental materials for this book.
>
> Alternatively, you can use emojipedia.org with the following search terms: "dart", "Mongolia flag" and "man woman girl boy". Or on macOS, you can also press **Command-Control-Space** to open the Character Viewer and search for emojis.

Getting Characters

If you're familiar with other programming languages, you may be wondering about a `Character` or `char` type. Dart doesn't have that. Take a look at this example:

```
const letter = 'a';
```

So here, even though `letter` is only one character long, it's still of type `String`.

But strings are a collection of characters, right? What if you want to know the underlying number value of the character? No problem. Keep reading.

Dart strings are a collection of **UTF-16 code units**. UTF-16 is a way to encode Unicode values by using 16-bit numbers. If you want to find out what those UTF-16 codes are, you can do it like so:

```
var salutation = 'Hello!';
print(salutation.codeUnits);
```

This will print the following list of numbers in decimal notation:

```
[72, 101, 108, 108, 111, 33]
```

H is `72`, **e** is `101`, and so on.

These UTF-16 code units have the same value as Unicode code points for most of the characters you see on a day-to-day basis. However, 16 bits only give you 65,536 combinations, and believe it or not, there are more than 65,536 characters in the world! Remember the large numbers that the emojis had in the last section? You'll need more than 16 bits to represent those values.

UTF-16 has a special way of encoding code points higher than 65,536 by using two code units called **surrogate pairs**.

```
const dart = '🎯';
print(dart.codeUnits);
// [55356, 57263]
```

The code point for 🎯 is `127919`, but the surrogate pair for that in UTF-16 is `55356` and `57263`. No one wants to mess with surrogate pairs. It would be much nicer to just get Unicode code points directly. And you can! Dart calls them **runes**.

```
const dart = '🎯';
print(dart.runes);
// (127919)
```

Problem solved, right? If only it were.

Unicode Grapheme Clusters

Unfortunately, language is messy and so is Unicode. Have a look at this example:

```
const flag = '🇲🇳';
print(flag.runes);
// (127474, 127475)
```

Why are there two Unicode code points!? Well, it's because Unicode doesn't create a new code point every time there is a new country flag. It uses a pair of code points called **regional indicator symbols** to represent a flag. That's what you see in the example for the Mongolian flag above. `127474` is the code for the regional indicator symbol letter `M`, and `127475` is the code for the regional indicator symbol letter `N`. `MN` represents Mongolia.

> **Note**: Windows computers may not display the flag emojis. See the forums for more information.

If you thought that was complicated, look at this one:

```
const family = '👨‍👩‍👧‍👦';
print(family.runes);
// (128104, 8205, 128105, 8205, 128103, 8205, 128102)
```

That list of Unicode code points is a man, a woman, a girl and a boy all glued together with a character called a **Zero Width Joiner** or **ZWJ**.

Now imagine trying to find the length of that string:

```
const family = '👨‍👩‍👧‍👦';

family.length;            // 11
family.codeUnits.length; // 11
family.runes.length;      // 7
```

Getting the length of the string with `family.length` is equivalent to finding the number of UTF-16 code units: There are surrogate pairs for each of the four people plus the three ZWJ characters for a total of 11. Finding the runes gives you the seven Unicode code points that make up the emoji: man + ZWJ + woman + ZWJ + girl + ZWJ + boy. However, neither 11 nor 7 is what you'd expect. The family emoji looks like it's just one character. You'd think the length should be one!

When a string with multiple code points looks like a single character, this is known as a **user-perceived character**. In technical terms, it's called a **Unicode extended grapheme cluster**, or more commonly, just **grapheme cluster**.

Although the creators of Dart didn't support grapheme clusters in the language itself, they did make an add-on package that handles them.

Adding the Characters Package

This is a good opportunity to try out your first Pub package. In the root folder of your project, open **pubspec.yaml**.

> **Note**: If you don't see **pubspec.yaml**, go back to Chapter 1 to see how to create a new project. Alternatively, open the starter project that comes with the supplemental materials for this chapter.

Add the following two lines at the bottom of the file:

```
dependencies:
  characters: ^1.2.1
```

Here are some things to note:

- `dependencies` is the section name. Large Dart projects will typically include multiple dependencies. If the project you're working on already has a `dependencies` section, then you only need to add the `characters` line.
- You may also notice another section called `dev_dependencies`. These are the list of dependencies you'll use during development but are unneeded for a published app. For example, the `lints` package helps you find problems in your code as you write it.
- Indentation is important in `.yaml` files, so make sure to indent the package name with two spaces. `dependencies` should have no spaces in front of it.
- The `^` carat character means that any version higher than or equal to `1.2.1` but lower than `2.0.0` is OK to use in your project. This is known as **semantic versioning**.

Now press **Command+S** on a Mac or **Control+S** on a PC to save the changes to **pubspec.yaml**. VS Code will automatically fetch the package from Pub. Alternatively, you can press the **Get Packages button** in the top right. It looks like a down arrow:

Both of these methods are equivalent to running the following command in the root folder of your project using the terminal:

```
dart pub get
```

> **Note**: Whenever you download and open a new Dart project that contains Pub packages, you'll need to run `dart pub get` first. This includes the final and challenge projects included in the supplemental materials for this chapter.

Now that you've added the `characters` package to your project, go back to your Dart code file and add the following import to the top of the page:

```
import 'package:characters/characters.dart';
```

Now you can use the code in the `characters` package to handle grapheme clusters. This package adds extra functionality to the `String` type.

```
const family = '👩‍👩‍👧';
family.characters.length; // 1
```

Aha! Now that's what you'd hope to see: just one character for the family emoji. The `characters` package extended `String` to include a collection of grapheme clusters called `characters`.

In your own projects, you can decide whether you want to work with UTF-16 code units, Unicode code points or grapheme clusters. However, as a general rule, you should strongly consider using grapheme clusters any time you're receiving text input from the outside world. That includes fetching data over the network or users typing things into your app.

Single Quotes vs. Double Quotes

Dart allows you to use either single quotes or double quotes for string literals. Both of these are fine:

```
'I like cats'
"I like cats"
```

Although Dart doesn't have a recommended practice, the Flutter style guide does recommend using single quotes, so this book will also follow that practice.

You might want to use double quotes, though, if your string includes any apostrophes.

```
"my cat's food"
```

Otherwise, you would need to use the backslash `\` as an escape character so that Dart knows that the string isn't ending early:

```
'my cat\'s food'
```

Concatenation

You can do much more than create simple strings. Sometimes you need to manipulate a string, and one common way to do so is to combine it with another string. This is called **concatenation**...with no relation to the aforementioned felines.

In Dart, you can concatenate strings simply by using the addition operator. Just as you can

add numbers, you can add strings:

```
var message = 'Hello' + ' my name is ';
const name = 'Ray';
message += name;
// 'Hello my name is Ray'
```

You need to declare `message` as a variable, rather than a constant, because you want to modify it. You can add string literals together, as in the first line, and you can add string variables or constants together, as in the third line.

If you find yourself doing a lot of concatenation, you should use a string buffer, which is more efficient. You'll learn how to do this in Chapter 1, "String Manipulation", in *Dart Apprentice: Beyond the Basics*.

Interpolation

You can also build up a string by using **interpolation**, which is a special Dart syntax that lets you build a string in a manner that's easy for other people reading your code to understand:

```
const name = 'Ray';
const introduction = 'Hello my name is $name';
// 'Hello my name is Ray'
```

This is much more readable than the example in the previous section. It's an extension of the string literal syntax, in which you replace certain parts of the string with other values. You add a dollar sign (`$`) in front of the value that you want to insert.

The syntax works in the same way to build a string from other data types such as numbers:

```
const oneThird = 1 / 3;
const sentence = 'One third is $oneThird.';
```

Here, you use a `double` for the interpolation. Your `sentence` constant will contain the following value:

```
One third is 0.3333333333333333.
```

Of course, it would actually take an infinite number of characters to represent one-third as

a decimal because it's a repeating decimal. You can control the number of decimal places shown on a `double` by using `toStringAsFixed` along with the number of decimal places to show:

```
final sentence = 'One third is ${oneThird.toStringAsFixed(3)}.';
```

There are a few items of interest here:

- You're requesting the string to show only three decimal places.
- Since you're performing an operation on `oneThird`, you need to surround the expression with curly braces after the dollar sign. This lets Dart know that the dot (`.`) after `oneThird` isn't just a regular period.
- The `sentence` variable needs to be `final` now instead of `const` because `toStringAsFixed(3)` is calculated at runtime.

Here's the result:

```
One third is 0.333.
```

Exercises

1. Create a string constant called `firstName` and initialize it to your first name. Also create a string constant called `lastName` and initialize it to your last name.
2. Create a string constant called `fullName` by adding the `firstName` and `lastName` constants together, separated by a space.
3. Using interpolation, create a string constant called `myDetails` that uses the `fullName` constant to create a string introducing yourself. For example, Ray Wenderlich's string would read: `Hello, my name is Ray Wenderlich.`

Multi-Line Strings

Dart has a neat way to express strings that contain multiple lines, which can be rather useful when you need to use very long strings in your code.

You can support multi-line text like so:

```
const bigString = '''
You can have a string
that contains multiple
lines
by
doing this.''';
print(bigString);
```

The three single quotes (`'''`) signify that this is a multi-line string. Three double quotes (`"""`) would have done the same thing.

The example above will print the following:

```
You can have a string
that contains multiple
lines
by
doing this.
```

Notice that all of the newline locations are preserved. If you just want to use multiple lines in code but don't want the output string to contain newline characters, then you can surround each line with single quotes:

```
const oneLine = 'This is only '
    'a single '
    'line '
    'at runtime.';
```

That's because Dart ignores whitespace outside of quoted text. This does the same thing as if you concatenated each of those lines with the `+` operator:

```
const oneLine = 'This is only ' +
    'a single ' +
    'line ' +
    'at runtime.';
```

Either way, this is what you get:

```
This is only a single line at runtime.
```

Like many languages, if you want to insert a newline character, you can use `\n` .

```
const twoLines = 'This is\ntwo lines.';
```

Printing this gives:

```
This is
two lines.
```

But sometimes you want to ignore any special characters that a string might contain. To do that, you can create a **raw string** by putting **r** in front of the string literal.

```
const rawString = r'My name \n is $name.';
```

And that's exactly what you get:

```
My name \n is $name.
```

Inserting Characters From Their Codes

Similar to the way you can insert a newline character into a string using the `\n` escape sequence, you can also add Unicode characters if you know their codes. Take the following example:

```
print('I \u2764 Dart\u0021');
```

Here, you've used `\u`, followed by a four-digit hexadecimal code unit value. `2764` is the hex value for the heart emoji, and `21` is the hex value for an exclamation mark. Since `21` is only two digits, you pad it with extra zeros as `0021`.

This prints:

```
I ❤ Dart!
```

For code points with values higher than hexadecimal `FFFF`, you need to surround the code with curly braces:

```
print('I love \u{1F3AF}');
```

This prints:

```
I love 🎯
```

In this way, you can form any Unicode string from its codes.

You've come to the end of this chapter, but you can look forward to Chapter 1, "String Manipulation", in *Dart Apprentice: Beyond the Basics* to take you to the next level of working with text in Dart.

Challenges

Before moving on, here are some challenges to test your knowledge of strings. It's best if you try to solve them yourself, but solutions are available with the supplementary materials for this book if you get stuck.

As described in the **Getting Characters** section above, you can find the required emoji characters in the **starter** project or from emojipedia.org where you can use the search terms "Chad flag", "Romania flag" and "thumbs up dark skin tone".

Challenge 1: Same Same, but Different

This string has two flags that look the same. But they aren't! One of them is the flag of Chad and the other is the flag of Romania.

```
const twoCountries = 'TDRO';
```

Which is which?

> **Hint**: Romania's regional indicator sequence is `RO`, and `R` is `127479` in decimal. Chad, which is *Tishād* in Arabic and *Tchad* in French, has a regional indicator sequence of `TD`. Sequence letter `T` is `127481` in decimal.

Challenge 2: How Many?

Given the following string:

```
const vote = 'Thumbs up! 👍🏿';
```

- How many UTF-16 code units are there?
- How many Unicode code points are there?
- How many Unicode grapheme clusters are there?

Challenge 3: Find the Error

What is wrong with the following code?

```
const name = 'Ray';
name += ' Wenderlich';
```

Challenge 4: In Summary

What is the value of the constant named `summary`?

```
const number = 10;
const multiplier = 5;
final summary = '$number \u00D7 $multiplier = ${number * multiplier}';
```

Key Points

- Unicode is the standard representation for mapping characters to numbers.
- Dart uses UTF-16 values known as code units to encode Unicode strings.
- A single mapping in Unicode is called a code point, which is known as a rune in Dart.
- User-perceived characters may be composed of one or more code points and are called grapheme characters.
- You can combine strings by using the addition operator.
- You can make multi-line strings using three single quotes or double quotes.
- You can use string interpolation to build a string in place.

5 Control Flow

Written by Jonathan Sande

When writing a computer program, you need to be able to tell the computer what to do in different scenarios. For example, a calculator app performs one action if the user taps the addition button and another if they tap the subtraction button.

In computer programming terms, this concept is known as **control flow**, because you can control the flow of decisions the code makes at multiple points. In this chapter, you'll learn how to make decisions in your programs.

Making Comparisons

You've already encountered a few different Dart types, such as `int`, `double` and `String`. Each of those types is a data structure that's designed to hold a particular type of data. The `int` type is for integers while the `double` type is for decimal numbers. `String`, by comparison, is useful for storing textual information.

A new way of structuring information, though, requires a new data type. Consider the answers to the following questions:

- Is the door open?
- Do pigs fly?
- Is that the same shirt you were wearing yesterday?
- Is the traffic light red?
- Are you older than your grandmother?
- Does this make me look fat?

These are all yes-no questions. If you want to store the answers in a variable, you could use strings like `'yes'` and `'no'`. You could even use integers where `0` means no and `1` means yes. The problem with that, though, is what happens when you get `42` or `'celery'`? It would be better to avoid any ambiguity and have a type in which the only possible values are `yes` and `no`.

Boolean Values

Dart has a data type just for this. It's called `bool`, which is short for **Boolean**. A Boolean value can have one of two states. While in general, you could refer to the states as yes and no, on and off, or 1 and 0, most programming languages, Dart included, call them **true** and **false**.

The word Boolean was named after **George Boole**, the man who pioneered an entire field of mathematics around the concept of true and false. Since computers themselves are based on electrical circuits which can be in a binary state of on or off, Boolean math is fundamental to computer science.

When programming in a high-level language like Dart, you don't need to understand all of the Boolean logic that's happening at the circuit level, but there's still a lot about Boolean math you can apply to decision making in your own code.

To start your exploration of Booleans in Dart, create two Boolean variables like so:

```
const bool yes = true;
const bool no = false;
```

Because of Dart's type inference, you can leave off the type annotation:

```
const yes = true;
const no = false;
```

In the code above, you use the keywords `true` and `false` to set the state of each Boolean constant.

Boolean Operators

Booleans are commonly used to compare values. For example, you may have two values and you want to know if they're equal. Either they *are* equal, which would be `true`, or they *aren't* equal, which would be `false`.

Next, you'll see how to make that comparison in Dart.

Testing Equality

You can test for equality using the **equality operator**, which is denoted by `==`, that is, two equals signs.

Write the following line:

```
const doesOneEqualTwo = (1 == 2);
```

Dart infers that `doesOneEqualTwo` is a `bool`. Clearly, `1` does not equal `2`, and therefore `doesOneEqualTwo` will be `false`. Confirm that result by printing the value:

```
print(doesOneEqualTwo);
```

Sometimes you need parentheses to tell Dart what should happen first. However, the parentheses in that last example were there only for readability, that is, to show you that the two objects being compared were `1` and `2`. You could have also written it like so:

```
const doesOneEqualTwo = 1 == 2;
```

> **Note**: You may use the equality operator to compare `int` to `double` since they both belong to the `num` type.

Testing Inequality

You can also find out if two values are *not* equal using the `!=` operator:

```
const doesOneNotEqualTwo = (1 != 2);
```

This time, the result of the comparison is `true` because `1` does not equal `2`, so `doesOneNotEqualTwo` will be `true`.

> **Note**: `doesOneNotEqualTwo` isn't a great variable name. Avoid negative names for Boolean variables because if the value is false then you have a double negative and that just makes your brain hurt!

The prefix `!` operator, also called the **not-operator** or **bang operator**, toggles `true` to `false` and `false` to `true`. Another way to write the statement above is:

```
const alsoTrue = !(1 == 2);
```

Because `1` does not equal `2`, `(1 == 2)` is `false`, and then `!` flips it to `true`.

Testing Greater and Less Than

There are two other operators to help you compare two values and determine if a value is greater than (`>`) or less than (`<`) another value. You know these from mathematics:

```
const isOneGreaterThanTwo = (1 > 2);
const isOneLessThanTwo = (1 < 2);
```

It's not rocket science to work out that `isOneGreaterThanTwo` will equal `false` and that `isOneLessThanTwo` will equal `true`.

The `<=` operator lets you test if a value is *less than or equal to* another value. It's a combination of `<` and `==`, and will therefore return `true` if the first value is less than, or equal to, the second value.

```
print(1 <= 2); // true
print(2 <= 2); // true
```

Similarly, the `>=` operator lets you test if a value is *greater than or equal to* another value.

```
print(2 >= 1); // true
print(2 >= 2); // true
```

Boolean Logic

Each of the examples above tests just one condition. When George Boole invented the Boolean, he had much more planned for it than these humble beginnings. He invented Boolean logic, which lets you combine multiple conditions to form a result.

AND Operator

Ray would like to go cycling in the park with Vicki this weekend. It's a little uncertain whether they can go, though. There's a chance that it might rain. Also, Vicky says she can't go unless she finishes up the art project she's working on. So Ray and Vicki will go cycling in the park if it's sunny *and* Vicki finishes her work.

When two conditions need to be true in order for the result to be true, this is an example of a Boolean **AND** operation. If both input Booleans are true, then the result is true. Otherwise, the result is false. If it rains, Ray won't go cycling with Vicki. Or if Vicki doesn't finish her work, they won't go cycling, either.

In Dart, the operator for Boolean AND is written `&&`, used like so:

```
const isSunny = true;
const isFinished = true;
const willGoCycling = isSunny && isFinished;
```

Print `willGoCycling` and you'll see that it's `true`. If either `isSunny` or `isFinished` were `false`, then `willGoCycling` would also be `false`.

OR Operator

Vicki would like to draw a platypus, but she needs a model. She could either travel to Australia *or* she could find a photograph on the internet. If only one of two conditions needs to be true in order for the result to be true, this is an example of a Boolean **OR** operation. The only instance where the result would be false is if *both* input Booleans were false. If Vicki doesn't go to Australia and she also doesn't find a photograph on the internet, then she won't draw a platypus.

In Dart, the operator for Boolean OR is written `||`, used like so:

```
const willTravelToAustralia = true;
const canFindPhoto = false;
const canDrawPlatypus = willTravelToAustralia || canFindPhoto;
```

Print `canDrawPlatypus` to see that its value is `true`. If both values on the right were `false`, then `canDrawPlatypus` would be `false`. If both were `true`, then `canDrawPlatypus` would still be `true`.

Operator Precedence

As was the case in the Ray and Vicki examples above, Boolean logic is usually applied to multiple conditions. When you want to determine if two conditions are true, you use AND, while if you only care whether one of the two conditions is true, you use OR.

Here are a few more examples:

```
const andTrue = 1 < 2 && 4 > 3;
const andFalse = 1 < 2 && 3 > 4;
const orTrue = 1 < 2 || 3 > 4;
const orFalse = 1 == 2 || 3 == 4;
```

Each of these tests two separate conditions, combining them with either AND or OR.

It's also possible to use Boolean logic to combine more than two comparisons. For example, you can form a complex comparison like so:

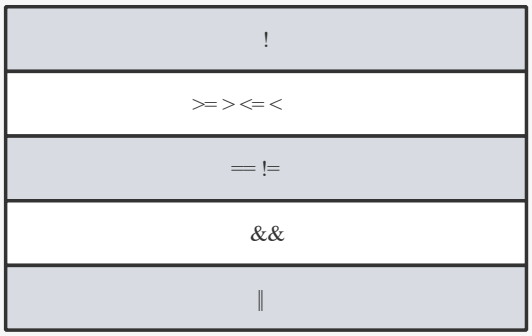
```
3 > 4 && 1 < 2 || 1 < 4
```

But now it gets a little confusing. You have three conditions with two different logical operators. With the comparisons simplified, you have the following form:

```
false && true || true
```

Depending on the order you perform the AND and OR operations, you get different results. If you evaluate AND first, the whole expression is `true`, while if you evaluate OR first, the whole expression is `false`.

This is where **operator precedence** comes in. The following list shows the order that Dart uses to evaluate expressions containing comparison and logical operators:



Operators higher in the list are executed before operators lower in the list. You can see that `&&` has a higher precedence than `||`. So back to the case from before:

```
false && true || true
```

First Dart will evaluate `false && true`, which is `false`. Then Dart will take that `false` to evaluate `false || true`, which is `true`. Thus the whole expression evaluates to `true`.

Overriding Precedence With Parentheses

If you want to override the default operator precedence, you can put parentheses around the parts Dart should evaluate first.

Compare the following two expressions:

```
3 > 4 && (1 < 2 || 1 < 4)  // false
(3 > 4 && 1 < 2) || 1 < 4  // true
```

The parentheses in the first line force Dart to do the OR operation before the AND operation, even though that isn't the default order. This results in the entire expression evaluating to `false` instead of `true`, as it would have if you hadn't used parentheses.

Even when parentheses are not strictly required, as in the second of the two expressions

above, they can still help to make the code more readable. For this reason, it's usually a good idea to use parentheses when you're performing a logical operation on more than two conditions.

String Equality

Sometimes you'll want to determine if two strings are equal. For example, a children's game of naming an animal in a photo would need to determine if the player answered correctly.

In Dart, you can compare strings using the standard equality operator, `==`, in exactly the same way as you compare numbers. For example:

```
const guess = 'dog';
const guessEqualsCat = guess == 'cat';
```

Here, `guessEqualsCat` is a Boolean, which in this case is `false` because the string `'dog'` does not equal the string `'cat'`.

Exercises

1. Create a constant called `myAge` and set it to your age. Then, create a constant named `isTeenager` that uses Boolean logic to determine if the age denotes someone in the age range of 13 to 19.
2. Create another constant named `maryAge` and set it to `30`. Then, create a constant named `bothTeenagers` that uses Boolean logic to determine if both you and Mary are teenagers.
3. Create a `String` constant named `reader` and set it to your name. Create another `String` constant named `ray` and set it to `'Ray Wenderlich'`. Create a Boolean constant named `rayIsReader` that uses string equality to determine if `reader` and `ray` are equal.

Now that you understand Boolean logic, you're going to use that knowledge to make decisions in your code.

The If Statement

The first and most common way of controlling the flow of a program is through the use of an `if` **statement**, which allows the program to do something only if a certain condition is true. For example, consider the following:

```
if (2 > 1) {
  print('Yes, 2 is greater than 1.');
}
```

This is a simple `if` statement. The **condition**, which is always a Boolean expression, is the

part within the parentheses that follows the `if` statement. If the condition is `true`, then the `if` statement will execute the code between the braces. If the condition is `false`, then the `if` statement *won't* execute the code between the braces.

Obviously, the condition `(2 > 1)` is `true`, so when you run that you'll see:

```
Yes, 2 is greater than 1.
```

The Else Clause

You can extend an `if` statement to provide code to run in the event that the condition turns out to be `false`. This is known as an **else clause**.

Here's an example:

```
const animal = 'Fox';
if (animal == 'Cat' || animal == 'Dog') {
  print('Animal is a house pet.');
} else {
  print('Animal is not a house pet.');
}
```

If `animal` equals either `'Cat'` or `'Dog'`, then the statement will execute the first block of code. If `animal` does *not* equal either `'Cat'` or `'Dog'`, then the statement will run the block inside the `else` clause of the `if` statement.

Run that code and you'll see the following in the debug console:

```
Animal is not a house pet.
```

Else-If Chains

You can go even further with `if` statements. Sometimes you want to check one condition, and then check another condition if the first condition isn't true. This is where `else-if` comes into play, nesting another `if` statement in the `else` clause of a previous `if` statement.

You can use it like so:

```
const trafficLight = 'yellow';
var command = '';
if (trafficLight == 'red') {
  command = 'Stop';
} else if (trafficLight == 'yellow') {
  command = 'Slow down';
} else if (trafficLight == 'green') {
  command = 'Go';
} else {
  command = 'INVALID COLOR!';
}
print(command);
```

In this example, the first `if` statement will check if `trafficLight` is equal to `'red'`. Since it's not, the next `if` statement will check if `trafficLight` is equal to `'yellow'`. It is equal to `'yellow'`, so no check will be made for the case of `'green'`.

Run the code and it will print the following:

```
Slow down
```

These nested `if` statements test multiple conditions, one by one, until a true condition is found. Only the code associated with the first true condition will be executed, regardless of whether there are subsequent `else-if` conditions that evaluate to `true`. In other words, the order of your conditions matters!

You can add an `else` clause at the end to handle the case where none of the conditions are `true`. This `else` clause is optional if you don't need it. In this example, you *do* need the `else` clause to ensure that `command` has a value by the time you print it out.

Variable Scope

`if` statements introduce a new concept called **scope**. Scope is the extent to which a variable can be seen throughout your code. Dart uses curly braces as the boundary markers in determining a variable's scope. If you define a variable inside a pair of curly braces, then you're not allowed to use that variable outside of those braces.

To see how this works, replace the `main` function with the following code:

```
const global = 'Hello, world';

void main() {
  const local = 'Hello, main';

  if (2 > 1) {
    const insideIf = 'Hello, anybody?';

    print(global);
    print(local);
    print(insideIf);
  }
```

```
  print(global);
  print(local);
  print(insideIf); // Not allowed!
}
```

Note the following points:

- There are three variables: `global`, `local` and `insideIf`.
- There are two sets of nested curly braces, one for the body of `main` and one for the body of the `if` statement.
- The variable named `global` is defined outside of the main function and outside of any curly braces. That makes it a **top-level variable**, which means it has a global scope. That is, it's visible everywhere in the file. You can see `print(global)` references it both in the `if` statement body and in the `main` function body.
- The variable named `local` is defined inside the body of the `main` function. This makes it a **local variable** and it has local scope. It's visible inside the `main` function, including inside the `if` statement, but `local` is not visible outside of the `main` function.
- The variable named `insideIf` is defined inside the body of the `if` statement. That means `insideIf` is only visible within the scope defined by the `if` statement's curly braces.

Since the final `print` statement is trying to reference `insideIf` outside of its scope, Dart gives you the following error:

```
Undefined name 'insideIf'.
```

Delete that final `print` statement to get rid of the error.

As a general rule, you should make your variables have the smallest scope that they can get by with. Another way to say that is, define your variables as close to where you use them as possible. Doing so makes their purpose more clear, and it also prevents you from using or changing them in places where you shouldn't.

The Ternary Conditional Operator

You've worked with operators that have two operands. For example, in `(myAge > 16)`, the two operands are `myAge` and `16`. But there's also an operator that takes three operands: the **ternary conditional operator**. It's related to `if` statements — you'll see why this is in just a bit.

