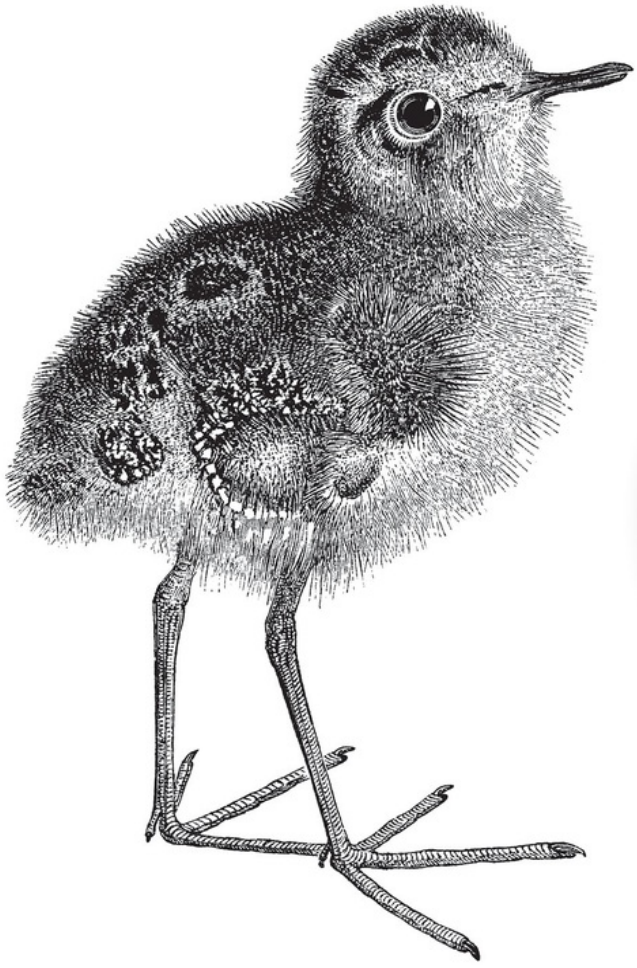
O'REILLY®

Flutter & Dart Cookbook

Developing Full-Stack Applications for the Cloud



Early Release

RAW & UNEDITED

Rich Rose

Flutter and Dart Cookbook

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Rich Rose

Flutter and Dart Cookbook

by Rich Rose

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Acquisitions Editor: Suzanne McQuade
- Development Editor: Jeff Bleiel
- Production Editor: Jonathon Owen
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea

Revision History for the Early Release

- 2022-03-24 First Release
- 2022-05-05 Second Release
- 2022-06-30 Third Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098119515 for release details.

978-1-098-11945-4

Chapter 1. Starting with Dart

Building beautiful, multi-platform applications has never been easy. The only constant has been the steep learning curve to understand and implement the desired design. Enter Dart and Flutter from Google. The combination of the Dart language and Flutter framework has established a synthesis between function, style, and ease of use.

In this chapter we begin our journey by learning Dart fundamentals. Dart is a feature-rich language that provides variables, data handling, control flow and much more. Experience with languages such as JavaScript, Python and C will mean your transition to Dart should not be difficult. If you are new to programming, I believe you will be pleasantly surprised how quickly you can produce an application.

The Dart and Flutter teams have provided a comprehensive treasure trove of tutorials. Even better, the community working in these technologies have raised the bar for immersive solutions and demonstrations to quickly get you up to speed.

Over the course of this chapter, you will learn how to install the Dart software development kit (SDK). Multiple options exist to run a Dart environment so I will walk you through the most common options e.g. DartPad, Android Studio and VS Code).

1.1 Determining which Dart installation to use

Problem

You are unsure how to access the Dart software development kit (SDK).

Solution

Identify the platform on which you wish to use the Dart SDK.

Discussion

Dart can be used directly from the browser in a pre-defined environment such as https://dartpad.dev.

Using Dart from the browser will be sufficient for a large number of situations. If you intend to build a modest application and do not require external dependencies e.g. graphics, files etc the browser environment will be aligned with your use case.

If you already have an IDE environment such as VS Code or Android Studio, use a plugin to add support for Dart/Flutter development.

Alternatively if you have the hardware to support it, the Dart SDK can be installed locally on your device. In this instance, there are a number of steps to fulfill prior to being able to start developing with Dart. Over the course of this chapter the main options are discussed in further detail to get you started. Personal preference will play a major part in the choice of installation followed.

1.2 Running Dart in DartPad

Problem

You want to develop with the Dart SDK without installing additional software.

Solution

Dart provides an online environment to test and run code. DartPad is an excellent online editor that can be used through a browser to develop and test your code. By using DartPad, you can quickly develop code and share that with a selected audience e.g. dart.dev or flutter.dev. Go to https://dartpad.dev and start to enter your code there.

Discussion

DartPad is a really powerful tool that you should consider irrespective of which type of installation you have selected. Figure 1-1 shows how a simple “Hello World” application would look in the DartPad interface.