First consider the example of telling a student whether their exam score is passing or not. Write an `if-else` statement to achieve this:

```
const score = 83;

String message;
if (score >= 60) {
  message = 'You passed';
} else {
  message = 'You failed';
}
```

That's pretty clear, but it's a lot of code. Wouldn't it be nice if you could shrink this to just a couple of lines? Well, you can, thanks to the ternary conditional operator!

The ternary conditional operator takes a condition and returns one of two values, depending on whether the condition is true or false. The syntax is as follows:

```
(condition) ? valueIfTrue : valueIfFalse;
```

Use the ternary conditional operator to rewrite your long code block above, like so:

```
const score = 83;
const message = (score >= 60) ? 'You passed' : 'You failed';
```

In this example, the condition to evaluate is `score >= 60`. If the condition is `true`, the result assigned to `message` will be `'You passed'`; if the condition is `false`, the result will instead be `'You failed'`. Since `83` is greater than `60`, the student receives good news.

The ternary conditional operator makes basic `if-else` statements much more compact, which in turn can make your code more readable.

However, for situations where using this operator makes your code *less* readable, then stick with the full `if-else` statement. Readability is always more important than fancy programming tricks that give the same result.

Exercises

1. Create a constant named `myAge` and initialize it with your age. Write an `if` statement to print out "Teenager" if your age is between `13` and `19`, and "Not a teenager" if your age is not between `13` and `19`.

2. Use a ternary conditional operator to replace the `else-if` statement that you used above. Set the result to a variable named `answer`.

Switch Statements

An alternate way to handle control flow, especially for multiple conditions, is with a `switch` statement. The `switch` statement takes the following form:

```
switch (variable) {
  case value1:
    // code
    break;
  case value2:
    // code
    break;

    ...

  default:
    // code
}
```

There are a few different keywords, so here are what they mean:

- **`switch`**: Based on the value of the variable in parentheses, which can be an `int`, `String` or compile-time constant, `switch` will redirect the program control to one of the `case` values that follow.
- **`case`**: Each `case` keyword takes a value and compares that value using `==` to the variable after the `switch` keyword. You add as many `case` statements as there are values to check. When there's a match, Dart will run the code that follows the colon.
- **`break`**: The `break` keyword tells Dart to exit the switch statement because the code in the `case` block is finished.
- **`default`**: If none of the `case` values match the `switch` variable, then the code after `default` will be executed.

The following sections will provide more detailed examples of `switch` statements.

Replacing Else-If Chains

Using `if` statements are convenient when you have one or two conditions, but the syntax can be a little verbose when you have a lot of conditions. Check out the following example:

```
const number = 3;
if (number == 0) {
  print('zero');
} else if (number == 1) {
  print('one');
} else if (number == 2) {
  print('two');
} else if (number == 3) {
  print('three');
} else if (number == 4) {
  print('four');
} else {
  print('something else');
}
```

Run that code and you'll see that it gets the job done — it prints "three" as expected. The wordiness of the `else-if` lines makes the code kind of hard to read, though.

Rewrite the code above using a `switch` statement:

```
const number = 3;
switch (number) {
  case 0:
    print('zero');
    break;
  case 1:
    print('one');
    break;
  case 2:
    print('two');
    break;
  case 3:
    print('three');
    break;
  case 4:
    print('four');
    break;
  default:
    print('something else');
}
```

Execute this code and you'll get the same result of "three" again. However, the code looks cleaner than the `else-if` chain because you didn't need to include the explicit condition check for every `case`.

> **Note**: In Dart, `switch` statements don't support ranges like `number > 5`. Only `==` equality checking is allowed. If your conditions involve ranges, then you should use `if` statements.

Switching on Strings

A `switch` statement also works with strings. Try the following example:

```
const weather = 'cloudy';
switch (weather) {
  case 'sunny':
    print('Put on sunscreen.');
    break;
  case 'snowy':
    print('Get your skis.');
    break;
  case 'cloudy':
  case 'rainy':
    print('Bring an umbrella.');
    break;
  default:
    print("I'm not familiar with that weather.");
}
```

Run the code above and the following will be printed in the console:

```
Bring an umbrella.
```

In this example, the `'cloudy'` case was completely empty, with no `break` statement. Therefore, the code "falls through" to the `'rainy'` case. This means that if the value is equal to either `'cloudy'` or `'rainy'`, then the `switch` statement will execute the same case.

Enumerated Types

Enumerated types, also known as **enums**, play especially well with `switch` statements. You can use them to define your own type with a finite number of options.

Consider the previous example with the `switch` statement about weather. You're expecting `weather` to contain a string with a recognized weather word. But it's conceivable that you might get something like this from one of your users:

```
const weather = 'I like turtles.';
```

You'd be like, "What? What are you even talking about?"

That's what the `default` case was there for — to catch all the weird stuff that gets through. Wouldn't it be nice to make weird stuff impossible, though? That's where enums come in.

Create the `enum` as follows, placing it outside of the `main` function:

```
enum Weather {
  sunny,
  snowy,
  cloudy,
  rainy,
}
```

Here are a couple of notes:

- The reason you need to put the enum outside of the `main` function is that enums are classes. You'll learn more about classes in Chapter 8, "Classes", but classes define new data types, and Dart requires these class definitions to be outside of functions.
- The enum above defines four different kinds of weather. Yes, yes, you can probably think of more kinds than that; feel free to add them yourself. But please don't make `iLikeTurtles` an option. Separate each of the values with a comma.

Formatting tip: If you like the enum options listed in a vertical column as they are above, make sure the final item in the list has a comma after it. On the other hand, if you like them laid out horizontally, remove the comma after the last item. Once you've done that, pressing **Shift+Option+F** on a Mac or **Shift+Alt+F** on a PC in VS Code will auto-format it to your preferred style:

`enum Weather { sunny, snowy, cloudy, rainy }`

This formatting trick works with many kinds of lists in Dart.

Naming Enums

When creating an `enum` in Dart, it's customary to write the `enum` name with an initial capital letter, as `Weather` was written in the example above. The values of an enum should use `lowerCamelCase` unless you have a special reason to do otherwise.

Switching on Enums

Now that you have the `enum` defined, you can use a `switch` statement to handle all the possibilities, like so:

```
const weatherToday = Weather.cloudy;
switch (weatherToday) {
  case Weather.sunny:
    print('Put on sunscreen.');
    break;
  case Weather.snowy:
    print('Get your skis.');
    break;
  case Weather.cloudy:
  case Weather.rainy:
    print('Bring an umbrella.');
    break;
}
```

As before, this will print the following message:

```
Bring an umbrella.
```

Notice that there was no `default` case this time since you handled every single possibility. In fact, Dart will warn you if you leave one of the enum items out. That'll save you some time chasing bugs.

Enum Values and Indexes

Before leaving the topic of enums, there's one more thing to note. If you try to print an enum, you'll get its value:

```
print(weatherToday);
// Weather.cloudy
```

Unlike some languages, a Dart enum isn't an integer. However, you can get the index, or ordinal placement, of a value in the enum like so:

```
final index = weatherToday.index;
```

Since `cloudy` is the third value in the enum, the zero-based `index` is `2`.

One reason you might want to convert an enum to an `int` is so that you can save its value. It often isn't possible to directly save an enum to storage, but you can save an `int` and then later convert the `int` back into an enum. You need to be careful, though, because if you subsequently change the order of the items in the enum, you'll get unexpected results when you try to convert the originally saved `int` back into the new enum.

You've learned how to use the basic features of enums, but they're a lot more powerful than the simple use case you saw here. In the next book, *Dart Apprentice: Beyond the Basics*, you'll learn about the powers of **enhanced enums**.

Avoiding the Overuse of Switch Statements

Switch statements, or long `else-if` chains, can be a convenient way to handle a long list of conditions. If you're a beginning programmer, go ahead and use them; they're easy to use and understand.

However, if you're an intermediate programmer and still find yourself using `switch` statements a lot, there's a good chance you could replace some of them with more advanced programming techniques that will make your code easier to maintain. If you're interested, do a web search for **refactoring switch statements with polymorphism** and read a few articles about it.

Challenges

Before moving on, here are some challenges to test your knowledge of control flow. It's best if you try to solve them yourself, but solutions are available in the `challenge` folder if you get stuck.

Challenge 1: Find the Error

What's wrong with the following code?

```
const firstName = 'Bob';
if (firstName == 'Bob') {
  const lastName = 'Smith';
} else if (firstName == 'Ray') {
  const lastName = 'Wenderlich';
}
final fullName = firstName + ' ' + lastName;
```

Challenge 2: Boolean Challenge

In each of the following statements, what is the value of the Boolean expression?

```
true && true
false || false
(true && 1 != 2) || (4 > 3 && 100 < 1)
((10 / 2) > 3) && ((10 % 2) == 0)
```

Challenge 3: Audio Enumerations

1. Make an `enum` called `AudioState` and give it values to represent `playing`, `paused` and `stopped` states.
2. Create a constant called `audioState` and give it an `AudioState` value.
3. Write a `switch` statement that prints a message based on the value.

Key Points

- The Boolean data type `bool` can represent `true` or `false`.
- The comparison operators, all of which return a Boolean, are:

Name	Operator
Equal	==
Not Equal	!=
Less than	<
Greater than	>
Less than or equal	<=
Greater than or equal	>=

- Use Boolean logic (`&&` and `||`) to combine comparison conditions.
- `if` statements are for making simple decisions based on a condition.
- Use `else` and `else-if` after an `if` statement to extend the decision making beyond a single condition.
- Variables and constants belong to a certain scope, beyond which you can't use them. A scope inherits variables and constants from its parent.
- The ternary operator (`a ? b : c`) can replace a simple `if-else` statement.
- `switch` statements can replace long `else-if` chains.
- Enums define a new type with a finite list of distinct values.

6 Loops

Written by Jonathan Sande

In the first five chapters of this book, your code ran from the top of the `main` function to the bottom, and then it was finished. With the addition of `if` statements in the previous chapter, you gave your code the opportunity to make decisions. However, it's still running from top to bottom, albeit following different branches.

Rather than just running through a set of instructions once, it's often useful to repeat tasks. Think about all the repetitious things you do every day:

- **Breathing**: Breathe in, breathe out, breathe in, breathe out...
- **Walking**: Right leg forward, left leg forward, right leg forward, left leg forward...
- **Eating**: Spoon up, spoon down, chew, chew, chew, swallow, repeat...

Computer programming is just as full of repetitive actions as your life is. The way you can perform these actions is by using **loops**. Dart, like many programming languages, has `while` loops and `for` loops. You'll learn how to make them in the following sections.

While Loops

A **while loop** repeats a block of code as long as a Boolean condition is true. You create a while loop like so:

```
while (condition) {
  // loop code
}
```

The loop checks the condition on every iteration. If the condition is `true`, then the loop executes and moves on to another iteration. If the condition is `false`, then the loop stops. Just like `if` statements, `while` loops introduce a scope because of their curly braces.

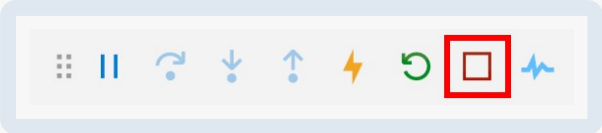
The simplest `while` loop takes this form:

```
while (true) { }
```

This is a `while` loop that never ends because the condition is always `true`. Of course, you would never write such a `while` loop, because your program would spin forever! This situation is known as an **infinite loop**, and while it might not cause your program to crash,

it will very likely cause your computer to freeze.

If you actually tried to run that infinite loop in VS Code and can't figure out how to make it stop, press the **Stop** button:



Here's a somewhat more useful example of a `while` loop:

```
var sum = 1;
while (sum < 10) {
  sum += 4;
  print(sum);
}
```

Run that to see the result. The loop executes as follows:

- Before 1st iteration: sum = `1`, loop condition = `true`
- After 1st iteration: sum = `5`, loop condition = `true`
- After 2nd iteration: sum = `9`, loop condition = `true`
- After 3rd iteration: sum = `13`, loop condition = `false`

After the third iteration, the `sum` variable is `13`, and therefore the loop condition of `sum < 10` becomes `false`. At this point, the loop stops.

Do-While Loops

A variant of the `while` loop is called the `do-while` loop. It differs from the `while` loop in that the condition is evaluated at the *end* of the loop rather than at the beginning. Thus, the body of a `do-while` loop is always executed at least once.

You construct a `do-while` loop like this:

```
do {
  // loop code
} while (condition)
```

Whatever statements appear inside the braces will be executed. Finally, if the `while` condition after the closing brace is `true`, you jump back up to the beginning and repeat the loop.

Here's the example from the last section, but using a `do-while` loop:

```
var sum = 1;
do {
  sum += 4;
  print(sum);
} while (sum < 10);
```

In this example, the outcome is the same as before.

Comparing While and Do-While Loops

It's possible to only use `while` loops, but in some cases, your code will be cleaner with a `do-while` loop. Take the following `while` loop as an example:

```
var sum = (1 + 3 - 2 * 4 + 8);
while (sum < 10) {
  sum += (1 + 3 - 2 * 4 + 8);
}
print('while loop sum: $sum');
```

All that math in the parentheses is there merely to represent a complex operation that you need to run at least once, and then run again on every iteration of the loop.

> **Note**: In Chapter 7, "Functions", you'll learn about grouping related code into a single complex operation called a **function**. Since you haven't studied functions yet, though, this chapter represents the idea by using a series of mathematical operations.

The problem with the `while` loop above is that you need to repeat `(1 + 3 - 2 * 4 + 8)` both to initialize the variable and on every iteration of the loop. Using a `do-while` loop eliminates the need for repetition:

```
var sum = 0;
do {
  sum += (1 + 3 - 2 * 4 + 8);
} while (sum < 10);
print('do-while loop sum: $sum');
```

Here, you only wrote the complex operation code once. Less repetition makes for cleaner code!

Breaking Out of a Loop

Sometimes you'll need to break out of a loop early. You can do this using the `break` statement, just as you did from inside the `switch` statement earlier. This immediately stops the execution of the loop and continues on to the code that follows the loop.

For example, consider the following `while` loop:

```
sum = 1;
while (true) {
  sum += 4;
  if (sum > 10) {
    break;
  }
}
```

Here, the loop condition is `true`, so the loop would normally iterate forever. However, the `break` means the `while` loop will exit once the sum is greater than `10`.

You've now seen how to write the same loop in different ways. This demonstrates that in computer programming there are often many ways to achieve the same result. You should choose the method that's easiest to read and that conveys your intent in the best way possible. This is an approach you'll internalize with enough time and practice.

Exercise

- Create a variable named `counter` and set it equal to `0`.
- Create a `while` loop with the condition `counter < 10`.
- The loop body should print out "counter is X" (where X is replaced with the value of `counter`) and then increment `counter` by 1.

For Loops

In addition to `while` loops, Dart has another type of loop called a **for loop**. This is probably the most common loop you'll see, and you use it to run a block of code a set number of times.

Here's a simple example of a C-style `for` loop in Dart:

```
for (var i = 0; i < 5; i++) {
  print(i);
}
```

If you have some prior programming experience, this C programming language style `for` loop probably looks very familiar to you. If not, though, the first line would be confusing. Here's a summary of the three parts between the parentheses and separated by semicolons:

- `var i = 0` (**initialization**): Before the loop starts, you create a counter variable to keep track of how many times you've looped. You could call the variable anything, but `i` is commonly used as an abbreviation for *index*. You then initialize it with some value; in this case, `0`.
- `i < 5` (**condition**): This is the condition that the `for` loop will check before every iteration of the loop. If it's `true`, then it will run the code inside the braces. But if it's `false`, then the loop will end.
- `i++` (**action**): The action runs at the end of every iteration, usually to update the loop index value. It's common to increment by `1` using `i++` but you could just as easily use `i += 2` to increment by `2` or `i--` to decrement by `1`.

Run the previous code and you'll see the following output:

```
0
1
2
3
4
```

The counter index `i` started at `0` and continued until it equaled `5`. At that point, the `for` loop condition `i < 5` was `false`, so the loop exited before running the `print` statement again.

The Continue Keyword

Sometimes you want to skip an iteration only for a certain condition. You can do that using the `continue` keyword. Have a look at the following example:

```
for (var i = 0; i < 5; i++) {
  if (i == 2) {
    continue;
  }
  print(i);
}
```

This example is similar to the last one, but this time, when `i` is `2`, the `continue` keyword will tell the `for` loop to immediately go on to the next iteration. The rest of the code in the block won't run on this iteration.

This is what you'll see:

```
0
1
3
4
```

No `2` here!

More For Loops

There are two more kinds of `for` loops that you'll learn about later:

- `for-in` loops
- `for-each` loops

You'll study `for-in` loops in Chapter 12, "Lists", and `for-each` loops in the "Anonymous Functions" chapter of *Dart Apprentice: Beyond the Basics*.

Exercise

- Write a `for` loop starting at `1` and ending with `10` inclusive.
- Print the square of each number.

Challenges

Before moving on, here are some challenges to test your knowledge of loops. It's best if you try to solve them yourself, but solutions are available in the `challenge` folder if you get stuck.

Challenge 1: Next Power of Two

Given a number, determine the next power of two above or equal to that number. Powers of two are the numbers in the sequence of 2^1, 2^2, 2^3, and so on. You may also recognize the series as 1, 2, 4, 8, 16, 32, 64...

Challenge 2: Fibonacci

Calculate the *n*th Fibonacci number. The Fibonacci sequence starts with 1, then 1 again, and then all subsequent numbers in the sequence are simply the previous two values in the sequence added together (1, 1, 2, 3, 5, 8...). You can get a refresher here: https://en.wikipedia.org/wiki/Fibonacci_number

Challenge 3: How Many Times?

In the following `for` loop, what will be the value of `sum`, and how many iterations will happen?

```
var sum = 0;
for (var i = 0; i <= 5; i++) {
  sum += i;
}
```

Challenge 4: The Final Countdown

Print a countdown from 10 to 0.

Challenge 5: Print a Sequence

Print the sequence `0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0`.

Key Points

- `while` loops perform a certain task repeatedly as long as a condition is true.
- `do-while` loops always execute the loop at least once.
- `for` loops allow you to perform a loop a set number of times.
- The `break` statement lets you break out of a loop.
- The `continue` statement ends the current iteration of a loop and begins the next iteration.

7 Functions

Written by Jonathan Sande

Each week, there are tasks that you repeat over and over: eat breakfast, brush your teeth, write your name, read books about Dart, and so on. Each of those tasks can be divided up into smaller tasks. Brushing your teeth, for example, includes putting toothpaste on the brush, brushing each tooth and rinsing your mouth out with water.

The same idea exists in computer programming. A **function** is one small task, or sometimes a collection of several related tasks, that you can use in conjunction with other functions to accomplish a larger task. In this chapter, you'll learn how to write functions in Dart.

Function Basics

You can think of functions like machines. They take something you provide to them, the input, and produce something different, the output.

There are many examples of this in daily life. With an apple juicer, you put in apples and you get out apple juice. The input is apples; the output is juice. A dishwasher is another example. The input is dirty dishes, and the output is clean dishes. Blenders, coffee makers, microwaves and ovens are all like real-world functions that accept an input and produce an output.

Don't Repeat Yourself

Assume you have a small, useful piece of code that you've repeated in multiple places throughout your program:

```
// one place
if (fruit == 'banana') {
  peelBanana();
  eatBanana();
}

// another place
if (fruit == 'banana') {
  peelBanana();
  eatBanana();
}

// some other place
if (fruit == 'banana') {
  peelBanana();
  eatBanana();
}
```

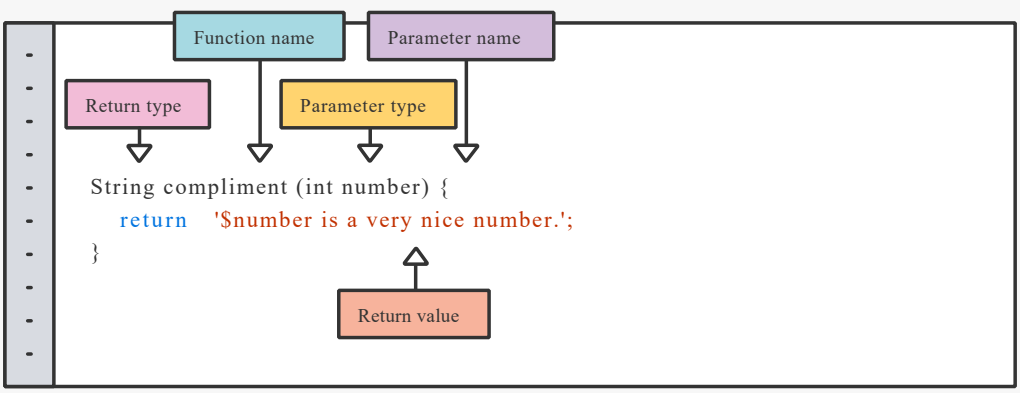
Now, that code works rather well, but repeating that code in multiple spots presents at least two problems. The first problem is that you're duplicating effort by having this code in multiple places in your program. The second and more troubling problem is that if you need to change the logic in that bit of code later on, you'll have to track down all of those instances of the code and change them in the same way. It's likely that you'll make a mistake somewhere, or even miss changing one of the instances because you didn't see it.

Over time, this problem has led to some sound advice for writing clean code: **don't repeat yourself**, abbreviated as **DRY**. This term was originally coined in the book *The Pragmatic Programmer* by Andrew Hunt and David Thomas. Writing DRY code will prevent many bugs from creeping into your programs.

Functions are one of the main solutions to the duplication problem in the example above. Instead of repeating blocks of code in multiple places, you can simply package that code into a function and call that function from wherever you need to.

Anatomy of a Dart Function

In Dart, a function consists of a return type, a name, a parameter list in parentheses and a body enclosed in braces.



Here is a summary of the labeled parts of the function:

- **Return type**: This comes first. It tells you immediately what the type will be of the function output. This particular function will return a `String`, but your functions can return any type you like. If the function won't return anything, that is, if it performs some work but doesn't produce an output value, you can use `void` as the return type.
- **Function name**: You can name functions almost anything you like, but you should follow the `lowerCamelCase` naming convention. You'll learn a few more naming conventions a little later in this chapter.
- **Parameters**: Parameters are the input to the function. They go inside the parentheses after the function name. This example has only one parameter, but if you had more than one, you would separate them with commas. For each parameter, you write the type first, followed by the name. Just as with variable names, you should use `lowerCamelCase` for your parameter names.
- **Return value**: This is the function's output, and it should match the return type. In the example above, the function returns a `String` value by using the `return` keyword. If the return type is `void`, though, then you don't return anything.

The return type, function name and parameters are collectively known as the **function signature**:

```
String compliment(int number)
```

The code between the braces is known as the **function body**:

```
{
  return '$number is a very nice number.';
}
```

This is what the function above looks like in the context of a program:

```
void main() {
  const input = 12;
  final output = compliment(input);
  print(output);
}

String compliment(int number) {
  return '$number is a very nice number.';
}
```

What have we here? Not one function, but two? Yes, `main` is also a function, and one you've seen many times already. It's the function that every Dart program starts with. Since `main` doesn't return a value, the return type of `main` must be `void`. Although `main` can take parameters, there aren't any in this case, so there's only a pair of empty parentheses that follow the function name.

Notice that the `compliment` function is *outside* of `main`. Dart supports **top-level functions**, which are functions that aren't inside a class or another function. Conversely, you may nest one function inside another. And when a function is inside a class, it's called a **method**, which you'll learn more about in Chapter 8, "Classes".

You call a function by writing its name and providing the **argument**, which is the value you provide inside the parentheses as the parameter to the function. In this case, you're calling the `compliment` function and passing in an argument of `12`. Run the code now and you'll see the following result:

```
12 is a very nice number.
```

Indeed, twelve *is* a nice number. It's the largest one-syllable number in English.

> **Note**: It's easy to get the words *parameter* and *argument* mixed up. A **parameter** is the name and type that you define as an input for your function. An **argument**, on the other hand, is the actual value that you pass in. A parameter is abstract, while an argument is concrete.

Parameters

Parameters are incredibly flexible in Dart, so they deserve their own section.

Using Multiple Parameters

In a Dart function, you can use any number of parameters. If you have more than one parameter for your function, simply separate them with commas. Here's a function with two parameters:

```
void helloPersonAndPet(String person, String pet) {
  print('Hello, $person, and your furry friend, $pet!');
}
```

Parameters like the ones above are called **positional parameters** because you have to supply the arguments in the same order that you defined the parameters when you wrote the function. If you call the function with the parameters in the wrong order, you'll get something obviously wrong:

```
nelloPersonAndPet('Fluffy', 'Chris');
// Hello, Fluffy, and your furry friend, Chris!
```

Making Parameters Optional

The function above was very nice, but it was a little rigid. For example, try the following:

```
helloPersonAndPet();
```

If you don't have exactly the right number of parameters, the compiler will complain to you:

```
2 positional argument(s) expected, but 0 found.
```

You defined `helloPersonAndPet` to take two arguments, but in this case, you didn't pass in any. It would be nice if the code could detect this, and just say, "Hello, you two!" if no names are provided. Thankfully, it's possible to have optional parameters in a Dart function.

Imagine you want a function that takes a first name, a last name and a title, and returns a single string with the various pieces of the person's name strung together:

```
String fullName(String first, String last, String title) {
  return '$title $first $last';
}
```

The thing is, not everyone has a title, or wants to use their title, so your function needs to treat the title as optional. To indicate that a parameter is optional, you surround the parameter with square brackets and add a question mark after the type, like so:

```
String fullName(String first, String last, [String? title]) {
  if (title != null) {
    return '$title $first $last';
  } else {
    return '$first $last';
  }
}
```

Here are a couple of points to note about the code above:

- Putting square brackets around `String? title` makes `title` optional. The `?` after `String` means it's a nullable type. If you don't pass in a value for `title`, then it will have the value of `null`, which means "no value". The updated code checks for `null` to decide how to format the return string.
- Optional parameters must go at the end of the parameter list. For example, you couldn't put `[String? title]` before `first` or `last` in the example above. The only exception to that is if you were to make all of the parameters optional, in which case you would surround the entire parameter list with the square brackets.

Write these two examples to test your code out:

```
print(fullName('Ray', 'Wenderlich'));
print(fullName('Albert', 'Einstein', 'Professor'));
```

Run that now and you'll see the following:

```
Ray Wenderlich
Professor Albert Einstein
```

The function correctly handles the optional title.

> **Note**: Technically speaking, the question mark in `String?` is not written *after* the type; it's an integral part *of* the type, that is, the nullable `String?` type. More on this in Chapter 11, "Nullability".

Providing Default Values

In the example above, you saw that the default value for an optional parameter was `null`. This isn't always the best value for a default, though. That's why Dart also gives you the power to change the default value of any parameter in your function by using the assignment operator.

Take a look at this example:

```
bool withinTolerance(int value, [int min = 0, int max = 10]) {
  return min <= value && value <= max;
}
```

There are three parameters here, two of which are optional: `min` and `max`. If you don't specify a value for them, then `min` will be `0` and `max` will be `10`.

Here are some specific examples to illustrate that:

```
withinTolerance(5)  // true
withinTolerance(15) // false
```

Since `5` is between `0` and `10`, this evaluates to `true`; but since `15` is greater than the default max of `10`, it evaluates to `false`.

If you want to specify values other than the defaults, you can do that as well:

```
withinTolerance(9, 7, 11) // true
```

Since `9` is between `7` and `11`, the function returns `true`.

Look at that function call again: `withinTolerance(9, 7, 11)`. Imagine that you're reading through your code for the first time in a month. What do those three numbers even mean? If you've got a good memory, you might recall that one of them is `value`, but which one? The first one? Or was it the second one? Or maybe it was the last one.

If that wasn't bad enough, the following function call also returns `true`:

```
withinTolerance(9, 7) // true
```

Since the function uses positional parameters, the provided arguments must follow the order you defined the parameters. That means `value` is `9`, `min` is `7` and `max` has the default of `10`. But who could ever remember that?

Of course, you could just **Command+click** the function name on a Mac, or **Control+click** on a PC, to go to the definition and remind yourself of what the parameters meant. But the point is that this code is extremely hard to read. If only there were a better way!

Well, now that you mention it...

Naming Parameters

Dart allows you to use **named parameters** to make the meaning of the parameters more clear in function calls.

To create a named parameter, surround it with curly braces instead of square brackets. Here's the same function as above, but using named parameters instead:

```
bool withinTolerance(int value, {int min = 0, int max = 10}) {
  return min <= value && value <= max;
}
```

Note the following:

- `min` and `max` are surrounded by braces, which means you *must* use the parameter names when you provide their argument values to the function.
- Like square brackets, curly braces make the parameters inside optional. Since `value` isn't inside the braces, though, it's still required.

To provide an argument, you use the parameter name, followed by a colon and then the argument value. Here is how you call the function now:

```
withinTolerance(9, min: 7, max: 11) // true
```

That's a lot clearer, isn't it? The names `min` and `max` make it obvious where the tolerance limits are now.

An additional benefit of named parameters is that you don't have to use them in the exact order in which they were defined. These are both equivalent ways to call the function:

```
withinTolerance(9, min: 7, max: 11) // true
withinTolerance(9, max: 11, min: 7) // true
```

They can also go before or after the positional parameter:

```
withinTolerance(min: 7, max: 11, 9) // true
withinTolerance(min: 7, 9, max: 11) // true
```

And since named parameters are optional, that means the following function calls are also valid:

```
withinTolerance(5)           // true
withinTolerance(15)          // false

withinTolerance(5, min: 7)   // false
withinTolerance(15, max: 20) // true
```

In the first two lines, since `min` is `0` and `max` is `10` by default, values of `5` and `15` evaluate to `true` and `false` respectively. In the last two lines, the `min` and `max` defaults were changed, which also changed the outcomes of the evaluations.

Making Named Parameters Required

You might like to make `value` a named parameter as well. That way you could call the function like so:

```
withinTolerance(value: 9, min: 7, max: 11)
```

However, this brings up a problem. Named parameters are optional by default, but `value` can't be optional. If it were, someone might try to use your function like this:

```
withinTolerance()
```

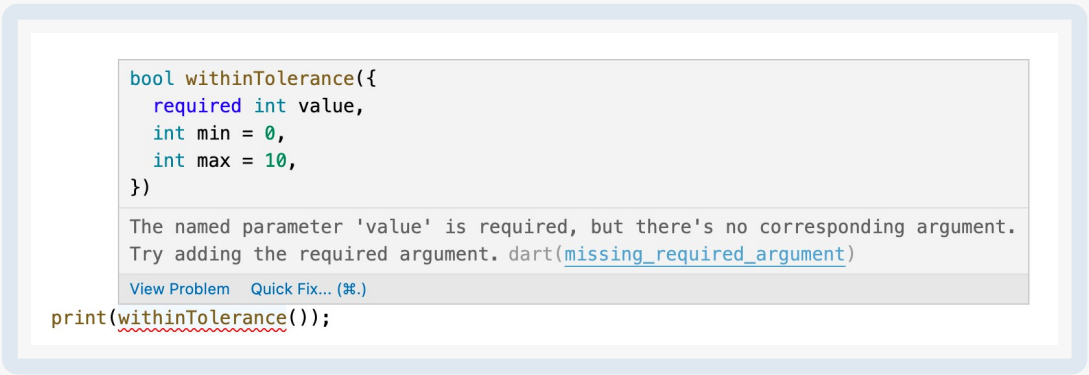
Should that return `true` or `false`? It doesn't make sense to return anything if you don't give the function a value. This is just a bug waiting to happen.

What you want is to make `value` required instead of optional, while still keeping it as a named parameter. You can achieve this by including `value` inside the curly braces and adding the `required` keyword in front:

```
bool withinTolerance({
  required int value,
  int min = 0,
  int max = 10,
}) {
  return min <= value && value <= max;
}
```

Since the function signature was getting a little long, adding a comma after the last parameter lets the IDE format it vertically. You still remember how to auto-format in VS Code, right? That's **Shift+Option+F** on a Mac or **Shift+Alt+F** on a PC.

With the `required` keyword in place, VS Code will warn you if you don't provide a value for `value` when you call the function:



Using named parameters makes your code more readable and is an important part of writing clean code when you have multiple inputs to a function. In the next section, you'll learn some more best practices for writing good functions.

Exercises

1. Write a function named `youAreWonderful`, with a string parameter called `name`. It should return a string using `name`, and say something like "You're wonderful, Bob."
2. Add another `int` parameter to that function called `numberPeople` so that the function returns something like "You're wonderful, Bob. 10 people think so."
3. Make both inputs named parameters. Make `name` required and set `numberPeople` to have a default of `30`.

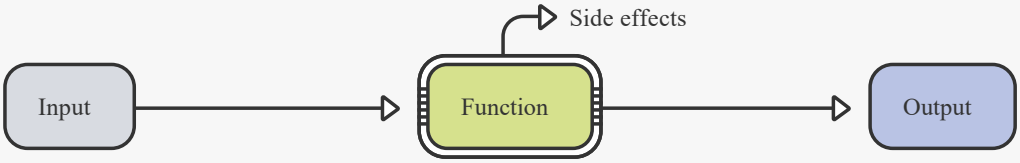
Writing Good Functions

People have been writing code for decades. Along the way, they've come up with some good practices to improve code quality and prevent errors. One of those practices is writing DRY code as you saw earlier. Here are a few more things to pay attention to as you learn about writing good functions.

Avoiding Side Effects

When you take medicine to cure a medical problem, but that medicine gives you a headache, that's known as a side effect. If you put some bread in a toaster to make toast, but the toaster burns your house down, that's also a side effect. Not all side effects are bad, though. If you take a business trip to Paris, you also get to see the Eiffel Tower. *Magnifique!*

When you write a function, you know what the inputs are: the parameters. You also know what the output is: the return value. Anything beyond that, that is, anything that affects the world outside of the function, is a side effect.



Have a look at this function:

```
void hello() {
  print('Hello!');
}
```

Printing something to the console is a side effect because it's affecting the world outside of the function. If you wanted to rewrite your function so that there were no side effects, you could write it like this:

```
String hello() {
  return 'Hello!';
}
```

Now, there's nothing inside the function body that affects the outside world. You'll have to write the string to the console somewhere outside of the function.

It's fine, and even necessary, for some functions to have side effects. But as a general rule,

functions without side effects are easier to deal with and reason about. You can rely on them to do exactly what you expect because they always return the same output for any given input. These kinds of functions are also called **pure functions**.

Here is another function with side effects to further illustrate the point:

```
var myPreciousData = 5782;

String anInnocentLookingFunction(String name) {
  myPreciousData = -1;
  return 'Hello, $name. Heh, heh, heh.';
}
```

Unless you took the time to study the code inside of `anInnocentLookingFunction`, you'd have no idea that calling this innocent-looking function would also change your precious data. That's because the function had an unknown side effect. This is also a good reminder about the dangers of using global variables like `myPreciousData`. You never know who might change it.

Make it your ambition to maximize your use of pure functions and minimize your use of functions with side effects.

Doing Only One Thing

Proponents of "clean code" recommend keeping your functions small and logically coherent. *Small* here means only a handful of lines of code. If a function is too big, or contains unrelated parts, consider breaking it into smaller functions.

Write your functions so that each one has only a single job to do. If you find yourself adding comments to describe different sections of a complex function, that's usually a good clue that you should break your function up into smaller functions. In clean coding, this is known as the **Single Responsibility Principle**. In addition to functions, this principle also applies to classes and libraries. But that's a topic for another chapter.

Choosing Good Names

You should always give your functions names that describe exactly what they do. If your code sounds like well-written prose, it'll be faster to read and easier to understand.

This naming advice applies to almost every programming language. However, there are a few additional naming conventions that Dart programmers like to follow. These are recommendations, not requirements, but keep them in mind as you code:

- Use noun phrases for pure functions; that is, ones without side effects. For example, use `averageTemperature` instead of `getAverageTemperature` and `studentNames` instead of `extractStudentNames`.
- Use verb phrases for functions with side effects. For example, `updateDatabase` or `printHello`.

- Also use verb phases if you want to emphasize that the function does a lot of work. For example, `calculateFibonacci` or `parseJson`.
- Don't repeat parameter names in the function name. For example, use `cube(int number)` instead of `cubeNumber(int number)`, or `printStudent(String name)` instead of `printStudentName(String name)`.

To be clear regarding the last point, it's not wrong to repeat the parameter name in the function name. However, there are a few reasons to prefer not repeating:

- All other things being equal, short names are easier to read than long names. Long function names are good when they're descriptive, but redundant information doesn't convey additional information. Since Dart supports named parameters, it's easy to make this information visible to the reader. Languages like C, C++ and Java don't have named parameters so it might be better to include the parameter name in the function name when programming in these languages.
- Longer function names make it more likely that the `dart format` tool will have to wrap a given line of code. Code that line-wraps is often harder to read than code that appears on a single line.

As a general rule, choose the function name that's the most readable.

Optional Types

Earlier you saw this function:

```
String compliment(int number) {
  return '$number is a very nice number.';
}
```

The return type is `String`, and the parameter type is `int`. Dart is an optionally-typed language, so it's possible to omit the types from your function declaration. In that case, the function would look like this:

```
compliment(number) {
  return '$number is a very nice number.';
}
```

Dart can infer that the return type here is `String`, but it has to fall back on `dynamic` for the unknown parameter type. The following function is the equivalent of what Dart sees:

```
String compliment(dynamic number) {
  return '$number is a very nice number.';
}
```

While it's permissible to omit return and parameter types, this book recommends that you include them at the very least for situations where Dart can't infer the type. As you learned in Chapter 3, "Types & Operations", there's a much greater advantage to writing Dart in a statically-typed way.

Arrow Functions

Dart has a special syntax for functions whose body is only one line. Consider the following function named `add` that adds two numbers together:

```
int add(int a, int b) {
  return a + b;
}
```

Since the body is only one line, you can convert it to the following form:

```
int add(int a, int b) => a + b;
```

You replaced the function's braces and body with an arrow (`=>`) and left off the `return` keyword. The return value is whatever the value of the expression is. Writing a function in this way is known as **arrow syntax** or **arrow notation**.

You can't use arrow syntax if the function body has more than one line, though, as in the following example:

```
void printTripled(int number) {
  final tripled = number * 3;
  print(tripled);
}
```

However, if you rewrote the body to fit on one line, arrow syntax would work:

```
void printTripled(int number) => print(number * 3);
```

You should now have a basic understanding of Dart functions. The functions you studied in this chapter were all **named functions**. In *Dart Apprentice: Beyond the Basics*, you'll learn about **anonymous functions**, which will take your skills to a whole new level.

Challenges

Before moving on, here are some challenges to test your knowledge of functions. It's best if you try to solve them yourself, but solutions are available in the **challenge** folder for this chapter if you get stuck.

Challenge 1: Circular Area

Write a function that tells you the area of a circle based on some input radius. If you recall from geometry class, you can find the area of a circle by multiplying pi times the radius squared.

Challenge 2: Prime Time

Write a function that checks if a number is prime.

First, write a function with the following signature to determine if one number is divisible by another:

```
bool isNumberDivisible(int number, int divisor)
```

The modulo operator `%` will help with that.

Then, write the function that returns `true` if prime and `false` otherwise:

```
bool isPrime(int number)
```

A number is prime if it's only divisible by 1 and itself. Loop through the numbers from 1 to the number and find the number's divisors. If it has any divisors other than 1 and itself, it's not prime.

Check the following cases:

```
isPrime(6); // false
isPrime(13); // true
isPrime(8893); // true
```

Here are a few more hints:

- Numbers less than zero are not considered prime.
- Use a `for` loop to look for divisors. You can start at 2 and if you end before the number, return false.
- If you're clever, you can loop from 2 until you reach the square root of the number. Add the following import to the top of the file to access the `sqrt` function:

```
import 'dart:math';
```

Key Points

- Functions package related blocks of code into reusable units.
- A function signature includes the return type, name and parameters. The function body is the code between the braces.
- Parameters can be positional or named, and required or optional.
- Side effects are anything besides the return value that change the world outside of the function body.
- To write clean code, use functions that are short and only do one thing.

Where to Go From Here?

This chapter spoke briefly about the Single Responsibility Principle and other clean coding principles. Do a search for **SOLID principles** to learn even more. It'll be time well spent.

Some aspects of Dart are hard-and-fast rules. Other aspects, like how to name variables and functions, are common practices, and while not required, they do make it easier to read the code and share it with others. Improve your understanding of these conventions by reading *Effective Dart* in the dart.dev guides.

8 Classes

Written by Jonathan Sande

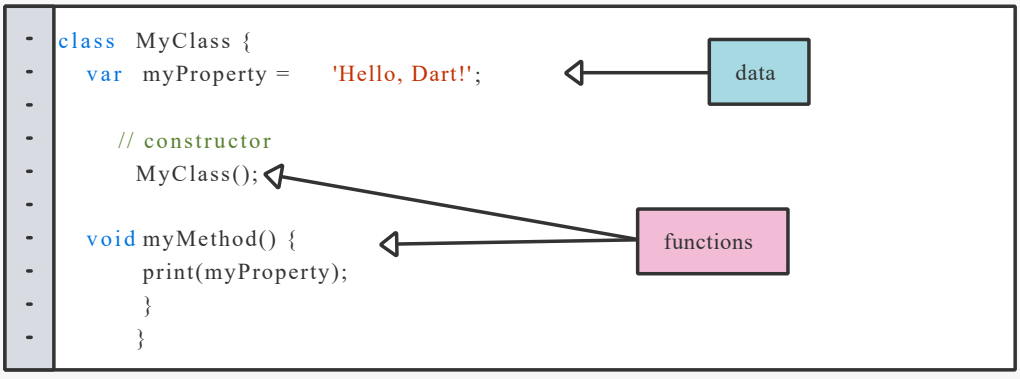
So far in this book, you've used built-in types such as `int`, `String` and `bool`. You've also seen one way to make custom types using `enum`. In this chapter, you'll learn a more flexible way to create types by using **classes**.

> **Note**: Because there's quite a bit to learn about classes and object-oriented programming (OOP) in Dart, you'll come back to the subject again in the following chapters as well as in the book *Dart Apprentice: Beyond the Basics*.

Classes are like architectural blueprints that tell the system how to make an **object**, where an object is the actual data stored in the computer's memory. If a class is the blueprint, you could say the object is like the house the blueprint represents. For example, the `String` class describes its data as a collection of UTF-16 code units, but a `String` object is something concrete like `'Hello, Dart!'`.

All values in Dart are objects that are built from a class. This includes the values of basic types like `int`, `double` and `bool`. That's different from other languages like Java, where basic types are primitive. For example, if you have `x = 10` in Java, the value of `x` is `10` itself. But Dart doesn't have primitive types. Even for a simple `int`, the value is an object that wraps the integer. You'll learn more about this concept later.

Classes are a core component of object-oriented programming. They're used to combine data *and* functions inside a single structure.



The functions exist to transform the data. Functions inside a class are known as **methods**, whereas **constructors** are special methods you use to create objects from the class. You'll learn about properties and methods in this chapter and about constructors in Chapter 9, "Constructors".

It's time to get your hands dirty. Working with classes is far more instructive than reading about them!

Defining a Class

To start creating types, you'll make a simple `User` class with `id` and `name` properties. This is just the kind of class you're highly likely to create in the future for an app that requires users to log in.

Write the following simple class at the top level of your Dart file. Your class should be outside of the `main` function, either above or below it.

```
class User {
  int id = 0;
  String name = '';
}
```

This creates a class named `User`. It has two properties: `id` is an `int` with a default value of `0`, and `name` is a `String` with the default value of an empty string.

The member variables of a class are generally called **fields**. But when the fields are public and visible to the outside world, you can also call them **properties**. The Encapsulation section below will show you how to simultaneously use public properties and private fields.

> **Note**: Depending on the situation, `null` may be a better default than `0` or `''`. But because nullable types and null safety won't be fully covered until Chapter 11, "Nullability", this chapter will simply use reasonable defaults for properties.

Creating an Object From a Class

As mentioned above, the value you create from a class is called an object. Another name for an object is **instance**, so creating an object is sometimes called **instantiating a class**.

Writing the `User` class as you did above simply created the blueprint; a `User` object doesn't exist yet. You can create the object by calling the class name as you would call a function. Add the following line inside the `main` function:

```
final user = User();
```

This creates an instance of your `User` class and stores that instance, or object, in `user`.

Notice the empty parentheses after `User`. It looks like you're calling a function without any parameters. In fact, you're calling a type of function called a **constructor method**. You'll learn a lot more about them in the next chapter. Right now, simply understand that using your class in this way creates an instance of your class.

The Optional Keyword New

Before version 2.0 of Dart came out, you had to use the `new` keyword to create an object from a class. At that time, creating a new instance of a class would have looked like this:

```
final user = new User();
```

This still works, but the `new` keyword is completely optional now, so it's better just to leave it off. Why clutter your code with unnecessary words, right?

You'll still come across `new` from time to time in the documentation or legacy code, but at least now it won't confuse you. If you meet it, just delete it.

Assigning Values to Properties

Now that you have an instance of `User` stored in `user`, you can assign new values to this object's properties using **dot notation**. To access the `name` property, type `user` *dot* `name` and then give it a value:

```
user.name = 'Ray';
```

Now, set the ID similarly:

```
user.id = 42;
```

Your code should look like the following:

```
void main() {
  final user = User();
  user.name = 'Ray';
  user.id = 42;
}

class User {
  int id = 0;
  String name = '';
}
```

You'll notice that you have both a function and a class together. Dart allows you to put multiple classes, top-level functions and even top-level variables together in the same file. Their order in the file isn't important. `User` is located below `main` here, but if you put it above `main`, that's fine as well.

You've defined a `User` data type with a class, created an object from it and assigned values to its parameters. Run the code now, though, and you won't see anything special happen. The following section will show you how to display data from an object.

Printing an Object

You can print any object in Dart. But if you try to print `user` now, you don't get quite what you hoped for. Add the following line at the bottom of the `main` function and run the code:

```
print(user);
```

Here's what you get:

```
Instance of 'User'
```

Hmm, you were likely expecting something about Ray and the ID. What gives?

Except for `Null`, all classes in Dart derive from `Object`, which has a `toString` method. In this case, your object doesn't tell Dart how to write its internal data when you call `toString` on it, so Dart gives you this generic, default output instead. But you can override the `Object` class's version of `toString` by writing your implementation of `toString` and thus customize how your object will print out.

Add the following method to the `User` class:

```
@override
String toString() {
  return 'User(id: $id, name: $name)';
}
```

Words that start with `@` are called **annotations**. Including them is optional and doesn't change how the code executes. But annotations *do* give the compiler more information so it can help you at compile time. Here, the `@override` annotation tells you and the compiler that `toString` is a method in `Object` that you want to override with your customized version. So if you accidentally wrote the `toString` method signature incorrectly, the compiler would warn you about it because of the `@override` annotation.

Because methods have access to the class properties, you simply use that data to output a more meaningful message when someone prints your object. Run the code now, and you'll see the following result:

```
User(id: 42, name: Ray)
```

That's far more useful!

> **Note**: Your `User` class only has a single method right now, but in classes with many methods, most programmers put the `toString` method at or near the bottom of the class instead of burying it in the middle somewhere. As you continue to add to `User`, keep `toString` at the bottom. Following conventions like this makes navigating and reading the code easier.

Adding Methods

Now that you've learned to override methods, you'll move on and add your methods to the `User` class. But before you do, there's a little background information that you should know.

Understanding Object Serialization

Organizing related data into a class is super useful, especially when you want to pass that data around as a unit within your app. One disadvantage, though, shows up when you're saving the object or sending it over the network. Files, databases and networks only know how to handle simple data types, such as numbers and strings. They don't know how to handle anything more complex, like your `User` data type.

Serialization is the process of converting a complex data object into a form you can store or transmit, usually a string. Once you've serialized the object, it's easy to save that data or transfer it across the network because everything from your app to the network and beyond knows how to deal with strings. Later, when you want to read that data back in, you can do so by way of **deserialization**, which is simply the process of converting a string back into an object of your data type.

You didn't realize it, but you serialized your `User` object in the `toString` method above. The code you wrote was good enough to get the job done, but you didn't follow any standardized format. You simply wrote it in a way that looked nice to the human eye. If you gave that string to someone else, though, they might have some difficulty understanding how to convert it back into a `User` object (deserialize it).

It turns out that serialization and deserialization are such common tasks that people have

devised standardized formats for serializing data. One of the most common is called **JSON**: JavaScript Object Notation. Despite the name, it's used far and wide outside the world of JavaScript.

JSON isn't difficult to learn, but this chapter will only show you enough JSON to serialize your `User` object into a more portable format. Check the *Where to Go From Here?* section at the end of the chapter to find out where you can learn more about this format.

Adding a JSON Serialization Method

You're going to add another method to your class now that will convert a `User` object to JSON format. It'll be like what you did in `toString`.

Add the following method to the `User` class, putting it above the `toString` method:

```
String toJson() {
  return '{"id":$id,"name":"$name"}';
}
```

Here are a few things to note:

- Because this is your custom method and you're not overriding a method that belongs to another class, you don't add the `@override` annotation.
- In Dart naming conventions, acronyms are treated as words. Thus, `toJson` is a better name than `toJSON`.
- There's nothing magic about serialization in this case. You simply used string interpolation to insert the property values in the correct locations in the JSON formatted string.
- In JSON, curly braces surround objects, commas separate properties, colons separate property names from property values, and double quotes surround strings. If a string needs to include a double-quote inside itself, you escape it with a backslash like so: `\"`
- JSON is similar to a Dart data type called `Map`. Dart even has built-in functions in the `dart:convert` library to serialize and deserialize JSON maps. And that's actually what most people use to serialize objects. But you haven't read Chapter 14, "Maps", so this example will be low-tech. You'll see a little preview of `Map` in the `fromJson` example in Chapter 9, "Constructors".

To test your new function, add the following line to the bottom of the main method:

```
print(user.toJson());
```

This code calls the custom `toJson` method on your `user` object using dot notation. The dot goes between the object name and method name, just like you saw earlier for accessing a property name.

Run the code, and you'll see the following:

```
{"id":42,"name":"Ray"}
```

It's very similar to what `toString` gave you, but this time it's in standard JSON format, so a computer on the other side of the world could easily convert that back into a Dart object.

Cascade Notation

When you created your `User` object above, you set its parameters like so:

```
final user = User();
user.name = 'Ray';
user.id = 42;
```

But Dart offers a cascade operator (`..`) that allows you to chain together multiple assignments on the same object without having to repeat the object name. The following code is equivalent:

```
final user = User()
  ..name = 'Ray'
  ..id = 42;
```

Note that the semicolon appears only on the last line.

Cascade notation isn't strictly necessary, but it makes your code a little tidier when you have to assign a long list of properties or repeatedly call a method that modifies your object.

Objects as References

Objects act as *references* to the instances of the class in memory. That means if you assign one object to another, the other object simply holds a reference to the same object in memory — not a new instance.

So if you have a class like this:

```
class MyClass {
  var myProperty = 1;
}
```

And you instantiate it like so:

```
final myObject = MyClass();
final anotherObject = myObject;
```

Then `myObject` and `anotherObject` both reference the *same* place in memory. Changing `myProperty` in either object will affect the other because they both reference the same instance:

```
print(myObject.myProperty);    // 1
anotherObject.myProperty = 2;
print(myObject.myProperty);    // 2
```

As you can see, changing the property's value on `anotherObject` also changed it in `myObject` because they are just two names for the same object.

> **Note**: If you want to make an *actual* copy of the class — not just a copy of its reference in memory but a whole *new* object with a deep copy of all the data it contains — you'll need to implement that mechanism by creating a method in your class that builds up a whole new object. Some call this method `copyWith` and allow the user to change selected properties while making the copy.

If that still doesn't make sense, you can look forward to sitting down by the fire and listening to the story of *The House on Wenderlich Way* in Chapter 12, "Lists", which will make this all clear or at least help you see it from a different perspective.

For now, though, there are a few more improvements you can make to the `User` class.

Encapsulation

One of the core tenets of object-oriented programming is known as **encapsulation**. This is the principle of hiding the internal data and logic in a class from the outside world. Doing so has a few benefits:

- The class controls its data, including who sees it and how it's modified.
- Isolated control means the logic is easier to reason about.
- Hidden data means better privacy.
- Encapsulation prevents unrelated classes from reaching in and modifying data, which can cause hard-to-find bugs.
- You can change the internal variable names and logic without breaking things in the outside world.

So those are the benefits. How do you accomplish encapsulation in Dart?

Hiding the Internals

You can make a variable private in Dart by prefixing the name with an underscore.

Create the following class in your project:

```
class Password {
  String _plainText = 'pass123';
}
```

The name `_value` begins with an underscore, so it's private. That means there's no way to access the user ID and name outside of the class, which makes your object kind of useless. You can solve this problem by adding a getter.

> **Note**: The above description isn't exactly accurate. In Dart, **private** means *library* private, not class private. A Dart library generally corresponds to a single file. That means other classes and functions in the same file have access to variable names that begin with an underscore. But these same variables are invisible from other libraries. You'll see this in action in Chapter 9, "Constructors".

Getters

A **getter** is a special method that returns the value of a private field variable. It's the public face of a private variable. If you've done any Java or C++ programming, you've probably seen getter methods with names like `getColor` or `getWidth`. Following this naming conversion, your Dart class would look like so:

```
class Password {
  String _plainText = 'pass123';

  String getPlainText() {
    return _plainText;
  }
}
```

You would access the password in `main` like so:

```
final myPassword = Password();
final text = myPassword.getPlainText();
```

But Dart discourages prefixing method names with `get` and instead has special syntax for getter properties.

A Dart getter uses the `get` keyword and returns a value. This gives you, as the class author, some control over how people access properties instead of allowing raw and unfettered access.

Replace your `Password` class with the updated form:

```
class Password {
  String _plainText = 'pass123';

  String get plainText => _plainText;
}
```

The `get` keyword provides a public-facing property name; in this case, `plainText`. The `get` method returns a value when you call the getter using its name. The getter method here is simply returning the `_plainText` field value.

Now that the property is exposed, you can use it like so:

```
final myPassword = Password();
final text = myPassword.plainText;
print(text); // pass123
```

Note that no `()` parentheses are needed after `plainText`. From the outside, this getter looks just like a normal property that you made with a field variable. The difference is that you can't set a value using a getter.

Getters Don't Set

Try adding the following line at the bottom of `main`:

```
myPassword.plainText = '123456';
```

Dart complains with an error:

```
There isn't a setter named 'plainText' in class 'Password'.
Try correcting the name to reference an existing setter or declare the setter.
```

You'll need to add a setter to change the internal value of `_plainText`. For now, just delete the line with the error in it.

Calculated Properties

You can also create getters that aren't backed by a dedicated field value but are calculated when called.

For example, you might not want to expose the plain text value of a password. Add the following getter to `Password`:

```
String get obfuscated {
  final length = _plainText.length;
  return '*' * length;
}
```

Here are some notes:

- There's no internal variable named `obfuscated` or `_obfuscated`. Rather, the return value of `obfuscated` is calculated when necessary.
- Getters can use arrow syntax or brace syntax just like regular methods. The example here uses brace syntax with a return statement because you're using more than a single line of code to calculate the return value.
- Multiplying a string by a number repeats the string that many times. In this case, you get a string of asterisks the same length as the password.

Replace the contents of `main` with the following code:

```
final myPassword = Password();
final text = myPassword.obfuscated;
print(text);
```

Run that, and you'll see the result below:

```
*******
```

This password is top secret!

Setters

Use a **setter** if you want to change the internal data in a class, Dart has a special `set` keyword for this to go along with `get`.

Add the following setter to your `Password` class:

```
set plainText(String text) => _plainText = text;
```

A setter starts with the `set` keyword. The `set` method takes a parameter, which you can use to set some value. In this case, you're setting the internal `_plainText` field.

You can now assign and retrieve the value of `_plainText` like this:

```
final myPassword = Password();
myPassword.plainText = r'Pa$$vv0Rd';
final text = myPassword.plainText;
print(text); // Pa$$vv0Rd
```

The second line sets the internal `_plainText` field, and the third line gets it.

You can see how this could give you extra control over what's assigned to your properties. For instance, you could use the setter to validate a good password.

Using Setters for Data Validation

Replace the `plainText` setter that you wrote above with the following version:

```
set plainText(String text) {
  if (text.length < 6) {
    print('Passwords must have 6 or more characters!');
    return;
  }
  _plainText = text;
}
```

If the user tries to set the value with a short password, there will be a warning and the internal field value won't change.

Test this out in `main`:

```
final shortPassword = Password();
shortPassword.plainText = 'aaa';
final result = shortPassword.plainText;
print(result);
```

Run this, and you'll see the following output:

```
Passwords must have 6 or more characters!
pass123
```

The password wasn't updated to `aaa`.

No Need to Overuse Getters And Setters

You don't always need to use getters and setters explicitly. If all you're doing is shadowing some internal field variable, you're better off just using a public variable.

For example, if you've written a class like this:

```
class Email {
  String _value = '';

  String get value => _value;
  set value(String value) => _value = value;
}
```

You might as well just simplify that to the following form:

```
class Email {
  String value = '';
}
```

Dart implicitly generates the needed getters and setters for you. That's quite a bit more readable and still works the same as if you had written your getters and setters:

```
final email = Email();
email.value = 'ray@example.com';
final emailString = email.value;
```

That's the beauty of how Dart handles class properties. You can change the internal implementation without the external world being any the wiser.

If you only want a getter but not a setter, just make the property `final` and set it in the constructor. You'll learn how to do that in Chapter 9, "Constructors".

Challenges

Before moving on, here's a challenge to test your knowledge of classes. It's best if you try to solve it yourself, but a solution is available with the supplementary materials for this book if you get stuck.

Challenge 1: Rectangles

- Create a class named `Rectangle` with properties for `_width` and `_height`.
- Add getters named `width` and `height`.
- Add setters for these properties that ensure you can't give negative values.
- Add a getter for a calculated property named `area` that returns the area of the rectangle.

Key Points

- Classes package data and functions inside a single structure.
- Variables in a class are called fields, and public fields or getter methods are called properties.
- Functions in a class are called methods.
- You can customize how an object is printed by overriding the `toString` method.
- Classes have getters and setters, which you can customize without affecting how the object is used.

Where to Go From Here?

This chapter touched briefly on JSON as a standard way to serialize objects. You'll certainly be using JSON in the future, so you can visit json.org to learn more about this format and why it's gained so much traction as a standard.

9 Constructors

Written by Jonathan Sande

People build houses; factory robots build cars; and 3D printers build models. In the programming world, **constructors** are methods that create, or *construct*, instances of a class. That is to say, constructors build new objects. Constructors have the same name as the class, and the implicit return type of the constructor method is also the same type as the class itself.

This chapter will teach you about the differences between the various types of constructor methods Dart provides for building classes, which include generative, named, forwarding and factory constructors.

Default Constructor

When you don't specify a constructor, Dart provides a default constructor that takes no parameters and just returns an instance of the class. For example, defining a class like this:

```
class Address {
  var value = '';
}
```

Is equivalent to writing it like this:

```
class Address {
  Address();
  var value = '';
}
```

Including the default `Address()` constructor is optional.

Sometimes you don't want the default constructor, though. You'd like to initialize the data in an object at the same time that you create the object. The next section shows how to do just that.

Custom Constructors

If you want to pass parameters to the constructor to modify how your class builds an object, you can. It's similar to how you wrote functions with parameters in Chapter 7, "Functions".

Like the default constructor above, the constructor name should be the same as the class name. This type of constructor is called a **generative constructor** because it directly generates an object of the same type.

You'll continue to build on the `User` class you wrote in Chapter 8, "Classes". Here's the code you had at the end of that chapter:

```
class User {
  int id = 0;
  String name = '';

  String toJson() {
    return '{"id":$id,"name":"$name"}';
  }

  @override
  String toString() {
    return 'User(id: $id, name: $name)';
  }
}
```

Copy that to your project below the `main` method before continuing on with the rest of the chapter.

Long-Form Constructor

In Dart, the convention is to put the constructor before the property variables. Add the following generative constructor method at the top of the class body:

```
User(int id, String name) {
  this.id = id;
  this.name = name;
}
```

This is known as a **long-form constructor**. You'll understand why it's considered "long-form" when you see the short-form constructor later.

Without the `toJson` and `toString` methods, this is what the class looks like:

```
class User {
  User(int id, String name) {
    this.id = id;
    this.name = name;
  }

  int id = 0;
  String name = '';

  // ...
}
```

`this` is a new keyword. What does it do?

The keyword `this` in the constructor body allows you to disambiguate which variable you're talking about. It means *this* object. So `this.name` refers to the object property called `name`, while `name` (without `this`) refers to the constructor parameter. Using the same name for the constructor parameters as the class properties is called **shadowing**. So the constructor above takes the `id` and `name` parameters and uses `this` to initialize the properties of the object.

Write the following code in `main` to create a `User` object by passing in some arguments:

```
final user = User(42, 'Ray');
print(user);
```

Once you've created the object, you can access its properties and other methods just as you did in the last chapter. However, you can't use the default constructor `User()` anymore since `id` and `name` are required positional parameters.

Short-Form Constructor

Dart also has a **short-form constructor** where you don't provide a function body, but you instead list the properties you want to initialize, prefixed with the `this` keyword. Arguments you send to the short form constructor are used to initialize the corresponding object properties.

Here's the long-form constructor you currently have:

```
User(int id, String name) {
  this.id = id;
  this.name = name;
}
```

Now replace that with the following short-form constructor:

```
User(this.id, this.name);
```

Dart infers the constructor parameter types of `int` and `String` from the properties themselves that are declared in the class body.

The class should now look like this:

```
class User {
  User(this.id, this.name);

  int id = 0;
  String name = '';

  // ...
}
```

Run the code again. You'll see the short-form constructor works just like the longer form you replaced, but it's just a little tidier now.

> **Note**: You could remove the default property values of `0` and `''` at this point since `id` and `name` are guaranteed to be initialized by the constructor parameters. However, there's an intermediate step in the next section where they'll still be useful. Keeping the default values a little longer will allow this chapter to postpone dealing with null safety until it can be covered more fully in Chapter 11, "Nullability".

Named Constructors

Dart also has a second type of generative constructor called a **named constructor**, which you create by adding an identifier to the class name. It takes the following pattern:

```
ClassName.identifierName()
```

From here on, this chapter will refer to a constructor without the identifier as an **unnamed constructor**:

```
// unnamed constructor
ClassName()

// named constructor
ClassName.identifierName()
```

Why would you want a named constructor instead of the nice, tidy default one? Well, sometimes you have some common cases that you want to provide a convenience constructor for. Or maybe you have some special edge cases for constructing certain classes that need a slightly different approach.

Say, for example, that you want to have an anonymous user with a preset ID and name. You can do that by creating a named constructor. Add the following named constructor below the short-form constructor:

```
User.anonymous() {
  id = 0;
  name = 'anonymous';
}
```

The identifier, or named part, of the constructor is `.anonymous`. Named constructors may have parameters, but in this case, there are none. And since there aren't any parameter names to get confused with, you don't need to use `this.id` or `this.name`. Rather, you just use the property variables `id` and `name` directly.

Call the named constructor in `main` like so:

```
final anonymousUser = User.anonymous();
print(anonymousUser);
```

Run that and you'll see the expected output:

```
User(id: 0, name: anonymous)
```

Note: Without default values for `id` and `name`, Dart would have complained that these variables weren't being initialized, even though `User.anonymous` does in fact initialize them in the constructor body. You could solve the problem by using the `late` keyword, but that's a topic for Chapter 11, "Nullability" — hence the default values here.

Forwarding Constructors

In the named constructor example above, you set the class properties directly in the constructor body. However, this doesn't follow the DRY principle you learned earlier. You're repeating yourself by having two different locations where you can set the properties. It's not a huge deal, but imagine that you have five different constructors instead of two. It

would be easy to forget to update all five if you had to make a change. And if the constructor logic were complicated, it would be easy to make a mistake.

One way to solve this issue is by calling the main constructor from the named constructor. This is called **forwarding** or **redirecting**. To do that, you use the keyword `this` again.

Delete the `anonymous` named constructor that you created above and replace it with the following:

```
User.anonymous() : this(0, 'anonymous');
```

This time there's no constructor body, but instead, you follow the name with a colon and then forward the properties to the unnamed constructor. The forwarding syntax replaces `User` with `this`.

Also, now that you've moved property initialization from the constructor body to the parameter list, Dart is finally convinced that `id` and `name` are guaranteed to be initialized.

Replace these two lines:

```
int id = 0;
String name = '';
```

with the following:

```
int id;
String name;
```

No complaints from Dart.

You call the named constructor exactly like you did before:

```
final anonymousUser = User.anonymous();
```

The results are the same as well.

Optional and Named Parameters

Everything you learned about function parameters in Chapter 7, "Functions", also applies to constructor method parameters. That means you can make parameters optional using square brackets:

```
MyClass([this.myProperty]);
```

Or you can make them optional and named using curly braces:

```
MyClass({this.myProperty});
```

Or named and required using curly braces and the `required` keyword:

```
MyClass({required this.myProperty});
```

Adding Named Parameters for User

Earlier when you instantiated a `User` object, you wrote this:

```
final user = User(42, 'Ray');
```

For someone not familiar with your `User` class, they might think `42` is Ray's age, or his password, or how many pet cats he has. Using named parameters here would help a lot with readability.

Refactor the unnamed constructor in `User` by adding braces around the parameters. Since that makes the parameters optional, you could use the `required` keyword, but this time simply give them default values:

```
User({this.id = 0, this.name = 'anonymous'});
```

The compiler will complain about this because that change also requires refactoring the `anonymous` named constructor to use the parameter names in the redirect.

However, since the parameter defaults are now just what the `anonymous` constructor was doing before, you can delete the `anonymous` constructor and replace it with the following:

```
User.anonymous() : this();
```

Using the named constructor will now forward to the unnamed constructor with no arguments.

In `main`, the way to create an anonymous user is still the same, but now the way to create Ray's user object is like so:

```
final user = User(id: 42, name: 'Ray');
```

That's a lot more readable. They're not cats; it's clearly just an ID.

In case you've gotten lost, here's what things look like now:

```
void main() {
  final user = User(id: 42, name: 'Ray');
  print(user);
  final anonymousUser = User.anonymous();
  print(anonymousUser);
}

class User {
  // unnamed constructor
  User({this.id = 0, this.name = 'anonymous'});

  // named constructor
  User.anonymous() : this();

  int id;
  String name;

  // ...
}
```

Initializer Lists

You might have discovered a small problem that exists with your class as it's now written. Take a look at the following way that an unscrupulous person could use this class:

```
final vicki = User(id: 24, name: 'Vicki');
vicki.name = 'Nefarious Hacker';
print(vicki);
// User(id: 24, name: Nefarious Hacker)
```

If those statements were spread throughout the codebase instead of being in one place as they are here, someone printing the `vicki` user object and expecting a real name would get a surprise. "Nefarious Hacker" is definitely not what you'd expect. Once you've created the `User` object, you don't want anyone to mess with it.

But forget nefarious hackers; *you're* the one who's most likely to change a property and then forget you did it.

There are a couple of ways to solve this problem. You'll see one solution now and a fuller solution later.

Private Variables

As you learned in the previous chapter, Dart allows you to make variables private by adding an underscore `_` in front of their name.

Change the `id` property to `_id` and `name` to `_name`. Since these variables are used in several locations throughout your code, let VS Code help you out. Put your cursor on the variable name and press `F2`. Edit the variable name and press **Enter** to change all of the references at once.

Oh..., that actually renamed a few more things than you intended since it also renamed what was in the `main` function. But that's OK. Just delete everything inside the body of `main` for now.

There is still one problem left with the unnamed constructor in `User`:

```
User({this._id = 0, this._name = 'anonymous'});
```

The compiler gives you an error:

```
Named parameters can't start with an underscore.
```

Fix that by deleting the unnamed constructor and replacing it with the following:

```
User({int id = 0, String name = 'anonymous'})
      : _id = id,
        _name = name;
```

Do you see the colon that precedes `_id`? The comma-separated list that comes after it is called the **initializer list**. One use for this list is exactly what you've done here. Externally, the parameters have one name, while internally, you're using private variable names.

The initializer list is always executed before the body of the constructor if the body exists. You don't need a body for this constructor, but if you wanted to add one, it would look like this:

```
User({int id = 0, String name = 'anonymous'})
    : _id = id,
      _name = name {
  print('User name is $_name');
}
```

The constructor would initialize `_id` and `_name` before it ran the `print` statement inside the braces.

Why Aren't the Private Properties Private?

It turns out that your nefarious hacker can still access the "private" fields of `User`. Add the following two lines to `main` to see this in action:

```
final vicki = User(id: 24, name: 'Vicki');
vicki._name = 'Nefarious Hacker';
```

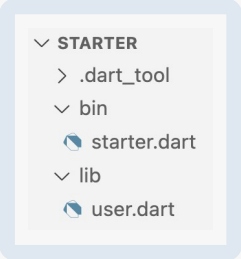
What's that all about? Well, using an underscore before a variable or method name makes it *library* private, not class private. For your purposes in this book, a library is simply a file. Since the `main` function and the `User` class are in the same file, nothing in `User` is hidden from `main`. To see private variables in action, you'll need to make another file so that you aren't using your class in the same file in which it's defined.

Create a new folder in the root of your project called **lib**, which is short for "library". This is the standard name and location where Dart expects to find the project files you want to import. To create the new folder in VS Code, you can click on the project folder in the Explorer panel and then press the **New Folder button**:

Next, create a new file in **lib** called **user.dart**. You can click on **lib** in the Explorer panel and then press the **New File button**:

Now move the entire `User` class over to **lib/user.dart**.

If you're using the starter project, your folder structure should look like this now:



Go back to **bin/starter.dart**, the one with the `main` function, and add the library import to the top of the class:

```
import 'package:starter/user.dart';
```

Here are some notes:

- If you aren't using the starter project, replace `starter` with whatever your project name is.
- There's no reference to the `lib` folder here. Dart knows that the `package` files with your project name are in the `lib` folder. A **package** is a collection of library files.

Now you'll notice that in the `main` function you no longer have access to `_name`:

```
vicki._name = 'Nefarious Hacker';
```

This produces an error:

```
The setter '_name' isn't defined for the type 'User'.
```

Great! Now it's no longer possible to change the properties after the object has been created. Delete or comment out that entire line:

```
// vicki._name = 'Nefarious Hacker';
```

Constant Constructors

You've already learned how to keep people from modifying the properties of a class by making them private. Another thing you can do is to make the properties **immutable**, that is, unchangeable. By using immutable properties, you don't even have to make them private.

Making Properties Immutable

There are two ways to mark a variable immutable in Dart: `final` and `const`. However, since the compiler won't know what the properties are until runtime, your only choice here is to use `final`.

In the `User` class in **lib/user.dart**, add the `final` keyword before both property declarations. Your code should look like this:

```
final String _name;
final int _id;
```

Adding `final` means that `_name` and `_id` can only be given a value once, that is, when the constructor is called. After the object has been created, those properties will be immutable. You should keep the `String` and `int` type annotations because removing them would cause the compiler to fall back to `dynamic`.

Making Classes Immutable

If the objects of a particular class can never change because all fields of the class are `final`, you can add `const` to the constructor to ensure that all instances of the class will be constants at compile time.

Since both the fields of your `User` class are now `final`, this class is a good candidate for a compile-time constant.

Replace both constructors with the following:

```
const User({int id = 0, String name = 'anonymous'})
    : _id = id,
      _name = name;

const User.anonymous() : this();
```

Note the `const` keyword in front of both constructors.

Now you can declare your `User` objects as compile-time constants like so:

```
const user = User(id: 42, name: 'Ray');
const anonymousUser = User.anonymous();
```

Benefits of Using Const

In addition to being immutable, another benefit of `const` variables is that they're **canonical instances**, which means no matter how many instances you create, as long as the properties used to create them are the same, Dart will only see a single instance. You could instantiate `User.anonymous()` a thousand times across your app without incurring the performance hit of having a thousand different objects.

> **Note**: Flutter uses this pattern frequently with its `const` widget classes in the user interface of your app. Since Flutter knows that the `const` widgets are immutable, it doesn't have to waste time recalculating and drawing the layout when it finds these widgets.

Make it your goal to use `const` objects and constructors as much as possible. It's a performance win!

Exercise

Given the following class:

```
class PhoneNumber {
  String value = '';
}
```

1. Make `value` a `final` variable, but not private.
2. Add a `const` constructor as the only way to initialize a `PhoneNumber` object.

Factory Constructors

All of the constructors that you've seen up until now have been generative constructors. Dart also provides another type of constructor called a **factory constructor**.

A factory constructor provides more flexibility in how you create your objects. A generative constructor can only create a new instance of the class itself. However, factory constructors can return *existing* instances of the class, or even subclasses of it. You'll learn about subclasses in *Dart Apprentice: Beyond the Basics*, Chapter 3, "Inheritance". This is useful when you want to hide the implementation details of a class from the code that uses it.

The factory constructor is basically a special method that starts with the `factory` keyword and returns an object of the class type. For example, you could add the following factory constructor to your `User` class:

```
factory User.ray() {
  return User(id: 42, name: 'Ray');
}
```

The factory method uses the generative constructor to create and return a new instance of `User`. You could also accomplish the same thing with a named constructor, though.

A more common example you'll see is using a factory constructor to make a `fromJson` method:

```
factory User.fromJson(Map<String, Object> json) {
  final userId = json['id'] as int;
  final userName = json['name'] as String;
  return User(id: userId, name: userName);
}
```

You would create a `User` object from the constructor like so:

```
final map = {'id': 10, 'name': 'Sandra'};
final sandra = User.fromJson(map);
```

You'll learn how the `Map` collection works in Chapter 14, "Maps". The thing to pay attention to now is that the factory constructor body allows you to perform some work before returning the new object, and you didn't exposing the details of that work to whoever is using the class.

For example, you could create a `User.fromJson` constructor with a named constructor like so:

```
User.fromJson(Map<String, Object> json)
    : _id = json['id'] as int,
      _name = json['name'] as String;
```

However, there isn't much else you can do with `id` and `name`. With a factory constructor, though, you could do all kinds of validation, error checking and even modification of the arguments before creating the object. This is actually highly desirable in the case here because if `'id'` or `'name'` didn't exist in the map, then your app would crash because you aren't handling `null`.

> **Note**: Using a factory constructor over a named constructor can also help to prevent breaking changes for subclasses of your class. That topic is a little beyond the scope of this chapter, but you can read https://stackoverflow.com/a/66117859 for a longer explanation.

You'll see a few more uses of the factory constructor in Chapter 10, "Static Members".

Constructor Comparisons

Since there are so many ways that constructors can vary, here's a brief comparison.

Constructors can be:

- Forwarding or non-forwarding.
- Named or unnamed.
- Generative or factory.
- Constant or not constant.
- With parameters or without.
- Short-form or long-form.
- Public or private.

Many of those options can vary independently of each other. For example, the following is a public, non-forwarding, unnamed, generative, const constructor with parameters.

```
const User(this.id, this.name);
```

Or here's a private, non-forwarding, named, generative, non-constant constructor without parameters:

```
DatabaseHelper._internal();
```

You'll learn one use for private constructors in the next chapter when you get to the section on singletons. See you there!

Challenges

Before moving on, here is a challenge to test your knowledge of constructors. It's best if you try to solve it yourself, but a solution is available if you get stuck. It's located with the supplementary materials for this book.

Challenge 1: Bert and Ernie

Create a `Student` class with final `firstName` and `lastName` string properties and a variable `grade` as an `int` property. Add a constructor to the class that initializes all the properties. Add a method to the class that nicely formats a `Student` for printing. Use the class to create students `bert` and `ernie` with grades of 95 and 85, respectively.

Key Points

- You create an object from a class by calling a constructor method.
- Generative constructors can be unnamed or named.
- Unnamed generative constructors have the same name as the class, while named generative constructors have an additional identifier after the class name.
- You can forward from one constructor to another by using the keyword `this`.
- Initializer lists allow you to initialize field variables.
- Adding `const` to a constructor allows you to create immutable, canonical instances of the class.
- Factory constructors allow you to hide the implementation details of how you provide the class instance.

10 Static Members

Written by Jonathan Sande

There's just one more thing to cover for your well-rounded foundation in Dart classes. That's the **static** keyword. No relationship to static electricity.

Putting `static` in front of a member variable or method causes the variable or method to belong to the *class* rather than the *instance*.

```
class SomeClass {
  static int myProperty = 0;
  static void myMethod() {
    print('Hello, Dart!');
  }
}
```

They work the same as top-level variables and functions but are wrapped in the class name.

You access them like so:

```
final value = SomeClass.myProperty;
SomeClass.myMethod();
```

In this case, you didn't have to instantiate an object to access `myProperty` or to call `myMethod`. Instead, you used the class name directly to get the value and call the method.

The following sections will cover a few common use cases for static members.

Static Variables

Static variables are often used for constants and in the singleton pattern.

> **Note**: Variables receive different names according to where they belong or are located. Because static variables belong to the class, they're called **class variables**. Non-static member variables are called **instance variables** because they only have a value after an object is instantiated. Variables within a method are called **local variables**. And top-level variables outside of a class are called **global variables**.

Constants

Many classes store useful values that never change. Some examples of these class constants include the following:

- Mathematical constants like pi or *e*.
- Predefined colors.
- Default font sizes.
- Hard-coded URLs.
- Names for difficult-to-remember Unicode values or invisible characters.

Default Font Size Example

You can define class constants by combining the `static` and `const` keywords. For example, add the following class to your project below `main`:

```
class TextStyle {
  static const _defaultFontSize = 17.0;

  TextStyle({this.fontSize = _defaultFontSize});
  final double fontSize;
}
```

The constant is on the second line. Although you could have directly written `this.fontSize = 17.0` right in the constructor, using the name `_defaultFontSize` makes the meaning more clear. Also, if you used the default font size at several different places in your class, you'd only need to change a single line if you ever decided to update the default size.

Color Codes Example

In the previous example, the constant was private, but it can also be useful to have public constants. You'll see this with Flutter's `Colors` class. You'll make a simplified version here.

Add the following class to your project:

```
class Colors {
  static const int red = 0xFFD13F13;
  static const int purple = 0xFF8107D9;
  static const int blue = 0xFF1432C9;
}
```

These are the hex values for shades of red, purple and blue. The `0x` prefix tells Dart that

you're using hexadecimal values for the integers. The eight hex characters after `0x` follow the `AARRGGBB` pattern, where `AA` is the amount of alpha or transparency, `RR` is the amount of red, `GG` is the amount of green and `BB` is the amount of blue.

Then, in `main`, add the following line:

```
final backgroundColor = Colors.purple;
```

This is a much more readable way of describing a color than scattering the hex values around your app.

Singleton Pattern

Another use of static variables is to create a **singleton** class. Singletons are a common design pattern with only one instance of an object. Although some people debate their benefits, they make certain tasks more convenient.

It's easy to create a singleton in Dart. You wouldn't want `User` to be a singleton because you'd likely have many distinct users requiring many distinct instances of `User`. But you might want to create a singleton class as a database helper to ensure you don't open multiple connections to the database.

Here's what a basic singleton class would look like:

```
class MySingleton {
  MySingleton._();
  static final MySingleton instance = MySingleton._();
}
```

The `MySingleton._()` part is a private, named constructor. Some people like to call it `_internal` to emphasize that it can't be called from the outside. The underscore makes it impossible to instantiate the class normally. But the static property, which is only initialized once, provides a reference to the instantiated object.

> **Note**: Static fields and top-level variables — global variables outside of a class — are **lazily initialized**. That means Dart doesn't calculate and assign their values until you use them first.

You would access the singleton like so:

```
final mySingleton = MySingleton.instance;
```

Because factory constructors don't need to return new instances of an object, you can also implement the singleton pattern with a factory constructor:

```
class MySingleton {
  MySingleton._();
  static final MySingleton _instance = MySingleton._();
  factory MySingleton() => _instance;
}
```

The advantage here is that you can hide the fact that it's a singleton from whoever uses it:

```
final mySingleton = MySingleton();
```

From the outside, this looks exactly like a normal object. This allows you to change it back into a generative constructor later without affecting the code in other parts of your project.

The past two sections have been about static variables. Next, you'll look at static methods.

Static Methods

You can do a few interesting things with static methods.

Utility Methods

One use for a static method is to create a utility or helper method associated with the class but not with any particular instance.

Add the following `Math` class to your project:

```
class Math {
  static num max(num a, num b) {
    return (a > b) ? a : b;
  }

  static num min(num a, num b) {
    return (a < b) ? a : b;
  }
}
```

The `static` keyword goes at the beginning of the method signature. Static methods can't access instance variables; they can only use static constants or the parameters that are passed in.

Use your static methods in `main` like so:

```
print(Math.max(2, 3));  // 3
print(Math.min(2, 3));  // 2
```

Just Because You Can, Doesn't Mean You Should

In other languages, some developers like to group related static utility methods in classes to keep them organized. But in Dart, it's usually better to put these utility methods in their own file as top-level functions. You can then import that file as a library wherever you need the utility methods contained within. The description below will show you how to refactor your `Math` class into a library of top-level functions.

Add a **lib** folder to the root of your project just like you did in Chapter 9, "Constructors". Then create a file named **math.dart** inside **lib**. Paste in the following top-level functions:

```
num max(num a, num b) {
  return (a > b) ? a : b;
}

num min(num a, num b) {
  return (a < b) ? a : b;
}
```

Because they're top-level functions rather than class methods, there's no need to use the `static` keyword.

Now, go back to the file with your `main` function. Add the following import at the top of the file:

```
import 'package:starter/math.dart';
```

Again, if you aren't using the starter project, change `starter` to your project name.

Then, replace the contents of `main` with the following code:

```
print(max(2, 3)); // 3
print(min(2, 3)); // 2
```

Your functions no longer reference an unnecessary `Math` wrapper class.

To prevent naming collisions, use the `as` keyword after the import. Replace the contents

of the file your `main` method is in with the following:

```
import 'package:starter/math.dart' as math;

void main() {
  print(math.max(2, 3)); // 3
  print(math.min(2, 3)); // 2
}
```

The name you put after `as` can be anything. In this case, you used the word `math`. Then, you used `math.max` and `math.min` to reference the functions. This technique is useful when two libraries have functions with the same name. The standard `dart:math` library, for example, also has a `max` and `min` function.

Creating New Objects

You can also use static methods to create new instances of a class based on some input passed in. For example, you could use a static method to achieve the same result as in the previous chapter with the `fromJson` factory constructor.

Here's the `fromJson` factory constructor from the `User` class in Chapter 9, "Constructors":

```
factory User.fromJson(Map<String, Object> json) {
  final userId = json['id'] as int;
  final userName = json['name'] as String;
  return User(id: userId, name: userName);
}
```

And here's the static method version:

```
static User fromJson(Map<String, Object> json) {
  final userId = json['id'] as int;
  final userName = json['name'] as String;
  return User(id: userId, name: userName);
}
```

From the outside as well, you use it as you did with the factory version:

```
final map = {'id': 10, 'name': 'Sandra'};
final sandra = User.fromJson(map);
```

This all shows that there are often multiple ways of accomplishing the same thing.

Comparing Static Methods With Factory Constructors

Factory constructors are like static methods in many ways, but there are a few differences:

- A factory constructor can only return an instance of the class type or subtype, whereas a static method can return anything. For example, a static method can be asynchronous and return a `Future` — which you'll learn about in *Dart Apprentice: Beyond the Basics*, Chapter 12, "Futures" — but a factory constructor can't do this.
- A factory constructor can be unnamed, so from the caller's perspective, it looks exactly like calling a generative constructor. The singleton example above is an example of this. A static method, on the other hand, must have a name.
- A factory constructor can be `const` if it's a forwarding constructor, but a static method can't.

Challenges

Before moving on, here's a challenge to test your knowledge of static members. It's best if you try to solve it yourself, but a solution is available with the supplementary materials for this book if you get stuck.

Challenge 1: Spheres

Create a `Sphere` class with a `const` constructor that takes a `radius` as a named parameter. Add getters for the volume and surface area but none for the radius. Don't use the `dart:math` package but store your version of `pi` as a `static` constant. Use your class to find the volume and surface area of a sphere with a radius of `12`.

Key Points

- Adding the `static` keyword to a property or method makes it belong to the class rather than the instance.
- Static constants are useful for storing values that don't change.
- A singleton is a class with only one instance of an object.
- A utility method is a method that's associated with the class but not with any particular instance.
- Group top-level functions into their own library rather than wrapping a bunch of static methods in a utility class.
- Static methods can replace factory constructors but have a few subtle differences from factory constructors.

Where to Go From Here?

This chapter touched on concepts such as singletons and factories. These concepts are known collectively as **design patterns**. Although you don't need to know design patterns to code in Dart, understanding them will make you a better programmer. The most famous book on this topic is *Design Patterns* by "The Gang of Four", but there are many other excellent works on the subject. A simple online search for **software design patterns** will provide you a wealth of information.

11 Nullability

Written by Jonathan Sande

You know that game where you try to find the item that doesn't belong in a list? Here's one for you:

```
horse, camel, pig, cow, sheep, goat
```

Which one doesn't belong?

It's the third one, of course! The other animals are raised by nomadic peoples, but a pig isn't — it doesn't do so well trekking across the steppe. About now you're probably muttering to yourself why your answer was just as good — like, a sheep is the only animal with wool, or something similar. If you got an answer that works, good job. Here's another one:

```
196, 144, 169, 182, 121
```

Did you get it? The answer is one hundred and eighty-two. All the other numbers are squares of integers.

One more:

```
3, null, 1, 7, 4, 5
```

And the answer is . . . `null`! All of the other items in the list are integers, but `null` isn't an integer.

What? Was that too easy?

Null Overview

As out of place as `null` looks in that list of integers, many computer languages actually include it. In the past Dart did, too, but starting with version 2.12, Dart decided to take `null` out of the list and only put it back if you allow Dart to do so. This feature is called **sound null safety**.

Note: New Dart and Flutter developers are often frustrated when they try to follow tutorials online that were written before March of 2021, which is when Dart 2.12 came out. Modern Dart complains with lots of errors about that old Dart code. Sometimes the solution is as easy as adding `?` after the type name. Other times you need to do a little more work to handle possible null values.

What Null Means

Null means "no value" or "absence of a value". It's quite useful to have such a concept. Imagine not having null at all. Say you ask a user for their postal code so that you can save it as an integer in your program:

```
int postalCode = 12345;
```

Everything will go fine until you get a user who doesn't have a postal code. Your program requires some value, though, so what do you give it? Maybe `0` or `-1`?

```
int postalCode = -1;
```

Choosing a number like `-1`, though, is somewhat arbitrary. You have to define it yourself to mean "no value" and then tell other people that's what it means.

```
// Hey everybody, -1 means that the user
// doesn't have a postal code. Don't forget!
int postalCode = -1;
```

On the other hand, if you can have a dedicated value called `null`, which everyone already understands to mean "no value", then you don't need to add comments explaining what it means.

```
int postalCode = null;
```

It's obvious here that there's no postal code. In versions of Dart prior to 2.12, that line of code worked just fine. However, now it's no longer allowed. You get the following error:

```
A value of type 'Null' can't be assigned to a variable of type 'int'.
```

What's wrong? Null is a useful concept to have! Why not allow it, Dart?

The Problem With Null

As useful as null is for indicating the absence of a value, developers do have a problem with it. The problem is that they tend to forget that it exists. And when developers forget about null, they don't handle it in their code. Those nulls are like little ticking time bombs ready to explode.

To see that in action, open the file with your `main` function and replace the contents of the file with the following:

```
void main() {
  print(isPositive(3));  // true
  print(isPositive(-1)); // false
}

bool isPositive(dynamic anInteger) {
  return !anInteger.isNegative;
}
```

Using `dynamic` turns off Dart's null safety checks and will allow you to observe what it's like to get a surprise `null` value.

Run that code and you'll get a result of `true` and `false` as expected. The `isPositive` method works fine as long as you give it integers. But what if you give it `null`?

Add the following line to the bottom of the `main` function:

```
print(isPositive(null));
```

Run that and your program will crash with the following error:

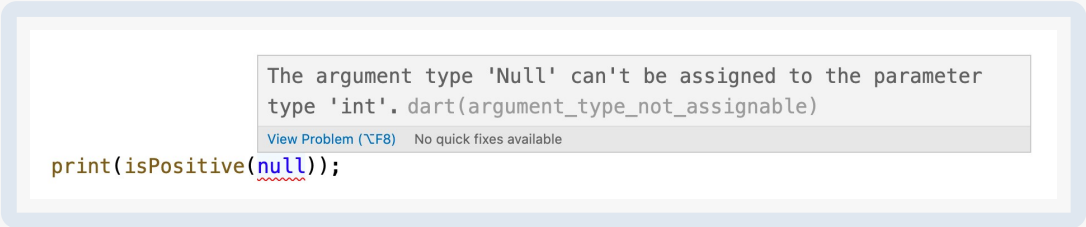
```
NoSuchMethodError: The getter 'isNegative' was called on null.
```

You learned previously that null means "no value", which is true, semantically. However, the Dart keyword `null` actually is a value in the sense that it's an object. That is, the object `null` is the sole instance of the `Null` class. Because the `Null` class doesn't have a method called `isNegative`, you get a `NoSuchMethodError` when you try to call `null.isNegative`.

Now replace `dynamic` with `int` in your `isPositive` function:

```
bool isPositive(int anInteger) {
  return !anInteger.isNegative;
}
```

This turns Dart's null checking back on. All of a sudden you now have an error where you tried to call your function with `null`:



With sound null safety, you *can't* assign a `null` value to an `int` even if you wanted to. Eliminating the possibility of being surprised by `null` prevents a whole class of errors.

Delete that line with the null error. Problem solved.

But wait? Isn't null useful? What about a missing postal code?

Yes, null is useful and Dart has a solution.

Nullable vs. Non-Nullable Types

Dart separates its types into nullable and non-nullable. Nullable types end with a question mark (`?`) while non-nullable types do not.

Non-Nullable Types

Dart types are **non-nullable by default**. That means they're *guaranteed* to never contain the value `null`, which is the essence of the meaning of **sound** in the phrase "sound null safety". These types are easy to recognize because, unlike nullable types, they don't have a question mark at the end.

Here are some example values that non-nullable types could contain:

- **`int`**: `3`, `1`, `7`, `4`, `5`
- **`double`**: `3.14159265`, `0.001`, `100.5`
- **`bool`**: `true`, `false`
- **`String`**: `'a'`, `'hello'`, `'Would you like fries with that?'`
- **`User`**: `ray`, `vicki`, `anonymous`

These are all acceptable ways to set the values:

```
int myInt = 1;
double myDouble = 3.14159265;
bool myBool = true;
String myString = 'Hello, Dart!';
User myUser = User(id: 42, name: 'Ray');
```

If you replaced any of the values on the right with `null`, Dart would give you a compile-time error.

Nullable Types

A **nullable type** can contain the `null` value in addition to its own data type. You can easily tell the type is nullable because it ends with a question mark (`?`), which is like saying, "Maybe you've got the data you want or maybe you've got `null`. That's the question." Here are some example values that nullable types could contain:

- **int?**: `3`, `null`, `1`, `7`, `4`, `5`
- **double?**: `3.14159265`, `0.001`, `100.5`, `null`
- **bool?**: `true`, `false`, `null`
- **String?**: `'a'`, `'hello'`, `'Would you like fries with that?'`, `null`
- **User?**: `ray`, `vicki`, `anonymous`, `null`

That means you can set any of them to `null`:

```
int? myInt = null;
double? myDouble = null;
bool? myBool = null;
String? myString = null;
User? myUser = null;
```

The question mark at the end of `String?` isn't an operator acting on the `String` type. Rather, `String?` is a whole new type separate from `String`. `String?` means that the variable can either contain a string or it can be `null`. It's a union of the `String` and `Null` types.

Every non-nullable type in Dart has a corresponding nullable type: `int` and `int?`, `bool` and `bool?`, `User` and `User?`, `Object` and `Object?`. By choosing the type, you get to choose when you want to allow null values and when you don't.

> **Note**: The non-nullable type is a subtype of its nullable form. For example, `String` is a subtype of `String?` since `String?` can be a `String`.

For any nullable variable in Dart, if you don't initialize it with a value, it'll be given the default value of `null`.

Create three variables of different nullable types:

```
int? age;
double? height;
String? message;
```

Then print them:

```
print(age);
print(height);
print(message);
```

You'll see `null` for each value.

Exercises

1. Create a `String?` variable called `profession`, but don't give it a value. Then you'll have `profession` `null`. Get it? Professional? :]
2. Give `profession` a value of "basketball player".
3. Write the following line and then hover your cursor over the variable name. What type does Dart infer `iLove` to be? `String` or `String?`?

```
const iLove = 'Dart';
```

Handling Nullable Types

The big problem with the old nullable types in the past was how easy it was to forget to add code to handle `null` values. That's no longer true. Dart now makes it impossible to forget because you really can't do much at all with a nullable value until you've dealt with the possibility of `null`.

Try out this example:

```
String? name;
print(name.length);
```

Dart doesn't let you run that code, so there isn't even an opportunity to get a runtime `NoSuchMethodError` like before. Instead, Dart gives you a compile-time error:

```
The property 'length' can't be unconditionally accessed because the receiver can be 'null'.
```

Compile-time errors are your friends because they're easy to fix. In the next few sections, you'll see how to use the many tools Dart has to deal with null values.

Type Promotion

The **Dart analyzer** is the tool that tells you what the compile-time errors and warnings are. It's smart enough to tell in a wide range of situations if a nullable variable is guaranteed to contain a non-null value or not.

Take the last example, but this time assign `name` a string literal:

```
String? name;
name = 'Ray';
print(name.length);
```

Even though the type is still nullable, Dart can see that `name` can't possibly be `null` because you assigned it a non-null value right before you used it. There's no need for you to explicitly "unwrap" `name` to get at its `String` value. Dart does this for you automatically. This is known as **type promotion**. Dart promotes the nullable and largely unusable `String?` type to a non-nullable `String` with no extra work from you! Your code stays clean and beautiful. Take some time right now to send the Dart team a thank-you letter.

Flow Analysis

Type promotion works for more than just the trivial example above. Dart uses sophisticated **flow analysis** to check every possible route the code could take. As long as none of the routes come up with the possibility of `null`, it's promotion time!

Take the following slightly less trivial example:

```
bool isPositive(int? anInteger) {
  if (anInteger == null) {
    return false;
  }
  return !anInteger.isNegative;
}
```

In this case, you can see that by the time you get to the `anInteger.isNegative` line, `anInteger` can't possibly be `null` because you've already checked for that. Dart's flow analysis could also see that, so Dart promoted `anInteger` to its non-nullable form, that is, to `int` instead of `int?`.

Even if you had a much longer and nested `if-else` chain, Dart's flow analysis would still be able to determine whether to promote a nullable type or not.

Null-Aware Operators

In addition to flow analysis, Dart also gives you a whole set of tools called **null-aware operators** that can help you handle potentially null values. Here they are in brief:

- `??` : If-null operator.
- `??=` : Null-aware assignment operator.
- `?.` : Null-aware access operator.
- `?.` : Null-aware method invocation operator.
- `!` : Null assertion operator.
- `?..` : Null-aware cascade operator.
- `?[]` : Null-aware index operator.
- `...?` : Null-aware spread operator.

The following sections describe in more detail how most of these operators work. The last two, however, require a knowledge of collections, so you'll have to wait until Chapter 12, "Lists", to learn about them.

If-Null Operator (??)

One convenient way to handle `null` values is to use the `??` double question mark, also known as the **if-null operator**. This operator says, "If the value on the left is `null`, then use the value on the right." It's an easy way to provide a default value for when a variable is empty.

Take a look at the following example:

```
String? message;
final text = message ?? 'Error';
```

Here are a couple of points to note:

- Since `message` is `null`, `??` will set `text` equal to the right-hand value: `'Error'`. If `message` hadn't been `null`, it would have retained its value.
- Using `??` ensures that `text` can never be `null`, thus Dart infers the variable type of `text` to be `String` and not `String?`.

Print `text` to confirm that Dart assigned it the `'Error'` string rather than `null`.

Using the `??` operator in the example above is equivalent to the following:

```
String text;
if (message == null) {
  text = 'Error';
} else {
  text = message;
}
```

That's six lines of code instead of one when you use the `??` operator. You know which one to choose.

Null-Aware Assignment Operator (??=)

In the example above, you had two variables: `message` and `text`. However, another common situation is when you have a single variable that you want to update if its value is `null`.

For example, say you have an optional font-size setting in your app:

```
double? fontSize;
```

When it's time to apply the font size to the text, your first choice is to go with the user-selected size. If they haven't chosen one, then you'll fall back on a default size of `20.0`. One way to achieve that is by using the if-null operator like so:

```
fontSize = fontSize ?? 20.0;
```

However, there's an even more compact way to do it. In the same way that the following two forms are equivalent,

```
x = x + 1;
x += 1;
```

there's also a **null-aware assignment operator** (`??=`) to simplify if-null statements that have a single variable:

```
fontSize ??= 20.0;
```

If `fontSize` is `null`, then it will be assigned `20.0`, but otherwise, it retains its value. The `??=` operator combines the null check with the assignment.

Both `??` and `??=` are useful for initializing variables when you want to guarantee a non-null value.

Null-Aware Access Operator (?.)

Earlier with `anInteger.isNegative`, you saw that trying to access the `isNegative` property when `anInteger` was `null` caused a `NoSuchMethodError`. There's also an operator for null safety when accessing object members. The **null-aware access operator** (`?.`) returns `null` if the left-hand side is `null`. Otherwise, it returns the property on the right-hand side.

Look at the following example:

```
int? age;
print(age?.isNegative);
```

Since `age` is `null`, the `?.` operator prevents that code from crashing. Instead, it just returns `null` for the whole expression inside the `print` statement. Run that and you'll see the following:

```
null
```

Internally, a property is just a getter method on an object, so the `?.` operator works the same way to call methods as it does to access properties.

Therefore, another name for `?.` is the **null-aware method invocation operator**. As you can see, invoking the `toDouble()` method works the same way as accessing the `isNegative` property:

```
print(age?.toDouble());
```

Run that and it'll again print "null" without an error.

The `?.` operator is useful if you want to only perform an action when the value is non-null. This allows you to gracefully proceed without crashing the app.

Null Assertion Operator (!)

Sometimes Dart isn't sure whether a nullable variable is `null` or not, but *you* know it's not. Dart is smart and all, but machines don't rule the world yet.

So if you're absolutely sure that a variable isn't `null`, you can turn it into a non-nullable

type by using the **null assertion operator** (`!`), or sometimes more generally referred to as the **bang operator**.

```
String nonNullableString = myNullableString!;
```

Note the `!` at the end of `myNullableString`.

> **Note**: In Chapter 5, "Control Flow", you learned about the not-operator, which is also an exclamation mark. To differentiate the not-operator from the null assertion operator, you can also refer to the not-operator as the **prefix `!` operator** because it goes before an expression. By the same reasoning, you can refer to the null assertion operator as the **postfix `!` operator** since it goes after an expression.

Here's an example to see the assertion operator at work. In your project, add the following function that returns a nullable Boolean:

```
bool? isBeautiful(String? item) {
  if (item == 'flower') {
    return true;
  } else if (item == 'garbage') {
    return false;
  }
  return null;
}
```

Now in `main`, write this line:

```
bool flowerIsBeautiful = isBeautiful('flower');
```

You'll see this error:

```
A value of type 'bool?' can't be assigned to a variable of type bool
```

The `isBeautiful` function returned a nullable type of `bool?`, but you're trying to assign it to `flowerIsBeautiful`, which has a non-nullable type of `bool`. The types are different, so you can't do that. However, you *know* that `'flower'` is beautiful; the function won't return `null`. So you can use the null assertion operator to tell Dart that.

Add the postfix `!` operator to the end of the function call:

```
bool flowerIsBeautiful = isBeautiful('flower')!;
```

Now there are no more errors.

Alternatively, since `bool` is a subtype of `bool?`, you could also cast `bool?` down using the `as` keyword that you learned about in Chapter 3, "Types & Operations".

```
bool flowerIsBeautiful = isBeautiful('flower') as bool;
```

This is equivalent to using the assertion operator. The advantage of `!` is that it's shorter.

Beware, though. Using the assertion operator, or casting down to a non-nullable type, will crash your app with a runtime error if the value actually does turn out to be `null`, so don't use it unless you can guarantee that the variable isn't `null`.

Here's an alternative that won't ever crash the app:

```
bool flowerIsBeautiful = isBeautiful('flower') ?? true;
```

You're leaving the decision up to the function, but giving it a default value by using the `??` operator.

Think of the `!` assertion operator as a dangerous option and one to be used sparingly. By using it, you're telling Dart that you want to opt out of null safety. This is similar to using `dynamic` to tell Dart that you want to opt out of type safety.

> **Note**: You'll see a common and valid use of the null assertion operator in the section below titled **No Promotion for Non-Local Variables**.

Null-Aware Cascade Operator (?..)

In Chapter 8, "Classes", you learned about the `..` **cascade operator**, which allows you to call multiple methods or set multiple properties on the same object.

Give a class like this:

```
class User {
  String? name;
  int? id;
}
```

If you know the object isn't nullable, you can use the cascade operator like so:

```
User user = User()
  ..name = 'Ray'
  ..id = 42;
```

However, if your object *is* nullable, like in the following example:

```
User? user;
```

Then you can use the `?..` **null-aware cascade operator**):

```
user
  ?..name = 'Ray'
  ..id = 42;
```

You only need to use `?..` for the first item in the chain. If `user` is `null`, then the chain will be **short-circuited**, or terminated, without calling the other items in the cascade chain.

This is similar for the null-aware access operator (`?.`) as well. Look at this example:

```
String? lengthString = user?.name?.length.toString();
```

Since `user` might be `null`, it needs the `?.` operator to access `name`. Since `name` also might be `null`, it needs the `?.` operator to access `length`. However, as long as `name` isn't `null`, `length` will never be `null`, so you only use the `.` dot operator to call `toString`. If either `user` or `name` is `null`, then the entire chain is immediately short-circuited and `lengthString` is assigned `null`.

Initializing Non-Nullable Fields

When you create an object from a class, Dart requires you to initialize any non-nullable member variables before you use them.

Say you have a `User` class like this:

```
class User {
  String name;
}
```

Since `name` is `String` and not `String?`, you must initialize it somehow. If you recall what you learned in Chapter 8, "Classes", there are a few different ways to do that.

Using Initializers

One way to initialize a property is to use an **initializer** value:

```
class User {
  String name = 'anonymous';
}
```

In this example, the value is `'anonymous'`, so Dart knows that `name` will always get a non-null value when an object is created from this class.

Using Initializing Formals

Another way to initialize a property is to use an **initializing formal**, that is, by using `this` in front of the field name:

```
class User {
  User(this.name);
  String name;
}
```

Having `this.name` as a required parameter ensures that `name` will have a non-null value.

Using an Initializer List

You can also use an **initializer list** to set a field variable:

```
class User {
  User(String name)
    : _name = name;
  String _name;
}
```

The private `_name` field is guaranteed to get a value when the constructor is called.

Using Default Parameter Values

Optional parameters default to `null` if you don't set them, so for non-nullable types, that means you *must* provide a default value.

You can set a default value for positional parameters like so:

```
class User {
  User([this.name = 'anonymous']);
  String name;
}
```

Or like this for named parameters:

```
class User {
  User({this.name = 'anonymous'});
  String name;
}
```

Now even when creating an object without any parameters, `name` will still at least have a default value.

Required Named Parameters

As you learned in Chapter 7, "Functions", if you want to make a named parameter required, use the `required` keyword.

```
class User {
  User({required this.name});
  String name;
}
```

Since `name` is required, there's no need to provide a default value.

Nullable Instance Variables

All of the methods above guaranteed that the class field will be initialized, and not only initialized, but initialized with a non-null value. Since the field is non-nullable, it's not even possible to make the following mistake:

```
final user = User(name: null);
```

Dart won't let you do that. You'll get the following compile-time error:

```
The argument type 'Null' can't be assigned to the parameter type 'String'
```

Of course, if you want the property to be nullable, then you can use a nullable type, and then there's no need to initialize the value.

```
class User {
  User({this.name});
  String? name;
}
```

`String?` makes `name` nullable. Now it's your responsibility to handle any `null` values it may contain.

No Promotion for Non-Local Variables

One topic that people often get confused about is the lack of type promotion for nullable instance variables.

As you recall from earlier, Dart promotes nullable variables in a method to their non-nullable counterpart if Dart's flow analysis can guarantee the variable will never be null:

```
bool isLong(String? text) {
  if (text == null) {
    return false;
  }
  return text.length > 100;
}
```

In this example, the local variable `text` is guaranteed to be non-null if the line with `text.length` is ever reached, so Dart promotes `text` from `String?` to `String`.

However, take a look at this modified example:

```
class TextWidget {
  String? text;

  bool isLong() {
    if (text == null) {
      return false;
    }
    return text.length > 100; // error
  }
}
```

The line with `text.length` now gives an error:

```
The property 'length' can't be unconditionally accessed because the receiver can be 'null'.
```

Why is that? You just checked for `null` after all.

The reason is that the Dart compiler can't guarantee that other methods or subclasses won't change the value of a non-local variable before it's used.

Since Dart has gone the path of *sound* null safety, this guarantee is essential before type promotion can happen.

You do have options, however. One is to use the `!` operator:

```
bool isLong() {
  if (text == null) {
    return false;
  }
  return text!.length > 100;
}
```

Even if the compiler doesn't know that `text` isn't null, *you* know it's not, so you apply that knowledge with `text!`.

Another option is to shadow the non-local variable with a local one:

```
class TextWidget {
  String? text;

  bool isLong() {
    final text = this.text; // shadowing
    if (text == null) {
      return false;
    }
    return text.length > 100;
  }
}
```

The local variable `text` shadows the instance variable `this.text` and the compiler is happy.

The Late Keyword

Sometimes you want to use a non-nullable type, but you can't initialize it in any of the ways you learned above.

Here's an example:

```
class User {
  User(this.name);

  final String name;
  final int _secretNumber = _calculateSecret();

  int _calculateSecret() {
    return name.length + 42;
  }
}
```

You have this non-nullable field named `_secretNumber`. You want to initialize it based on the return value from a complex algorithm in the `_calculateSecret` instance method. You have a problem, though, because Dart doesn't let you access instance methods during initialization.

```
The instance member '_calculateSecret' can't be accessed in an initializer.
```

To solve this problem, you can use the `late` keyword. Add `late` to the start of the line initializing `_secretNumber`:

```
late final int _secretNumber = _calculateSecret();
```

Dart accepts it now, and there are no more errors.

Using `late` means that Dart doesn't initialize the variable right away. It only initializes it when you access it the first time. This is also known as **lazy initialization**. It's like procrastination for variables.

It's also common to use `late` to initialize a field variable in the constructor body. Here's an alternate version of the example above:

```
class User {
  User(this.name) {
    _secretNumber = _calculateSecret();
  }
  late final int _secretNumber;
  // ...
}
```

Initializing a final variable in the constructor body wouldn't have been allowed if it weren't marked as `late`.

Dangers of Being Late

The example above was for initializing a final variable, but you can also use `late` with non-final variables. You have to be careful with this, though:

```
class User {
  late String name;
}
```

Dart doesn't complain at you, because using `late` means that you're promising Dart that you'll initialize the field before it's ever used. This moves checking from compile-time to runtime.

Now add the following code to `main` and run it:

```
final user = User();
print(user.name);
```

You broke your word and never initialized `name` before you used it. Dart is disappointed with you, and complains accordingly:

```
LateInitializationError: Field 'name' has not been initialized.
```

For this reason, it's somewhat dangerous to use `late` when you're not initializing it either in the constructor body or in the same line that you declare it.

Like with the null assertion operator (`!`), using `late` sacrifices the assurances of sound null safety and puts the responsibility of handling `null` into your hands. If you mess up, that's on you.

Benefits of Being Lazy

Who knew that it pays to be lazy sometimes? Dart knows this, though, and uses it to great advantage.

There are times when it might take some heavy calculations to initialize a variable. If you never end up using the variable, then all that initialization work was a waste. Since *lazy* initialization is never done until you actually use the variable, though, this kind of work will never be wasted.

Top-level and static variables have always been lazy in Dart. As you learned above, the `late` keyword makes other variables lazy, too. That means even if your variable is nullable, you can still use `late` to get the benefit of making it lazy.

Here's what that would look like:

```
class SomeClass {
  late String? value = doHeavyCalculation();
  String? doHeavyCalculation() {
    // do heavy calculation
  }
}
```

The method `doHeavyCalculation` is only run after you access `value` the first time. And if you never access it, you never do the work.

Well, that wraps up this chapter. Sound null safety makes Dart a powerful language and is a relatively rare feature among the world's computer programming languages. Aren't you glad you chose to learn Dart?

Challenges

Before moving on, here are some challenges to test your knowledge of nullability. It's best if you try to solve them yourself, but solutions are available with the supplementary materials for this book if you get stuck.

Challenge 1: Naming Customs

People around the world have different customs for giving names to children. It would be difficult to create a data class to accurately represent them all, but try it like this:

- Create a class called `Name` with `givenName` and `surname` properties.
- Some people write their surname last and some write it first. Add a Boolean property called `surnameIsFirst` to keep track of this.
- Not everyone in the world has a surname.
- Add a `toString` method that prints the full name.

Key Points

- Null means "no value."
- A common cause of errors for programming languages in general comes from not properly handling `null`.
- Dart 2.12 introduced sound null safety to the language.
- Sound null safety distinguishes nullable and non-nullable types.
- A non-nullable type is guaranteed to never be `null`.
- Null-aware operators help developers to gracefully handle `null`.

```
??    if-null operator
??=   null-aware assignment operator
?.    null-aware access operator
?.    null-aware method invocation operator
!     null assertion operator
?..   null-aware cascade operator
?[]   null-aware index operator
...?  null-aware spread operator
```

- The `late` keyword allows you to delay initializing a field in a class.
- Using `late` also makes initialization lazy, so a variable's value won't be calculated until you access the variable for the first time.
- `late` and `!` opt out of sound null safety, so use them sparingly.

Where to Go From Here?

In the beginning, Dart didn't support null safety. It's an evolving and ever-improving language. Since development and discussions about new features all happen out in the open, you can watch and even participate. Go to dart.dev/community to learn more.

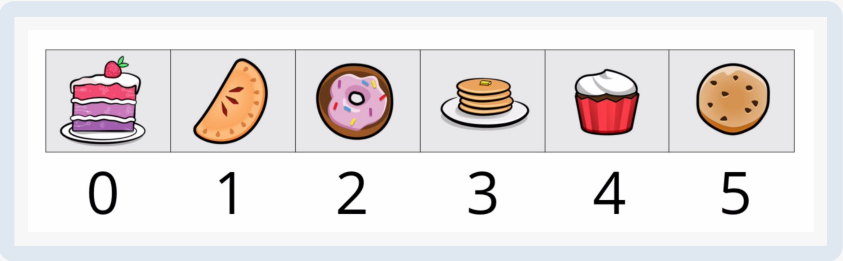
12 Lists

Written by Jonathan Sande

Need to go shopping? Make a shopping list so you don't forget what to buy. Have a bunch of tasks to finish? Make a to-do list. Lists are a part of everyday life. They're also an important part of programming.

In almost every application you make, you'll deal with data collections. `List` is the primary collection type you'll work with in Dart. A **list** is ideal for storing many objects of the same type in an ordered way. Lists in Dart are like what other languages call **arrays**.

The image below represents a list with six **elements**. Lists are **zero-based**, so the first element is at **index** `0`. The **value** of the first element is "cake", the value of the second element is "pie" and so on until the last element at index `5`, which is "cookie".



The order of a list matters. Pie comes after cake but before donut. If you loop through the list multiple times, you can be sure the elements will stay in the same location and order.

Basic List Operations

Like a good dessert, Dart lists have a lot of goodness baked right in. In the next few sections, you'll learn how to create lists, modify them and access their elements.

Creating a List

You can create a list by specifying its initial elements in square brackets. This is called a **list literal**.

Write the following line in `main`:

```
var desserts = ['cookies', 'cupcakes', 'donuts', 'pie'];
```

Because all the elements in this list are strings, Dart infers this to be a list of `String` types.

You can reassign `desserts` (but why would one ever want to reassign desserts?) with an empty list like so:

```
desserts = [];
```

Dart still knows that `desserts` is a list of strings. However, if you were to initialize a new, empty list like this:

```
var snacks = [];
```

Dart wouldn't have enough information to know what kind of objects the list should hold. In this case, Dart simply infers it to be a list of `dynamic`. This causes you to lose type safety, which you don't want. If you're starting with an empty list, specify the type like so:

```
List<String> snacks = [];
```

`List` is the data type, or class name, as you learned in Chapter 8, "Classes".

The `< >` angle brackets here are the notation for **generic types** in Dart. A generic list means you can have a list of anything; you just put the type you want inside the angle brackets. In this case, you have a list of strings, but you could replace `String` with any other type. For example, `List<int>` would make a list of integers, `List<bool>` would make a list of Booleans and `List<Grievance>` would make a list of grievances — but you'd have to define that type yourself because Dart doesn't come with any by default.

> **Note**: There's a whole chapter on generic types in *Dart Apprentice: Beyond the Basics*.

A slightly nicer syntax for creating an empty list is to use `var` or `final` and move the generic type to the right:

```
var snacks = <String>[];
```

Dart still has all the information it needs to know this is an empty list of type `String`.

Printing Lists

As you can do with any collection, you can print the contents of a list with a `print` statement. Because `desserts` is currently empty, give it a list with elements again so you have something interesting to show when you print it:

```
desserts = ['cookies', 'cupcakes', 'donuts', 'pie'];
print(desserts);
```

Run that, and you'll see the following:

```
[cookies, cupcakes, donuts, pie]
```

Accessing Elements

To access a list's elements, you reference its index via **subscript notation**, where the index number goes in square brackets after the list name.

```
final secondElement = desserts[1];
print(secondElement);
```

Don't forget that lists are zero-based, so index `1` fetches the second element. Run that code, and you'll see `cupcakes` as expected.

If you know the value but don't know the index, you can use `indexOf` to look it up:

```
final index = desserts.indexOf('pie');
final value = desserts[index];
print('The value at index $index is $value.');
```

Because `'pie'` is the fourth item in the zero-based list, `index` is `3` and `value` is `pie`. Verify that by running the code above:

```
The value at index 3 is pie.
```

Assigning Values to List Elements

You can change the value of a list element the same way you access its value — that is, by using subscript notation:

```
desserts[1] = 'cake';
print(desserts);
```

This changes the value at index `1` from `cupcakes` to `cake`. Run the code to see the difference:

```
[cookies, cake, donuts, pie]
```

Adding Elements to the End of a List

Lists are expandable by default in Dart, so you can use the `add` method to add an element.

```
desserts.add('brownies');
print(desserts);
```

This adds `brownies` to the end of the list.

Run that, and you'll see:

```
[cookies, cake, donuts, pie, brownies]
```

Now, `desserts` has five elements, ending with `brownies`.

Inserting Elements

Sometimes you want to add an element somewhere besides the end of the list. You can accomplish this with the `insert` method.

Write the following code at the bottom of `main`:

```
desserts.insert(1, 'ice cream');
print(desserts);
```

This inserts `ice cream` at index `1`, the second position in the zero-based list. All the elements after ice cream move back one index position.

Run your code, and you'll see the following result in the terminal:

```
[cookies, ice cream, cake, donuts, pie, brownies]
```

Removing Elements

You can remove elements from a list using the `remove` method. So if you'd gotten a little hungry and eaten the cake, you'd write:

```
desserts.remove('cake');
print(desserts);
```

This leaves a list with five elements:

```
[cookies, ice cream, donuts, pie, brownies]
```

If you know the index of the element you want to remove, use `removeAt`. Write the following below your other code:

```
desserts.removeAt(0);
print(desserts);
```

The cookies were at index `0`, so they're gone now, too:

```
[ice cream, donuts, pie, brownies]
```

No worries — there's still plenty of dessert for a midnight snack tonight!

> **Note**: When you remove an item from the beginning or middle of a list, Dart must internally move up by one index all the elements that occur after it. Think of this as standing in a line. When the person at the front of the line leaves, everyone behind them moves up. Similar moving occurs when you insert in a list.
>
> It can harm performance if you need to do frequent insertions or removals near the beginning of a large list. In a case like this, you might consider using a different data structure, such as a linked list, which has good insertion and removal performance anywhere in the list. Read *Data Structures and Algorithms in Dart* if you'd like to learn more.

Sorting Lists

Having order in your life makes things easier. The same is often true for lists. For example, it's easier to tell what the largest or smallest numbers are if you sort the list first. The easiest way to sort a list in Dart is through the `sort` method.

Write the following in `main`:

```
final integers = [32, 73, 2, 343, 7, 10, 1];
integers.sort();
print(integers);
```

You begin here with an unsorted list of integers. Calling `sort` doesn't create a new list but sorts the original list in place.

Observe that by running the code above:

```
[1, 2, 7, 10, 32, 73, 343]
```

Now, you can easily find the smallest and largest integers like so:

```
final smallest = integers[0];
print(smallest); // 1

final lastIndex = integers.length - 1;
final largest = integers[lastIndex];
print(largest); // 343
```

The `sort` method even works on strings:

```
final animals = ['zebra', 'dog', 'alligator', 'cat'];
animals.sort();
print(animals);
```

Run that, and you'll get the expected results:

```
[alligator, cat, dog, zebra]
```

By default, `sort` sorts strings alphabetically and numbers from smallest to largest. You can also sort in other ways if you provide a custom sorting function as the optional

parameter to `sort`. However, you might want to read *Dart Apprentice: Beyond the Basics*, Chapter 2, "Anonymous Functions", first to get a better idea of how these function parameters work.

Exercise

1. Create a list of type `String` and name it `months`.
2. Use the `add` method to add the names of the twelve months.
3. Find the index of March in the list.
4. Use the index to remove March.
5. Insert March back in at the correct position.
6. Print the list after each change to ensure your code is correct.

Mutable and Immutable Lists

In the examples above, you were able to reassign list literals to `desserts` like so:

```
var desserts = ['cookies', 'cupcakes', 'donuts', 'pie'];
desserts = [];
desserts = ['cookies', 'cupcakes', 'donuts', 'pie'];
```

You could do this because you defined `desserts` using the `var` keyword. This has nothing to do with the list itself being immutable or not. It only means you can swap out *different* lists in `desserts`.

Now, try the following using `final`:

```
final desserts = ['cookies', 'cupcakes', 'donuts', 'pie'];
desserts = [];          // not allowed
desserts = ['cake', 'ice cream']; // not allowed
desserts = someOtherList;     // not allowed
```

Unlike `var`, using `final` means you're not allowed to use the `=` assignment operator to give `desserts` a new list.

However, look at this:

```
final desserts = ['cookies', 'cupcakes', 'donuts', 'pie'];
desserts.remove('cookies');  // OK
desserts.remove('cupcakes'); // OK
desserts.add('ice cream');  // OK
```

Obviously, the `final` keyword isn't keeping you from changing the list's contents. What's happening? Perhaps a little story will help.

The House on Wenderlich Way

You live in a house at 321 Lonely Lane. All you have at home are a few brownies, which you munch on as you scour the internet in hopes of finding work. Finally, you get a job as a junior Flutter developer, so you buy a new house at 122 Wenderlich Way. Best of all, your neighbor Ray brings over some cookies, cupcakes, donuts and pie as a housewarming gift! The brownies are still at your old place, but in your excitement about the move, you've forgotten all about them.

Using `var` is like giving you permission to move houses. The first house had brownies. The second house had cookies, cupcakes, donuts and pie: different houses, different desserts.

Using `final`, on the other hand, is like saying, "Here's your house, but this is the last place you can ever live." However, living at a fixed location doesn't mean you can't change what's inside the house. You might live permanently at 122 Wenderlich Way, but it's fine to eat all the cookies and cupcakes in the house and then go to the store and bring home some ice cream. Well, it's fine in that it's permissible, but maybe you should pace yourself a little on the sweets.

So, too, with a `final` list. Even though the memory address is constant, the values at that address are mutable.

Mutable data is nice and all, but it's not so pleasant when you open your cupboard expecting to find donuts but instead discover the neighbor kids traded them for slugs. It's the same with lists — sometimes you just don't want to be surprised.

So how do you get an immutable list? Have you already guessed the answer? Good job if you have!

Creating Deeply Immutable Lists

The solution to creating an immutable list is to mark the variable name with the `const` keyword. This forces the list to be **deeply immutable**, meaning every element of the list must also be a compile-time constant.

Const Variables

Replace the body of `main` with the following example:

```
const desserts = ['cookies', 'cupcakes', 'donuts', 'pie'];
desserts.add('brownie'); // not allowed
desserts.remove('pie'); // not allowed
desserts[0] = 'fudge';  // not allowed
```

Because `const` precedes `desserts` in the example above, you can't add to, remove from or update the list. It would be nice if VS Code would tell you immediately that you're not allowed to do that. But unfortunately, modifying an unmodifiable list will cause a runtime error — not a compile-time error.

Run the code above, and you'll see the following message:

```
UnsupportedError (Unsupported operation: Cannot add to an unmodifiable list)
```

Press the **Stop button** in VS Code to cancel the error and exit your program execution:

Const List Literals

If you're not able to use `const` for the variable itself, you can still make the value deeply immutable by adding the optional `const` keyword before the list literal:

```
final desserts = const ['cookies', 'cupcakes', 'donuts', 'pie'];
```

Such a situation occurs when providing a default value to a `final` field in a class, as in the following example:

```
class Desserts {
 Desserts([this.desserts = const ['cookies']]);
 final List<String> desserts;
}
```

`desserts` is `final`, while the default value is a `const` list literal. This ensures no one can change the contents of the default list. Recall from Chapter 8, "Classes", that `const` values are canonical instances, so there's a performance benefit in addition to the immutability benefit.

Unmodifiable Lists

Finally, if you want an immutable list but won't know the element values until runtime, you can create a list with the `List.unmodifiable` named constructor:

```
final modifiableList = [DateTime.now(), DateTime.now()];
final unmodifiableList = List.unmodifiable(modifiableList);
```

`DateTime.now()` returns the date and time when it's called. You're not going to know that until runtime, which prevents the list from taking `const`. However, passing that list into `List.unmodifiable` makes the new list immutable.

That's enough about mutability for now. Next, you'll see some properties you can access on lists.

List Properties

Collections such as `List` have several properties. To demonstrate them, use the following list of drinks:

```
const drinks = ['water', 'milk', 'juice', 'soda'];
```

Accessing First and Last Elements

Access the first and last element in a list with `first` and `last`:

```
drinks.first // water
drinks.last  // soda
```

This is equivalent to using the first and last index:

```
drinks[0]                  // water
drinks[drinks.length - 1]  // soda
```

Checking If a List Contains Any Elements

You can also check whether a list is empty or not.

```
drinks.isEmpty   // false
drinks.isNotEmpty // true
```

This is equivalent to the following:

```
drinks.length == 0 // false
drinks.length > 0 // true
```

However, it's more readable to use `isEmpty` and `isNotEmpty`.

Looping Over the Elements of a List

When you have a collection like a list, you often need to perform some action on or with each list element. As you learned in Chapter 5, "Control Flow", loops are a great way to perform a repeated task.

Using a For Loop

To perform an action on each list element using a `for` loop, you'll need to combine your knowledge of loops with what you learned about using an index to access the list elements.

Replace the contents of `main` with the following code:

```
const desserts = ['cookies', 'cupcakes', 'donuts', 'pie'];

for (int i = 0; i < desserts.length; i++) {
 final item = desserts[i];
 print('I like $item.');
}
```

Pay attention to the following points of interest:

- Lists are zero-based, so you begin at index `0`.
- `desserts.length` tells you how many elements are in the list, `4` in this case. You continue iterating as long as index `i` is less than `4` because the index of the last index is `3`.
- `desserts[i]` gives you the value of the current element.

Run the code above, and you'll see the following lines printed in the output window:

```
I like cookies.
I like cupcakes.
I like donuts.
I like pie.
```

Using a For-In Loop

It's such a common activity to iterate over the elements of a collection that Dart provides a special loop precisely for this purpose. It's called a **for-in loop**. These loops don't have any sort of index or counter variable associated with them, but they make iterating over a collection convenient.

Add the following code below what you wrote earlier:

```
for (final item in desserts) {
 print('I also like $item!');
}
```

This is much more readable, isn't it? The `in` keyword tells the `for-in` loop to iterate over the collection in order, and on each iteration, to assign the current element to the `item` variable. Because `desserts` is a collection of strings, `item` is inferred to be of type `String`.

Rerun your code, and you'll see the following result:

```
I also like cookies!
I also like cupcakes!
I also like donuts!
I also like pie!
```

Exercise

Start with the following list of numbers:

```
const numbers = [1, 2, 4, 7];
```

Print the square of each number: 1, 4, 16 and 49.

1. First, use a `for` loop.
2. Solve the problem again using a `for-in` loop.

Code as UI

The Flutter framework chose Dart because of its unique characteristics. However, Flutter has also influenced the development of Dart. You can see this with the following additions to the Dart language:

- spread operator.
- collection `if`.
- collection `for`.

They make it easier for Flutter developers to compose user interface layouts completely in code without a separate markup language.

Flutter UI code consists of classes called **widgets**. Three common Flutter widgets are rows, columns and stacks, all of which store their children as `List` collections. Being able to manipulate lists using the spread operator, collection `if` and collection `for` makes it easier to build the UI with code.

The examples below use strings, but in a Flutter app, you would see the same pattern with lists of `Text`, `Icon`, `ElevatedButton` and other `Widget` elements.

Spread Operator (...)

You can combine lists in a few ways. Start by creating the following two lists so you have something to work with:

```
const pastries = ['cookies', 'croissants'];
const candy = ['Junior Mints', 'Twizzlers', 'M&Ms'];
```

One way to combine lists is with the `addAll` method. Write the following code below the two lists you just made:

```
final desserts = ['donuts'];
desserts.addAll(pastries);
desserts.addAll(candy);
print(desserts);
```

This first adds all the elements in `pastries` to `desserts` and then adds all the elements in `candy` to `desserts`.

Run the code to show that's true:

```
[donuts, cookies, croissants, Junior Mints, Twizzlers, M&Ms]
```

Although the above method works fine in normal program execution, it doesn't fit so well with Flutter's code-as-UI style. This is where the **spread operator** (`...`) comes in. This operator expands one list into another.

Replace the code block above with the following version:

```
const desserts = ['donuts', ...pastries, ...candy];
```

The `...` operator takes the elements of `pastries` and `candy` and adds them directly to `desserts` without needing to call any additional methods.

This is much more concise, and if you add a comma after `candy`, you can format the list vertically in typical Flutter fashion:

```
const desserts = [
 'donuts',
 ...pastries,
 ...candy,
];
```

Print `desserts`, and you'll find you have the same results as you did with `addAll`:

```
[donuts, cookies, croissants, Junior Mints, Twizzlers, M&Ms]
```

Collection if

When creating a list, you can use a **collection** `if` to determine whether to include a particular element.

If you had a peanut allergy, for example, you'd want to avoid adding certain candy with peanut butter to a list of candy. Express that with the following code:

```
const peanutAllergy = true;

const sensitiveCandy = [
 'Junior Mints',
 'Twizzlers',
 if (!peanutAllergy) 'Reeses',
];
print(sensitiveCandy);
```

Run that, and you'll see that the `false` condition for the collection `if` prevented `Reeses` from being included in the list:

```
[Junior Mints, Twizzlers]
```

Collection for

There's also a **collection** `for`, which you can use within a list to generate elements based on another list.

Add the following code at the bottom of `main`:

```
const deserts = ['gobi', 'sahara', 'arctic'];
var bigDeserts = [
 'ARABIAN',
 for (var desert in deserts) desert.toUpperCase(),
];
print(bigDeserts);
```

Here, you've created a new list where the final three elements are the uppercase version of the elements from the input list. The syntax is much like a `for-in` loop but without the braces.

Run the code to see:

```
[ARABIAN, GOBI, SAHARA, ARCTIC]
```

In Flutter, you might use the collection `for` to convert a list of strings into `Text` widgets.

Handling Nullable Lists

Thus far in this chapter, you haven't had to worry about nullable values. You'll need to consider them, though.

Nullable Lists vs. Nullable Elements

A few possibilities exist when dealing with null values and lists. Either the list itself could be null, or the values within the list could be null.

Nullable Lists

For example, you might have a list where the list itself is null. Here's what that would look like:

```
List<int>? nullableList = [2, 4, 3, 7];
nullableList = null;
```

For the nullable list, you add the `?` after the angle brackets. That makes it apply to the list, so in effect, you have `List?`. Note that *within* the angle brackets, you have `int` and not

`int?`. As long as the list exists, its elements must also have non-null values.

Nullable Elements

In contrast to that, you might have a list where one or more of the elements are null:

```
List<int?> nullableElements = [2, 4, null, 3, 7];
```

Dart indicates that by marking the type within the angle brackets as nullable — `int?` in this case. `List` itself isn't nullable.

Nullable Lists With Nullable Elements

Finally, you can also have a nullable list with nullable elements:

```
List<int?>? nullableListAndElements = [2, 4, null, 3, 7];
nullableListAndElements = null;
```

`List<int?>?` is the combination of `List?` and `int?`.

Using the Basic Null-Aware Operators

Everything you learned in Chapter 11, "Nullability", applies to handling nullable lists or nullable elements.

Write the following example in `main`:

```
List<String?>? drinks = ['milk', 'water', null, 'soda'];
// 1
for (String? drink in drinks) {
 // 2
 int letters = drink?.length ?? 0;
 print(letters);
}
```

The following explanations apply to the numbered comments:

1. Although the list is nullable, Dart uses flow analysis to see that you've given the list a value. Dart then applies type promotion so you can loop through the elements without doing additional null checks. Here, the type for `drink` is explicitly written as `String?` for clarity, but Dart could use type inference to learn the same thing if you had written `final drink in drinks`.

2. Because an element may be null, you use the `.?` null-aware access operator to get `length` and the `??` if-null operator to provide a default value in case of a null.

Run that, and you'll see the following output for the number of letters in every word:

```
4
5
0
4
```

Using Null-Aware Collection Operators

In addition to the standard ways of handling null that you've learned, two operators apply specifically to lists:

- `?[]` : Null-aware index operator.
- `...?` : Null-aware spread operator.

The following two sections will introduce these.

Null-Aware Index Operator (?[])

The null-aware index operator (`?[]`) is used to access a list's elements when the list itself might be `null` .

Take the following example of a nullable list:

```
List<String>? myDesserts = ['cake', 'pie'];
```

Here, `myDesserts` isn't `null` because you assigned it the value `['cake', 'pie']` .

Now, set `myDesserts` to `null` :

```
myDesserts = null;
```

Try to get the value of one of the items in the list:

```
String? dessertToday = myDesserts?[1];
```

Print `dessertToday` , and you'll see `null` .

If you had tried to retrieve a value from a null list in the days before null safety, you would have crashed your app. However, the `?[]` operator gracefully passes a `null` value on to `dessertToday` .

Null-Aware Spread Operator (...?)

There's also a null-aware spread operator (`...?`), which will omit a list if the list itself is null.

Write the following in `main` :

```
List<String>? coffees;
final hotDrinks = ['milk tea', ...?coffees];
print(hotDrinks);
```

Here, `coffees` hasn't been initialized and therefore is `null` . By using the `...?` operator, you avoid an error that would come by trying to add a null list. The list `hotDrinks` will only include `milk tea` .

Run your code to check that:

```
[milk tea]
```

This completes your study of lists for now. In the next chapter, you'll learn about another type of collection called a set.

Challenges

Before moving on, here are some challenges to test your knowledge of lists. It's best to try to solve them yourself, but solutions are available with the supplementary materials for this book if you get stuck.

Challenge 1: Longest and Shortest

Given the following list:

```
const strings = ['cookies', 'ice cream', 'cake', 'donuts', 'pie', 'brownies'];
```

Find the longest and shortest strings.

Challenge 2: Repetitious Repeats

How can you tell if a list contains duplicates?

Use the following list as an example:

```
final myList = [1, 4, 2, 7, 3, 4, 9];
```

Challenge 3: Sorting it All Out

Write an algorithm to sort a list of integers without using the `sort` method. If you need some help, search online for "bubble sort" and then implement that algorithm in Dart.

Key Points

- Lists store an ordered collection of elements.
- List elements are accessible using a zero-based index.
- The elements of a list are mutable by default.
- The `for-in` loop is a convenient way to iterate over the elements of a list.
- The spread operator (`...`) allows you to expand one list inside another.
- Collection `if` and `for` can be used to create the content of a list dynamically.
- The nullable collection operators `?[]` and `...?` provide additional ways of dealing with nullable lists.

Where to Go From Here?

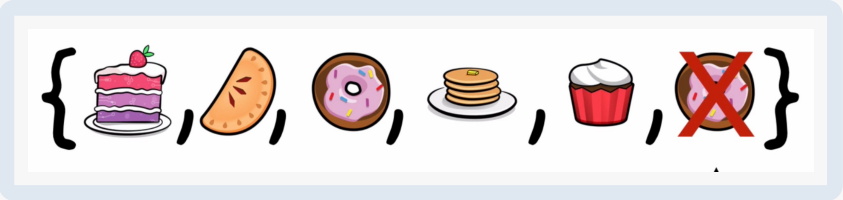
You saw in the "Creating Deeply Immutable Lists" section above that if you try to modify an immutable list, you won't discover your mistake until after you run your program. Runtime mistakes are more difficult to track down. A good practice is to write tests to ensure your code works as intended. Do a web search for "unit testing" and "test driven development" to learn more. Dart has strong support for testing with the `test` package on pub.dev.

13 Sets

Written by Jonathan Sande

The word “set” has more than 400 different definitions in the English language. Are you set to set the set set of sets on the set? That is to say: Are you ready to put the fixed group of mathematical collections on the TV?

The term’s meaning in Dart relates closely to the mathematical meaning. A **set** is a collection of elements where the order isn’t important and duplicate elements are ignored.



This contrasts with Dart lists, in which order is important and duplicates are allowed.

Because of their characteristics, sets can be faster than lists for certain operations, especially when dealing with large datasets. This makes them ideal for quickly checking whether an element exists in a collection. If you wanted to know whether a list of desserts contained donuts, you’d have to check each dessert to find out. Not so with sets. You ask a set if there are donuts, and the set tells you immediately: Yes, there’s one donut!

Creating a Set

You can create an empty set in Dart using the `Set` type annotation like so:

```
final Set<int> someSet = {};
```

The generic syntax with `int` in angle brackets tells Dart the set allows only integers. You could just as easily make the type `String` or `bool` or `double`, though.

The following form is shorter but identical in result:

```
final someSet = <int>{};
```

The curly braces are the same symbols used for sets in mathematics, which should help

you remember them. Be sure to distinguish curly braces in this context from their use for defining scopes, though.

You can also use type inference with a **set literal** to let Dart determine the element types in the set.

```
final anotherSet = {1, 2, 3, 1};
print(anotherSet);
```

Because the set literal contains only integers, Dart can infer the type as `Set<int>`.

Additionally, you probably noticed there are two `1`s there, but because sets ignore duplicates, `anotherSet` ends up with only one `1`. Run that code to verify that `anotherSet` has only one `1`:

```
{1, 2, 3}
```

Operations on a Set

In this section, you'll see some general collection operations that apply to sets.

Checking the Contents

To see whether a set contains an item, use the `contains` method, which returns a `bool`.

Add the following to `main` and run the code:

```
final desserts = {'cake', 'pie', 'donut'};
print(desserts.contains('cake'));    // true
print(desserts.contains('cookies')); // false
```

Because `desserts` contains `cake`, the method returns `true`, whereas checking for `cookies` returns `false`.

Note: The `contains` method of the default `Set` implementation is quite fast. It has **constant time complexity**, which is written as **O(1)** in **Big O notation**. That means no matter how many elements the set contains, it will take the same amount of time to tell you whether a particular value exists in the set or not. There are minor exceptions to that rule, but it's true in the average case.

`List` also has a `contains` method. However, this method is much slower if the list contains many elements. The reason is that `List.contains` has to look at each element to check if the value is the same. This is known as **linear time complexity** and is written as **O(n)** in Big O notation, where *n* is the number of elements in the collection.

Read Chapter 2, "Complexity", in *Data Structures & Algorithms in Dart* to learn more about time complexity and Big O notation.

Adding Single Elements

Like growable lists, you can add and remove elements in a set. To add an element, use the `add` method:

```
final drinks = <String>{};
drinks.add('cola');
drinks.add('water');
drinks.add('cola');
print(drinks);
```

Run that to see the following set:

```
{cola, water}
```

You added `cola` twice, but only one `cola` shows up, as expected.

Removing Elements

You can also remove elements using the `remove` method.

Write the following line below the code you wrote previously:

```
drinks.remove('water');
```

Print `drinks` to reveal only a single element remains:

```
{cola}
```

Just like `contains`, the `add` and `remove` methods of the default `Set` implementation are on average fast, constant-time operations.

Adding Multiple Elements

Use `addAll` to add elements from a list into a set:

```
drinks.addAll(['juice', 'coffee', 'milk']);
```

Print `drinks` again to show the new contents:

```
{cola, juice, coffee, milk}
```

Looping Over the Elements

Like lists, sets are also iterable collections, so you can iterate over the elements with a `for-in` loop.

Write the following to try that out:

```
for (final drink in drinks) {
  print("I'm drinking $drink.");
}
```

Run your code to produce the output below:

```
I'm drinking cola.
I'm drinking juice.
I'm drinking coffee.
I'm drinking milk.
```

Note: Although order isn't an inherent characteristic of sets, the default implementation of `Set` in Dart uses `LinkedHashSet`, which does maintain the order in which the elements were added. Other optional implementations include `HashSet`, with unordered elements, and `SplayTreeSet`, with sorted elements.

Copying Sets

A set's elements are mutable in the same way a list's elements are. If you forget that, you might be surprised if you try to copy a set in the following way:

```
final beverages = drinks;
print(drinks);

beverages.remove('milk');
print(drinks);
print(beverages);
```

`beverages` points to the same object in memory as `drinks` does, so calling `beverages.remove` also removes the element from `drinks`.

Run the code above to see the result:

```
{cola, juice, coffee, milk}
{cola, juice, coffee}
{cola, juice, coffee}
```

The milk is gone from both `drinks` and `beverages`.

If you need to make a copy of a set, a convenient way to achieve this is by calling `toSet` like so:

```
final liquids = drinks.toSet();
print(drinks);

liquids.remove('coffee');
print(drinks);
print(liquids);
```

This time, `liquids` is a different object in memory than `drinks` is.

Run that, and you'll see that removing an element from `liquids` doesn't affect the contents of `drinks`:

```
{cola, juice, coffee}
{cola, juice, coffee}
{cola, juice}
```

Note: The `Set.toSet` trick also works with `List.toList` when you need to copy a list. This only makes a **shallow copy** of the collection, though. That means the elements in the copy still point to the same objects in memory as the elements in the original collection; the elements aren't recreated as new objects with the same values. If those elements are mutable, changing them will modify their values in both copies of the collection. If you need a **deep copy**, where even the elements are copied to new objects, you have to implement that behavior yourself.

Other Operations

Almost everything you learned about lists in Chapter 12, "Lists", also applies to sets. Specifically, you can perform any of the following operations with sets:

- collection `if`
- collection `for`
- spread operators

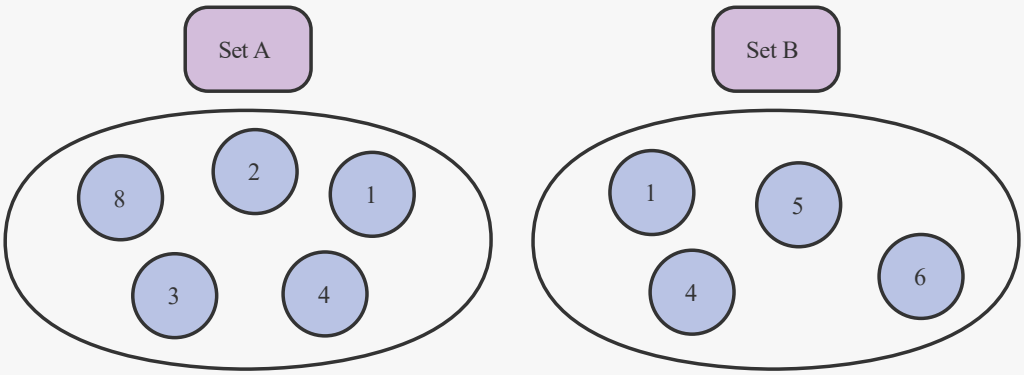
The main thing you can do with lists that you can't do with sets is access the elements by index.

Exercise

1. Create an empty set of type `String` and name it `animals`.
2. Add three animals to it.
3. Check if the set contains `'sheep'`.
4. Remove one of the animals.

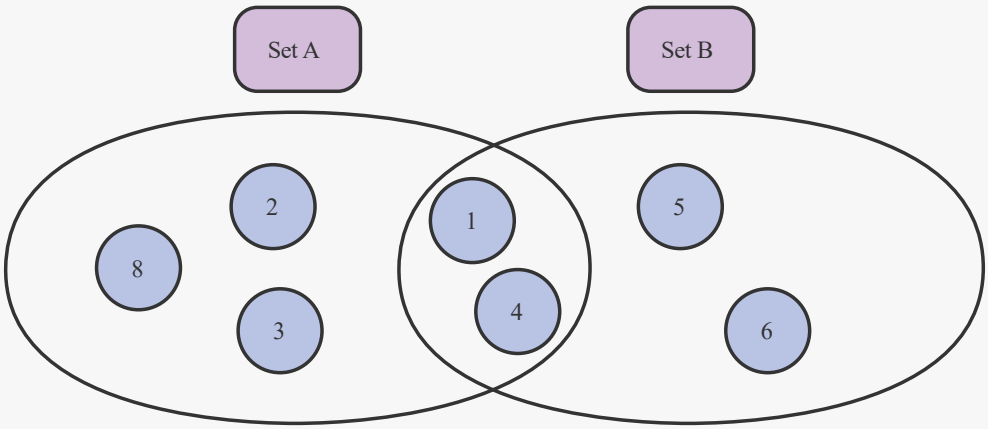
Set Relationships

Sometimes you have multiple datasets and want to know how they compare. Set A and Set B in the diagram below both contain integers, some of which are the same and others are different:



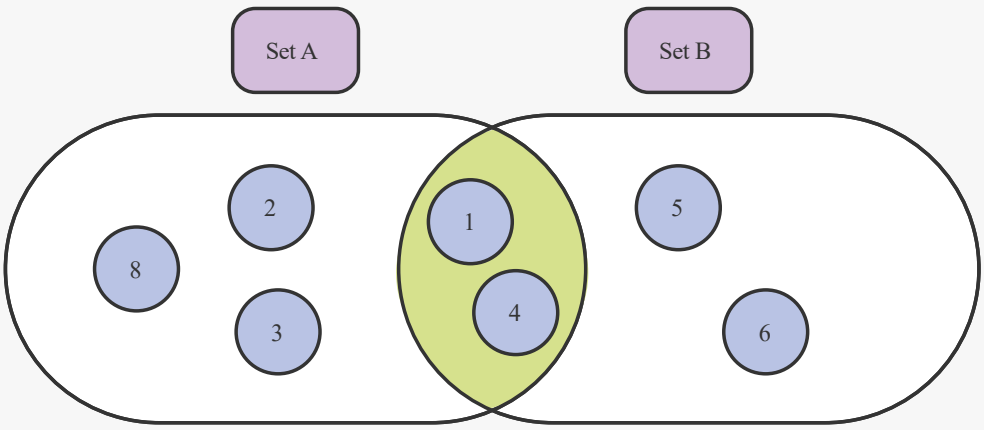
You've probably seen your share of Venn diagrams; if not in school, then at least as memes on the internet. They're useful for showing the common elements between two sets.

In the diagram above, the numbers 1 and 4 occur in both sets. You can represent that with the following Venn diagram, where repeated members appear in the overlapping portion of both ovals:



Intersections

Finding the **intersection** of two sets in Dart tells you the common elements in them. The numbers 1 and 4 are the intersection of Set A and Set B:



Intersection

Given the following two sets:

```
final setA = {8, 2, 3, 1, 4};
final setB = {1, 6, 5, 4};
```

Find the intersection like so:

```
final intersection = setA.intersection(setB);
```

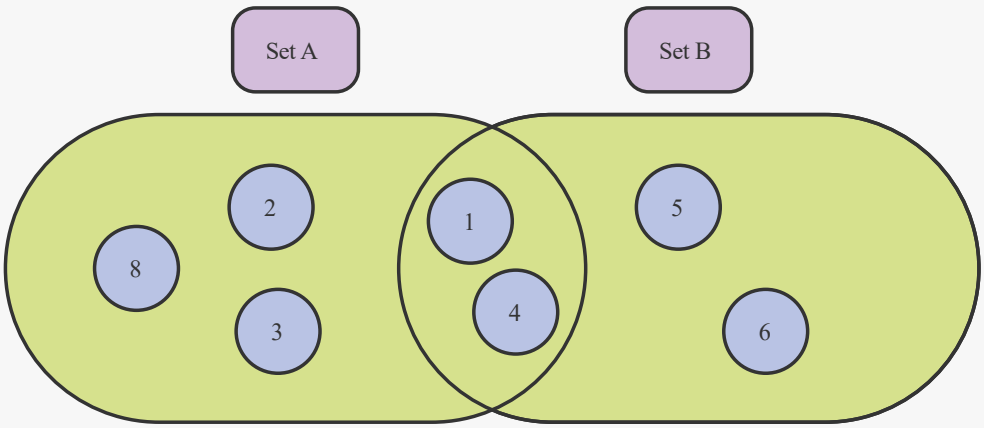
Because both sets contain the numbers 1 and 4, that's the answer you're expecting. Print `intersection` to see:

```
{1, 4}
```

No disappointments there.

Unions

Combining two sets gives you the **union**. If there are any duplicates between the subsets, the union only includes them once:



Union

Unions are as easy to find in Dart as intersections were.

Add the following line below what you wrote earlier:

```
final union = setA.union(setB);
```

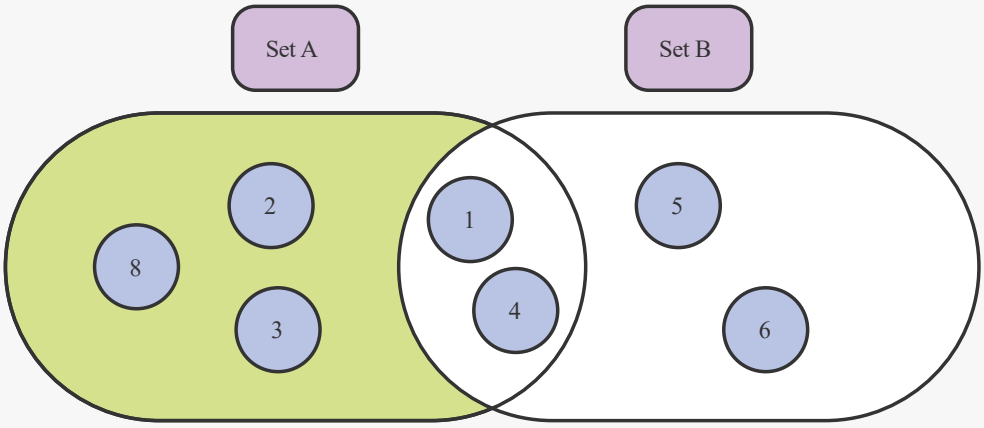
Print `union` to see the results:

```
{8, 2, 3, 1, 4, 6, 5}
```

This union represents all the elements from both sets. Remember — sets do not need to be in order.

Difference

The **difference** is what one set contains that another does not. You can think of it as subtracting one set from another. In the image below, Set A is different than Set B because Set A includes the values 8, 2 and 3 whereas Set B doesn't. Or an alternate way of stating that would be: Set A minus Set B equals the new set `{8, 2, 3}`:



Difference of A - B

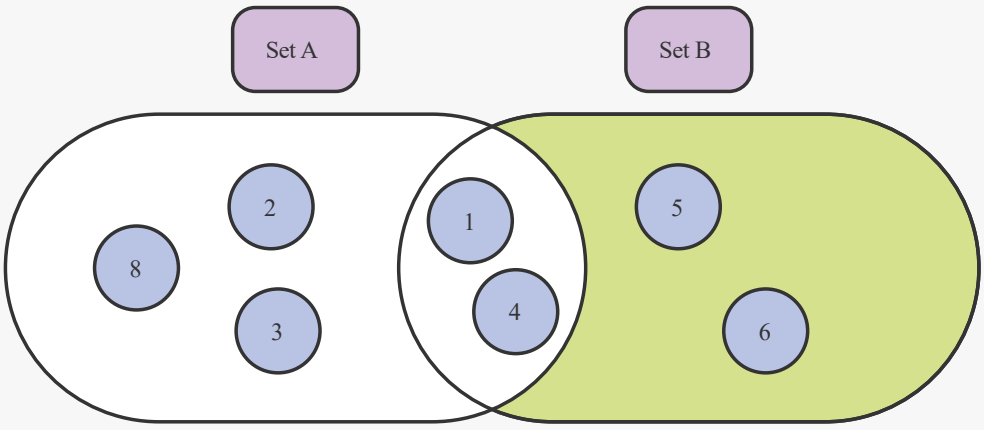
Find the difference in Dart by writing the following:

```
final differenceA = setA.difference(setB);
```

Print `differenceA` to see the set below:

```
{8, 2, 3}
```

Of course, it's also possible to find the difference going the other way:



Difference of B - A

Reverse your As and Bs to write the following:

```
final differenceB = setB.difference(setA);
```

Print `differenceB` to get the expected output:

```
{6, 5}
```

Exercise

Find the intersection and union of the following three sets:

```
final nullSafe = {'Swift', 'Dart', 'Kotlin'};
final pointy = {'Sword', 'Pencil', 'Dart'};
final dWords = {'Dart', 'Dragon', 'Double'};
```

Finding Duplicates

Because adding to and checking a set's elements are fast operations, one application of this data structure is for finding duplicate elements in other collections.

In the sections below, you'll first generate a list of random numbers and then use a set to check whether the list contains any duplicates.

Random Number Generation

Many applications require randomization, especially games. This makes your program more interesting and less predictable. Here are a few examples of where you would want to generate random values:

- Dealing a hand of cards.
- Rolling dice.
- Creating terrain and landscapes.
- Suggesting a secure password.

If the cards were always in the same order, your Solitaire game would get boring, wouldn't it?

Generating a List of Random Integers

You can generate random values in Dart with the `Random` class from the `dart:math` library.

First, import the `dart:math` library at the top of your Dart file:

```
import 'dart:math';
```

Then, add the following code to `main` to generate a list of 10 random integers in the range 1-10:

```
// 1
final randomGenerator = Random();
final randomIntList = <int>[];

for (int i = 0; i < 10; i++) {
  // 2
  final randomInt = randomGenerator.nextInt(10) + 1;
  randomIntList.add(randomInt);
}

print(randomIntList);
```

Here's what's happening:

1. `Random` is a class that generates random values of types `int`, `double` and `bool`.
2. In this case, you want integers, so you call `nextInt`. This method generates a random integer between `0` and one less than the maximum value you give it — in this case, `10`. Because you want a number between `1` and `10`, not `0` and `9`, you must add `1` to the random number.

Run the code above, and you'll see a list of 10 integers. You have a better chance of getting struck by lightning than getting the same numbers as in the list below, but here's what this chapter got:

```
[8, 3, 2, 3, 7, 7, 4, 5, 6, 2]
```

Notice there are some repeats in there. Those are what you want to find in the next step.

Finding Duplicate Integers in the List

Now, add the following code below what you wrote before:

```
// 1
final uniqueValues = <int>{};
final duplicates = <int>{};
```

```
for (int number in randomIntList) {
  // 2
  if (uniqueValues.contains(number)) {
    duplicates.add(number);
  }
  // 3
  uniqueValues.add(number);
}

print(duplicates);
```

This is how you would find the duplicate values in `randomIntList`:

1. Here, you initialize two empty sets: one to store every unique value in `randomIntList`, the other to store the duplicate values.
2. Add any duplicate you find to your duplicate list.
3. You could put this line in an `else` block, but sets ignore adding a duplicate, so it doesn't matter.

Note: If you only wanted to find the unique values in `randomIntList`, you could call `randomIntList.toSet()`. Converting a list to a set removes all the duplicates.

Rerun your code to see something like the following:

```
[2, 4, 3, 7, 9, 5, 1, 10, 2, 9]
{2, 9}
```

The first line is the random list of 10 integers this chapter got. It's different than the list generated earlier because `Random` gives new values every time it's run. The second line shows the result of printing the `duplicates` set. In this instance of `randomIntList`, `2` and `9` are both duplicates.

Note: If you're familiar with random number generation in other programming languages, you might wonder about the **seed**, a number used to initialize the internal algorithm that produces the pseudo-random values. For those who aren't aware of it, computers don't produce truly random numbers. `Random` doesn't require a seed, though, and gives you different values every time you run the code. However, you can optionally provide a seed as an argument to the constructor if you like.

That wraps up your study of sets. You'll learn about another important collection data structure called a **map** in the next chapter. But first, take some time to work on a few challenges.

Challenges

The following challenges will test your knowledge of sets. It's best if you try to solve them yourself, but solutions are available with the supplementary materials for this book if you get stuck.

Challenge 1: A Unique Request

Write a function that takes a paragraph of text and returns a collection of unique `String` characters that the text contains.

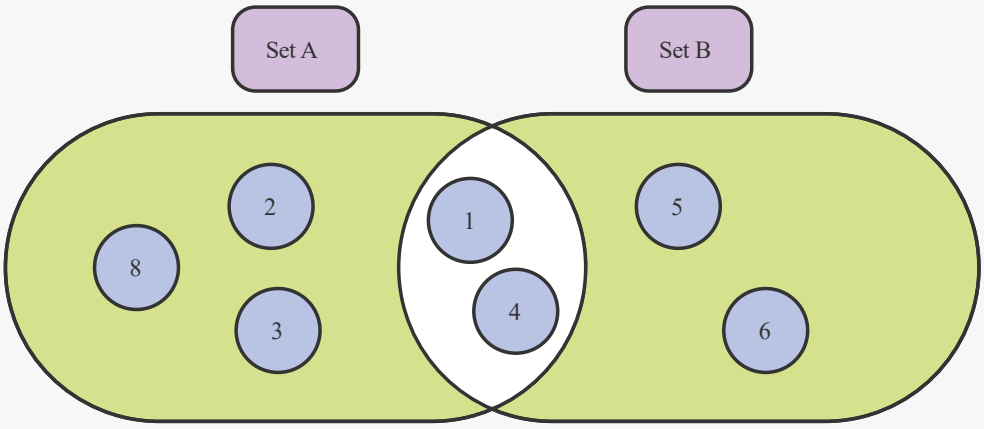
> **Hint**: Use `String.fromCharCode` to convert a code point back to a string.

Challenge 2: Symmetric Difference

Earlier in the chapter, you found the intersection and the union of the following sets:

```
final setA = {8, 2, 3, 1, 4};
final setB = {1, 6, 5, 4};
```

How would you find the set of all values that are unique to each set, meaning everything but the duplicates `1` and `4`:



Symmetric Difference

Key Points

- Sets store unordered collections of unique elements.
- Adding, removing and checking for a value's existence in a set are all fast operations.
- The intersection of two sets is the shared values between them.
- A union of two sets is found by combining the values of both sets.
- The difference of two sets is found by subtracting one set from the other.
- One application of sets is for finding duplicate elements in other collections.

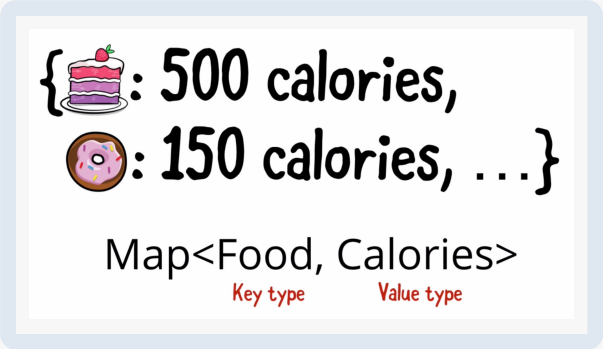
14 Maps

Written by Jonathan Sande

There are no streets or lakes or countries on these maps. **Maps** in Dart are the data structure used to hold key-value pairs. They're like hash maps and dictionaries in other languages.

If you're not familiar with maps, think of them as collections of variables containing data. The **key** is the variable name, and the **value** is the data the variable holds. To find a particular value, give the map the name of the key *mapped* to that value.

In the image below, the cake is mapped to 500 calories and the donut is mapped to 150 calories. `cake` and `donut` are keys, whereas `500` and `150` are values.



Colons separate the key and value in each pair, and commas separate consecutive key-value pairs.

Creating a Map

Like `List` and `Set`, `Map` is a generic type, but `Map` takes two type parameters: one for the key and one for the value. You can create an empty map variable using `Map` and specifying the type for both the key and value:

```
final Map<String, int> emptyMap = {};
```

In this example, `String` is the type for the key, and `int` is the type for the value.

A slightly shorter way to do the same thing is to move the generic types to the right side:

```
final emptyMap = <String, int>{};
```

Notice that maps also use curly braces just as sets do. What do you think you'd get if you wrote this?

```
final emptySomething = {};
```

Is `emptySomething` a set or a map?

It turns out map literals came before set literals in Dart's history, so Dart infers the empty braces to be a `Map` of `<dynamic, dynamic>`. In other words, the types of the key and value are both `dynamic`. If you want a set rather than a map, then you must be explicit:

```
final mySet = <String>{};
```

The single `String` type inside the angle brackets clarifies that this is a set and not a map.

Initializing a Map With Values

You can initialize a map with values by supplying the key-value pairs within the braces. This is known as a **map literal**.

Write the code below in `main`:

```
final inventory = {
  'cakes': 20,
  'pies': 14,
  'donuts': 37,
  'cookies': 141,
};
```

Observe the following points:

- Dart uses type inference to recognize that `inventory` is a map of `String` to `int`, from bakery item to quantity in stock. All the keys are strings, and all the values are integers.
- A colon separates the key and the value. For example, `'cakes'` is the key for the value `20`.
- Commas separate multiple key-value pairs. The first key-value pair is `'cakes': 20`, the second is `'pies': 14` and so on. The comma after the final pair, `'cookies': 141`, is optional. However, including it ensures Dart will auto-format the code vertically.

The key doesn't have to be a string. For example, here's a map of `int` to `String`, from a digit to its English spelling:

```
final digitToWord = {
  1: 'one',
  2: 'two',
  3: 'three',
  4: 'four',
};
```

Print both of those:

```
print(inventory);
print(digitToWord);
```

You'll see the output in horizontal format rather than the vertical format you had above:

```
{cakes: 20, pies: 14, donuts: 37, cookies: 141}
{1: one, 2: two, 3: three, 4: four}
```

Unique Keys

The keys of a map must be unique. A map like the following wouldn't work:

```
final treasureMap = {
  'garbage': 'in the dumpster',
  'glasses': 'on your head',
  'gold': 'in the cave',
  'gold': 'under your mattress',
};
```

There are two keys named `gold`. How are you going to know where to look? You're probably thinking, "Hey, it's gold. I'll just look both places." If you wanted to set it up like that, you could map `String` to `List`:

```
final treasureMap = {
  'garbage': ['in the dumpster'],
  'glasses': ['on your head'],
  'gold': ['in the cave', 'under your mattress'],
};
```

Now, every key contains a list of items, but the keys themselves are unique.

Values don't have that same restriction of being unique. This is fine:

```
final myHouse = {
  'bedroom': 'messy',
  'kitchen': 'messy',
  'living room': 'messy',
  'code': 'clean',
};
```

Operations on a Map

Dart makes it easy to access, add, remove, update and iterate over the key-value pairs in a map.

The examples that follow will continue to use the `inventory` map you made earlier:

```
final inventory = {
  'cakes': 20,
  'pies': 14,
  'donuts': 37,
  'cookies': 141,
};
```

Accessing Key-Value Pairs

You can look up the value of a particular key in a map using a subscript notation similar to that of lists. For maps, though, you use the key rather than an index.

Add the following line at the bottom of `main`:

```
final numberOfCakes = inventory['cakes'];
```

The key `cakes` is mapped to the integer `20`, so print `numberOfCakes` to see `20`.

A map will return `null` if the key doesn't exist. Because of this, accessing an element from a map always gives a nullable value. In the example above, Dart infers `numberOfCakes` to be of type `int?`. If you want to use `numberOfCakes`, you must treat it as you would any other nullable value.

In this case, use the null-aware access operator to check if the number of cakes is even:

```
print(numberOfCakes?.isEven);
```

There were `20`, so that's `true`.

Adding Elements to a Map

You can add new elements to a map simply by assigning them to keys not yet in the map.

Add the following line below what you wrote previously:

```
inventory['brownies'] = 3;
```

Print `inventory` to see `brownies` and its value at the end of the map:

```
{cakes: 20, pies: 14, donuts: 37, cookies: 141, brownies: 3}
```

Updating an Element

Remember that the keys of a map are unique, so if you assign a value to a key that already exists, you overwrite the existing value.

```
inventory['cakes'] = 1;
```

Print `inventory` to confirm that `cakes` was `20` but now is `1`:

```
{cakes: 1, pies: 14, donuts: 37, cookies: 141, brownies: 3}
```

Removing Elements From a Map

Use `remove` to delete elements from a map by key.

```
inventory.remove('cookies');
```

COOKIE! Om nom nom nom nom.

```
{cakes: 1, pies: 14, donuts: 37, brownies: 3}
```

No more cookies.

Accessing Properties

Maps have properties just as lists do. For example, the following properties indicate whether the map is empty:

```
inventory.isEmpty     // false
inventory.isNotEmpty  // true
inventory.length      // 4
```

You also can access the keys and values separately using the `keys` and `values` properties.

```
print(inventory.keys);
print(inventory.values);
```

When you print that, you'll see the following:

```
(cakes, pies, donuts, brownies)
(1, 14, 37, 3)
```

Checking for Key or Value Existence

To check whether a key is in a map, you can use the `containsKey` method:

```
print(inventory.containsKey('pies'));
// true
```

You can do the same for values using `containsValue`.

```
print(inventory.containsValue(42));
// false
```

Note: Adding to, updating and removing key-value pairs from a map are fast operations. Checking for a key with `containsKey` is fast, but checking for a value with `containsValue` is potentially slow because Dart has to iterate through possibly the entire collection of values. This only matters for large collections, though. You probably have nothing to worry about if you're doing one-off value lookups on small maps. However, it's still good to be aware of the performance characteristics of the underlying data structures.

Looping Over Elements of a Map

Unlike lists or sets, you can't directly iterate over a map using a `for-in` loop:

```
for (var item in inventory) {
  print(inventory[item]);
}
```

This produces the following error:

```
The type 'Map<String, int>' used in the 'for' loop must implement Iterable.
```

`Iterable` is a type that knows how to move sequentially, or *iterate*, over its elements. `List` and `Set` both implement `Iterable`, but `Map` does not. You'll learn more in Chapter 15, "Iterables". You'll also learn what "implement" means in *Dart Apprentice: Beyond the Basics*, Chapter 5, "Interfaces".

There are a few solutions, though, for looping over a map. One solution is to use the `Map.forEach` method, which you'll learn in *Dart Apprentice: Beyond the Basics*, Chapter 2, "Anonymous Functions". Additionally, map's `keys` and `values` properties are iterables, so you can loop over them. Here's an example of iterating over the keys:

```
for (var key in inventory.keys) {
  print(inventory[key]);
}
```

You can also use `entries` to iterate over the elements of a map, which gives you both the keys and the values:

```
for (final entry in inventory.entries) {
  print('${entry.key} -> ${entry.value}');
}
```

Run that to see the following result:

```
cakes -> 1
pies -> 14
donuts -> 37
brownies -> 3
```

Before going on, test your knowledge of maps with the following exercise.

Exercise

1. Create a map with the following keys: `name`, `profession`, `country` and `city`. For the values, add your information.
2. You decide to move to Toronto, Canada. Programmatically update the values for `country` and `city`.
3. Iterate over the map and print all the values.

Maps, Classes and JSON

In Chapter 8, "Classes", you learned how JSON is used as a format to convert objects into strings. This is known as **serialization**, and it allows you to send objects over the network or save them in local storage. The JSON format is quite close to the structure of Dart maps, so you'll often use maps as an intermediary data structure when converting between JSON and Dart objects.

The following sections will walk you through these conversion steps:

- Object to map.
- Map to JSON.
- JSON to map.
- Map to object.

You'll start with the object-to-map conversion.

Converting an Object to a Map

Add the following Dart class to your project:

```
class User {
  const User({
    required this.id,
    required this.name,
    required this.emails,
  });

  final int id;
  final String name;
  final List<String> emails;
}
```

There are three properties, each of different types.

Now in `main`, create an object from that class like so:

```
final userObject = User(
  id: 1234,
  name: 'John',
  emails: [
    'john@example.com',
    'jhagemann@example.com',
  ],
);
```

If you were to write that information in the form of a map, it would look like so:

```
final userMap = {
  'id': 1234,
  'name': 'John',
  'emails': [
    'john@example.com',
    'jhagemann@example.com',
  ],
};
```

Notice how similar they are? The disadvantage of using a map is that the parameter names are strings. If you spell them wrong, you get a runtime error rather than the compile-time error the object parameter names would give you.

Many data classes such as `User` have a `toJson` method that returns a map like the one you saw above. Add the following method to `User`:

```
Map<String, dynamic> toJson() {
  return <String, dynamic>{
    'id': id,
    'name': name,
    'emails': emails,
  };
}
```

Here are some notes to pay attention to:

- JSON itself is a string rather than a map, so technically, you should probably call this method `toMap`. But common practice is to call it `toJson`.
- When serializing objects, you lose all type safety. So even though the key is `String`, the values are `dynamic`.

Use the following line to create a map from `userObject`:

```
final userMap = userObject.toJson();
```

Getting a map is just the intermediate step. Next, you'll convert your map to JSON.

Converting a Map to a JSON String

It would be a chore to build a JSON string by hand. Thankfully, the `dart:convert` library already has the map-to-JSON converter built-in.

Add the import at the top of your project file:

```
import 'dart:convert';
```

Now, write the following code at the bottom of `main` to convert your map to a JSON string:

```
final userString = jsonEncode(userMap);
print(userString);
```

`jsonEncode` comes from the `dart:convert` library. You could also replace `jsonEncode` with `json.encode`. They're equivalent. The map you pass in must have string keys, and the values must be simple types like `String`, `int`, `double`, `null`, `bool`, `List` and `Map`.

Run the code above, and you'll see the JSON string printed to the console:

```
{"id":1234,"name":"John","emails":["john@example.com","jhagemann@example.com"]}
```

White space doesn't matter in JSON. So, reformatted, that string would look like so:

```
{
  "id": 1234,
  "name": "John",
  "emails": [
    "john@example.com",
    "jhagemann@example.com"
  ]
}
```

Very similar to the Dart map, isn't it? The difference is that the braces, brackets, colons, commas and quotation marks all are part of the string.

Now that you have a string, it's in a form that allows you to easily save to a database or send across the internet.

How about the other way? Can you take a JSON string and convert it back to an object?

Converting a JSON String to a Map

In *Dart Apprentice: Beyond the Basics*, Chapter 12, "Futures", you'll get some firsthand experience retrieving an actual JSON string from a server on the internet. For now, though, you'll practice with the hard-coded sample string below.

Write the following at the bottom of `main`:

```
final jsonString =
  '{"id":4321,"name":"Marcia","emails":["marcia@example.com"]}';
```

This JSON string contains an ID, user name and an email list with a single email address.

Now, add the following line below that:

```
final jsonMap = jsonDecode(jsonString);
```

This decodes the string and gives you a map. However, if you hover your cursor over `jsonMap`, you see the inferred type is `dynamic`. That's all `jsonDecode` will tell you. It doesn't know beforehand what types might be hiding in that string.

Because that's the case, using `dynamic` for now is fine. But you might run into trouble if you forget that you're using it. It's easy to assume you're working with a map or some other type when you might not be.

Preventing Hidden Dynamic Types

You can keep yourself from forgetting about `dynamic` by adding a setting to your analysis options file. Open **analysis_options.yaml** in the root of your project. Then, add the following lines at the bottom of the file and save your changes:

```
analyzer:
  strong-mode:
    implicit-dynamic: false
```

This tells Dart to always remind you to choose a type or explicitly write `dynamic`. That way, you won't ever forget about it. This is a good idea to use in all your Dart and Flutter projects.

Explicitly Writing Dynamic

Back in your Dart file with the `main` function, the compiler is complaining at you now:

```
Missing variable type for 'jsonMap'.
Try adding an explicit type or removing implicit-dynamic from your analysis options file.
```

To make that error go away, replace `final` in the `jsonDecode` line above with `dynamic`, like so:

```
dynamic jsonMap = jsonDecode(jsonString);
```

Now, you can clearly see that you're working with a `dynamic` value.

Handling Errors

You can do a little error checking on the type using the `is` keyword:

```
if (jsonMap is Map<String, dynamic>) {
  print("You've got a map!");
} else {
  print('Your JSON must have been in the wrong format.');
}
```

Run that, and you'll see:

```
You've got a map!
```

In *Dart Apprentice: Beyond the Basics*, Chapter 10, "Error Handling", you'll learn even more sophisticated ways to deal with errors.

Converting a Map to an Object

You could, at this point, extract all the data from your map and pass it into the `User` constructor. However, that's error-prone if you have to create many objects. Earlier, you added a `toJson` method. Now you'll add a `fromJson` constructor to go the other way.

Add the following factory constructor to your `User` class:

```
factory User.fromJson(Map<String, dynamic> jsonMap) {
  // 1
  dynamic id = jsonMap['id'];
  if (id is! int) id = 0;
  // 2
  dynamic name = jsonMap['name'];
  if (name is! String) name = '';
  // 3
  dynamic maybeEmails = jsonMap['emails'];
  final emails = <String>[];
  if (maybeEmails is List) {
    for (dynamic email in maybeEmails) {
      if (email is String) emails.add(email);
    }
  }
  // 4
  return User(
    id: id,
    name: name,
    emails: emails,
  );
}
```

The numbered comments above correspond with the following notes:

1. First, you attempt to extract the user ID from the map. You don't have type information, so if `id` turns out to be something besides an `int`, you just give `id` a default value of `0`.

2. If `name` isn't a `String`, give it a default value of an empty string. This will keep your app from crashing if it gets junk data. However, rather than creating a user without a name, you might want to throw an error instead so your app can cancel the creation of this user or show an error message. As mentioned earlier, *Dart Apprentice: Beyond the Basics* will talk more about this.

3. First, check that `jsonMap['emails']` is actually a `List`. If it is, make sure each item in the list is a `String` and then add it to your list of known `emails`.

4. Return a new `User` object with the values from the map.

Make your object printable by also adding `toString` to your `User` class:

```
@override
String toString() {
  return 'User(id: $id, name: $name, emails: $emails)';
}
```

Now, you can easily create an object in `main` like so:

```
final userMarcia = User.fromJson(jsonMap);
print(userMarcia);
```

Run the code to check that you've correctly set the values:

```
User(id: 4321, name: Marcia, emails: [marcia@example.com])
```

You've come full circle. You're back to having an object again!

Challenges

Before moving on, here are some challenges to test your knowledge of maps. It's best if you try to solve them yourself, but solutions are available with the book's supplementary materials if you get stuck.

Challenge 1: Counting on You

Write a function that takes a paragraph of text as a parameter. Count the frequency of each character. Return this data as a map where the key is the character and the value is the frequency count.

Challenge 2: To JSON and Back

Create an object from the following class:

```
class Widget {
  Widget(this.width, this.height);
  final double width;
  final double height;
}
```

Then:

1. Add a `toJson` method to `Widget`. It should return a map.
2. Use `toJson` to convert your object to a map.
3. Convert the map to a JSON string.
4. Convert the JSON string back to a map.
5. Add a `fromJson` factory constructor to `Widget`.
6. Use `fromJson` to convert the map back to a widget object.

Key Points

- Maps store a collection of key-value pairs.
- Using a key to access its value always returns a nullable type.
- If you use a `for-in` loop with a map, you need to iterate over the keys or values.
- You can convert a Dart object to a map by making the property names from the object the keys of the map.
- JSON objects use similar syntax to Dart maps.
- The `dart:convert` library allows you to convert between JSON and Dart maps.

Where to Go From Here?

As you continue to create Dart applications, you'll find that you're writing methods and constructors like `toJson` and `fromJson` repeatedly. If you find that tiring, consider using a **code generation** package from pub.dev, such as `freezed` or `json_serializable`. You only need to write the properties for the class, and these packages will generate the rest.

15 Iterables

Written by Jonathan Sande

What comes next in the sequence **a**, **b**, **c**,...? You don't need to think twice to know it's **d**. How about **2**, **4**, **8**, **16**,...? The next power of two is **32**. Again, not much of a challenge. Here's one that's a little trickier: **O**, **T**, **T**, **F**, **F**,...? What letter comes next? You can find the answer at the end of the chapter if you need it.

All these sequences were iterable, and you were the iterator by providing the next value in the sequence. In this chapter, you'll learn what iterables and iterators are in Dart, why they're useful and how to create your own.

What's an Iterable?

An **iterable** in Dart is any collection that lets you loop through its elements. In more technical speak, it's a class that implements the `Iterable` interface. `List` and `Set` are two iterables you're already familiar with. Not every Dart collection is iterable, though. You learned in the previous chapter that you can't directly loop over a map. If you want to visit all of a map's elements, you have to iterate over the `keys`, `values` or `entries`, all of which are iterables.

Reviewing List Iteration

To review, take a look at the following example.

Add the following alphabetically sorted list to `main`:

```
final myList = ['bread', 'cheese', 'milk'];
print(myList);
```

Run that code, and you'll see:

```
[bread, cheese, milk]
```

The `[]` square brackets tell you this is a list.

Now, iterate over the elements with a `for-in` loop:

```
for (final item in myList) {
  print(item);
}
```

Run that, and you'll see each item in the list printed in order:

```
bread
cheese
milk
```

Meeting an Iterable

Lists are one specific kind of iterable, but you can also work directly with the `Iterable` type. These are often accessible as properties of another collection. The example below will demonstrate that.

Add the following code to what you've already written:

```
final reversedIterable = myList.reversed;
print(reversedIterable);
```

Run that, and you'll see the following:

```
(milk, cheese, bread)
```

The items in the list are now shown in reverse order. Additionally, they're surrounded by `()` parentheses rather than `[]` square brackets. The parentheses are Dart's way of telling you this is an `Iterable` and not a `List`.

So how are lists and iterables different? One important point is that an iterable is **lazy**. If you recall from Chapter 11, "Nullability", a lazy property is one whose value isn't calculated until you access it the first time. It's similar for iterables: the elements of an iterable aren't known until you ask for them.

You can imagine that if you had a list of a thousand elements, it would take a little work to reverse that list. You'd have to create a new list, copy the elements from the back of the original list to the front of the new list and continue all the way through for the rest of the elements. The `reversed` getter you saw above doesn't do any of that work. It doesn't create a new list. It doesn't copy anything. It just instantly returns an `Iterable`. The key is that the `Iterable` knows how to give you the reversed elements when you need them...and not before. However, when you printed the list, you needed to know all the elements, so Dart ran through them at that point.

Converting an Iterable to a List

Another way to force Dart to get all the elements from an iterable is to convert the iterable to a list.

Write the following at the end of `main`:

```
final reversedList = reversedIterable.toList();
print(reversedList);
```

Calling `toList` causes the iterable to loop through its contents and store them as values in a new list.

Run the code above, and you'll see a list with the familiar square brackets:

```
[milk, cheese, bread]
```

> **Note**: If you know converting an iterable to a list in your application might be costly, don't call `toList` until you need to or when the app isn't busy with other tasks.

Operations on Iterables

This chapter will cover a few basic operations on iterables. You'll learn other more advanced operations in Chapter 2, "Anonymous Functions", in *Dart Apprentice: Beyond the Basics*.

Creating an Iterable

Trying to directly instantiate an `Iterable` will give you an error. To see that, write the code below in `main`:

```
final myIterable = Iterable();
```

The compiler complains that "Abstract classes can't be instantiated."

Look at the source code of `Iterable` if you want to learn more. In the code above, if you're using VS Code, **Command-click** the class name `Iterable` on a Mac (or **Control-click** it on a PC). This will bring you to the **iterable.dart** source file, and you'll see the following Dart

source code:

```
abstract class Iterable<E> {
  const Iterable();

  // ...
}
```

Abstract classes are used to define all of the methods you want in a class. They can also include logic, which `Iterable` does. You'll learn more about abstract classes in *Dart Apprentice: Beyond the Basics*. For now, it's only important to know you can't create an object directly from an abstract class unless it has a factory constructor, which `const Iterable()` isn't.

But you can create an iterable by specifying the type annotation as `Iterable` and then assigning the variable a list value. Replace the `myIterable` line you wrote earlier with the following:

```
Iterable<String> myIterable = ['red', 'blue', 'green'];
```

The actual implementation is a list, which you know because of the square brackets. But by explicitly writing `Iterable<int>`, you surrender any superpowers that lists have and tell Dart you want to treat the collection as a humble iterable. The reason you do so here is so you can explore how to work directly with iterables in the examples that follow.

Accessing Elements

The way to access a particular element in the collection is to use `elementAt` with an index.

Write the following in `main`:

```
final thirdElement = myIterable.elementAt(2);
print(thirdElement);
```

Recall that `2` is the third element when the indexing starts at zero.

Run that, and you'll see the `green` printed out in the console.

An iterable finds a specific element by starting at the beginning of the collection and counting to the desired element. **Command-click** `elementAt` if you're using a Mac or **Control-click** it if you're using a PC, and view the Dart source code:

```
E elementAt(int index) {
  // ...
  int elementIndex = 0;
  for (E element in this) {
    if (index == elementIndex) return element;
    elementIndex++;
  }
  // ...
}
```

Some of the code in there you may not recognize, but you should be able to see the `elementIndex` counting up inside the `for-in` loop as it searches through the elements until it finds the desired index. You can imagine that this could take some time for a large collection. This is in contrast to lists, where you can immediately access an element using a subscript notation like `myList[2]`. An iterable only knows how to iterate, though. So when you want to find an element, you've got to start counting from the beginning.

Finding the First and Last Elements

Use `first` and `last` to get the first and last elements of an iterable:

```
final firstElement = myIterable.first;
final lastElement = myIterable.last;

print(firstElement);
print(lastElement);
```

The code is simple, but don't let the simplicity deceive you. This is an iterable, so it calculates these values by iterating. You don't need to iterate far to find the *first* element, of course. However, if you want the *last* element, Dart finds it by starting at the beginning of the collection and moving through every element until there aren't any left.

Getting the Length

Finding the number of elements in a collection is as deceptively simple as finding the last element.

Write the following in `main`:

```
final numberElements = myIterable.length;
print(numberElements);
```

Run that, and you'll see `3` as the result.

The way Dart got that `3` was to loop through all the elements and count them one by one.

That's like asking a czar with a jellybean jar how many there are. "Hold on," they say. "I'll tell you today." After waiting some, the answer comes: "5,381". Then a passerby happens by: "My, a jellybean jar! I wonder how many there are." "Just a minute," replies the czar, "and I'll tell you how many there are,"...and then proceeds to count them all over again. That's how iterables work. A list knows how many elements it has. So does a set. But not an iterable. It has to count.

By this point, you may be asking, "Why would I ever want to use an iterable?" That's a good question. If you need to frequently access elements by index or find the length, you shouldn't use an iterable. Using a list is more efficient.

However, remember the advantage of an iterable is that it's lazy. Imagine a two-gigabyte text file and you want to process the whole thing one line at a time. Theoretically, you could create a giant list by splitting on the newline character (see *Dart Apprentice: Beyond the Basics*, Chapter 1, "String Manipulation"), but that likely would freeze your computer. How much better to just wait until it's time to start working, then grab a little data, process it and grab a little more. That's the way an iterable works.

Other Important Methods on Iterable

The `Iterable` class has many other important methods you should learn. Here are a few of them:

- `map`
- `where`
- `expand`
- `contains`
- `forEach`
- `reduce`
- `fold`

The thing is, these methods take anonymous functions as parameters, and you haven't learned what anonymous functions are yet. Don't worry, though — you'll get to them in *Dart Apprentice: Beyond the Basics*. At that time, you'll come back to the `Iterable` methods listed above.

Exercise

1. Create a map of key-value pairs.
2. Make a variable named `myIterable` and assign it the `keys` of your map.
3. Print the third element.
4. Print the first and last elements.
5. Print the length of the iterable.
6. Loop through the iterable with a `for-in` loop.

Creating an Iterable From Scratch

A great way to learn how iterables work is to make an iterable collection from scratch. You're going to create a collection that contains all the squares from 1 to 10,000: 1, 4, 9, 16,... all the way to 10,000.

There's more than one way to make an iterable. The simplest way is probably to use a generator, but you can also go low-level and use an iterator. You'll get a taste of both.

Using a Generator

The functions you've seen previously returned at most a single value. A **generator** is a function that produces multiple values before finishing. With the generator that you're going to create in this chapter, the values come in the form of an iterable. This is known as a **synchronous generator** because all the values are available on demand when you need them.

> **Note**: There's another type of generator called an **asynchronous generator** where you have to wait for the values to become available. You'll learn about this in *Dart Apprentice: Beyond the Basics*, Chapter 13, "Streams".

Creating a Synchronous Generator

Add the following function to your project file:

```
Iterable<int> hundredSquares() sync* {
  for (int i = 1; i <= 100; i++) {
    yield i * i;
  }
}
```

There are two new keywords here:

- `sync*` : Read that as "sync star". You're telling Dart that this function is a synchronous generator. You must return an `Iterable` from such a function.
- `yield` : This is similar to the `return` keyword for normal functions except that `yield` doesn't exit the function. Instead, `yield` generates a single value and then pauses until the next time you request a value from the iterable. Because iterables are lazy, Dart doesn't start the generator function until the first time you request a value from the iterable.

Running the Code

Now that your generator function is finished, replace the contents of `main` with the following:

```
final squares = hundredSquares();
for (int square in squares) {
  print(square);
}
```

Dart won't run `hundredSquares` until it gets to the `for` loop because that's when Dart accesses the values of the `squares` iterable.

Run your code, and you'll see the results below:

```
1
4
9
16
25
...
9604
9801
10000
```

Next, you'll implement this functionality again but at a lower level.

Using an Iterator

Iterables don't know how to move from element to element within their collections. That's the job of an **iterator**. In the previous example, the generator function served the purpose of the iterator, but in this example, you'll create an iterator using the `Iterator` class.

Every `Iterator` must contain the following two components:

- **`bool moveNext()`**: In this method, you provide the logic for how to get the next element in the collection. The method needs to return a Boolean value. As long as more elements remain in the collection, it returns `true`. A value of `false` means the iterator has reached the end of the collection.
- **`current`**: This is a getter that returns the value of the element in your current progress of iterating through the collection. `current` is considered undefined until you call `moveNext` at least once. It's also undefined after the iterator reaches the end of the collection, that is, after `moveNext` has returned `false`. Depending on the implementation, trying to access an undefined `current` might return a reasonable value or it might cause a crash. The behavior is undefined, though, so don't try. That's the agreement you make when you use an iterator.

Implementing the Iterator

Now, you'll make an iterator that knows how to find the next squared number in the series of squares.

Add a **lib** folder to the root of your project if you don't already have one. Then add a new file to **lib** named **squares.dart**.

Next, write the following code in **squares.dart**:

```
class SquaredIterator implements Iterator<int> {

  int _index = 0;

  // 1
  @override
  bool moveNext() {
    _index++;
    return _index <= 100;
  }

  // 2
  @override
  int get current => _index * _index;
}
```

You haven't learned much about the keywords `implements` and `override`. You'll learn about them in *Dart Apprentice: Beyond the Basics*. For now, just pay attention to the implementation of `moveNext` and `current`:

1. Every time `moveNext` is called, you add one to `_index`. This incrementally moves `_index` from 1 to 100. Once `_index` reaches 101, `moveNext` will return `false`, signaling that the iterator has reached the end of the collection.

2. Sometimes, it's useful to have `current` return a stored private value, which you could name `_current`, for example. However, in this case, it was easy enough to just multiply `_index` by itself right here to find the square.

As you can see, you're not storing the values of this collection anywhere. You're simply calculating them as you need them.

Implementing the Iterable

Now that you have your iterator, you can write the code for your iterable.

Add the following class to **lib/squares.dart**:

```
class HundredSquares extends Iterable<int> {
  @override
  Iterator<int> get iterator => SquaredIterator();
}
```

Here are a few comments:

- This is a class, so use upper camel case for the name `HundredSquares`. That's in contrast to the lower camel case you used previously for the generator function named `hundredSquares`.
- `extends` is a keyword you'll learn about in *Dart Apprentice: Beyond the Basics*, Chapter 3, "Inheritance". It means your `HundredSquares` class also gets to use all the logic from `Iterable`.
- Creating a custom iterable only requires providing an `iterator`. This is where you return the instance of your `SquaredIterator` that you made in the previous step.

That's all there is to it. It's time to try your iterable out.

Running the Code

Open the file with your `main` method again and add the following import at the top:

```
import 'package:starter/squares.dart';
```

If your project is named something besides `starter`, then replace `starter` with your project name.

Now in `main`, delete the previous contents, and create your collection like so:

```
final squares = HundredSquares();
```

Because your collection is iterable, you can loop through it with a `for-in` loop just as you would with any other collection.

Add the following code at the bottom of `main`:

```
for (int square in squares) {
  print(square);
}
```

Run that to observe the list of squared numbers displayed in the console:

```
1
4
9
16
25
...
9604
9801
10000
```

Note: Don't try this without adult supervision, but if you're adventurous, you can make an infinitely large collection simply by always returning `true` from the `moveNext` method of your iterator.

When to Use Lists, Sets, Maps or Iterables

Each type of collection has its strengths. Here's some advice about when to use which:

- Choose **lists** if order matters. Try to insert values at the end of lists wherever possible to keep things running smoothly. And be aware that searching can be slow with large collections.
- Choose **sets** if you're only concerned with whether something is in the collection or not. This is faster than searching a list.
- Choose **maps** if you frequently need to look up a value by its key. This is also a fast operation.
- Choose **iterables** if you have large collections where you need to visit all the elements lazily.

Challenges

Before moving on, here are some challenges to test your knowledge of iterables. It's best if you try to solve them yourself, but solutions are available in the **challenge** folder for this chapter if you get stuck.

Challenge 1: Iterating by Hand

1. Create a list named `myList` and populate it with four values.
2. Use `myList.iterator` to access the iterator.
3. Manually step through the list using `moveNext` and print each value using `current`.

Challenge 2: Fibonacci to Infinity

Create a custom iterable collection that contains all the Fibonacci numbers. Add an optional constructor parameter that will stop iteration after the nth number.

Key Points

- Iterables are collections in which you can step through each element individually.
- Iterables are lazy, meaning no work is done to determine the collection elements until you ask for them.
- Finding `length` or `elementAt` may be slow because the iterable calculates them by stepping through the elements one by one.
- `List` and `Set` are iterable collections with additional features.
- A synchronous generator is a function that returns multiple values on demand.
- An `Iterable` uses an `Iterator` to determine the next element in the collection.

Where to Go From Here?

To explore more about collections and their methods in Dart, browse the contents of the `dart:collection` library. Also, check out *Data Structures & Algorithms in Dart* to learn how to build custom collections such as the following:

- **Stack**: a collection with a first-in-last-out (FILO) data structure.
- **Queue**: a collection with a first-in-first-out (FIFO) data structure.
- **Linked list**: a list where one element points to the next rather than using an indexing system.
- **Tree**: A collection where elements are arranged in a hierarchical parent-child relationship.

Solution: The answer to the sequence riddle at the beginning of the chapter is **S**, the first letter of the word *six*: **O**ne, **T**wo, **T**hree, **F**our, **F**ive, **S**ix.

16 Conclusion

Congratulations! You've reached the end of *Dart Apprentice: Fundamentals*. We hope you've enjoyed reading this book and that the skills you've acquired will help you in all your future Dart projects.

At this point, you've reached the upper-beginner level in programming. You understand essential programming concepts like variables, control flow, loops, functions, classes and collections. The second book in the series, *Dart Apprentice: Beyond the Basics*, will build on this foundation to move you to the intermediate level. Here are some topics that you'll learn:

- Advanced string manipulation using regular expressions.
- Functions as inputs and outputs of other functions.
- Object-oriented programming concepts like inheritance, abstraction and interfaces.
- Asynchronous programming with futures and streams.

Hope to see you there!

If you have any questions or comments as you work through this book, please stop by our forums at https://forums.kodeco.com and look for the particular forum category for this book.

Thank you again for purchasing this book. Your continued support is what makes the books, tutorials, videos and other things we do at kodeco.com possible. We truly appreciate it!

– The *Dart Apprentice: Fundamentals* team