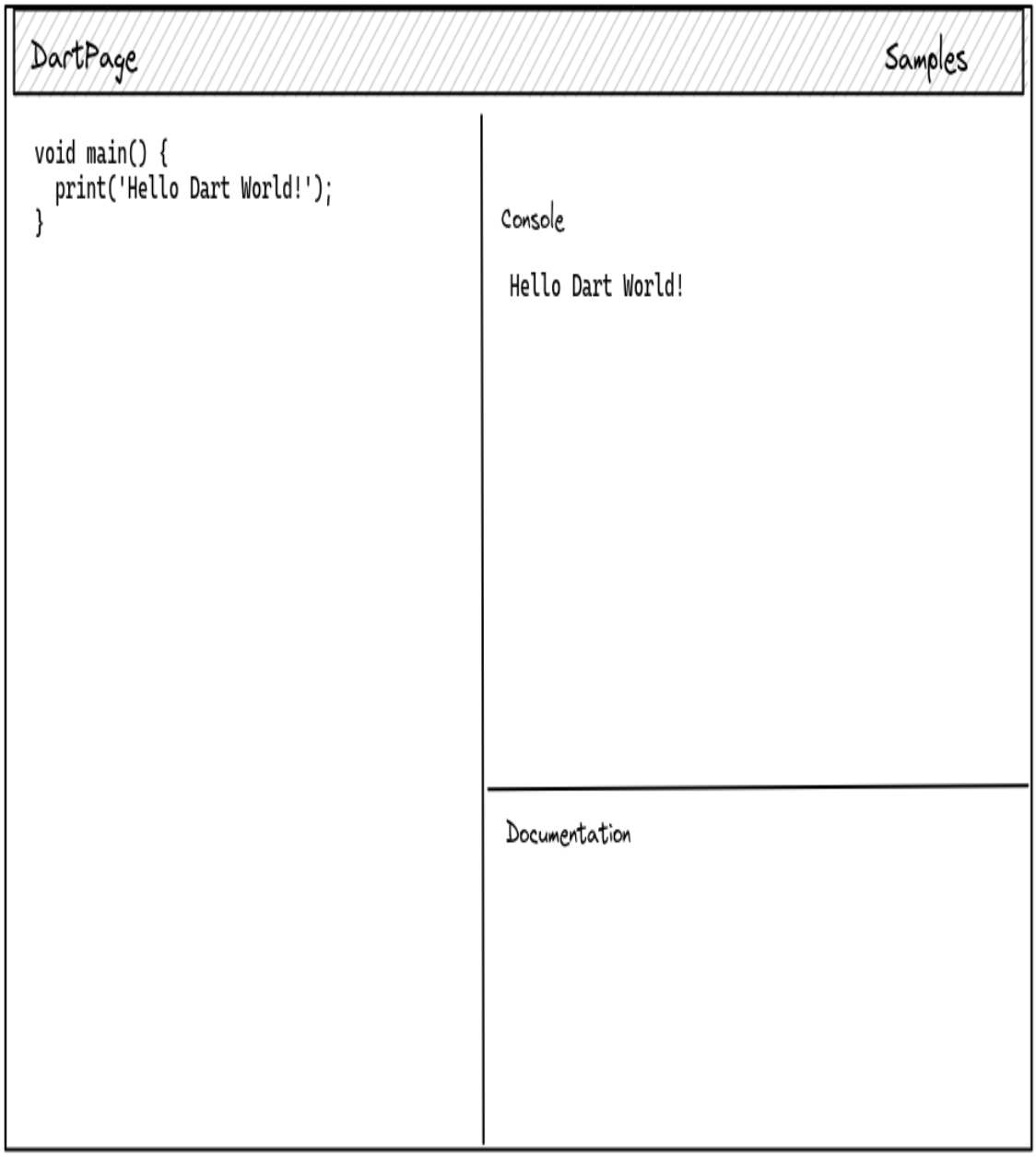


Figure 1-1. The DartPad interface

Starting DartPad presents a new instance ready to develop. The interface presents a simple editor environment with a code editor, a console and documentation windows. The code editor on the left hand side enables you to start coding in an intelligent window. The editor constantly provides feedback on helpful information for you to develop your code.

Once your code is ready to run, the output will be displayed in the console window. In the above example our program output is shown. Note: DartPad also works with Flutter applications, so you get the best of both worlds.

Another helpful feature is that there are a number of predefined sample applications available that cover both Dart and Flutter. If you want to try the Dart language you will find the samples will quickly demonstrate a variety of use cases. The typical use case for DartPad is to quickly write small sections of code that can be publicly shared using GitHub gists. When using this approach the rendered output can then be accessed via a unique URL associated with the gist.

1.3 Extending Android Studio to support Dart

Problem

You want to use Android Studio to build Dart applications.

Solution

Use Android Studio as an integrated developer environment (IDE).

To use Android Studio, you can simply select the Flutter or Dart plugin. Doing this will add the desired functionality to the development environment. Once selected, Android Studio will then go about configuring your environment with defaults to use the necessary plugin.

Discussion

The main use case for Android Studio is developing Android based applications. Adding support for Flutter and Dart is relatively easy via the application interface. Instructions for Android Studio are documented at https://docs.flutter.dev/development/tools/android-studio#installation-and-setup

Despite the name, Android Studio can actually work with other languages. The process of adding Flutter is available at the click of a button and is fully integrated into the environment.

Once installed Dart/Flutter applications can be selected as the target platform from project initiation. To get started, Android Studio will offer access to Flutter templates as part of the user interface.

1.4 Developing with VS Code

Problem

You want to use VS Code to build Dart/Flutter applications.

Solution

Use VS Code as an integrated developer environment or IDE. If you choose to work in this environment, adding Dart/Flutter is just a case of adding the relevant extension. Instructions for VS Code are documented at https://docs.flutter.dev/development/tools/vs-code

From within the editor, select the extension icon and search for Flutter. Click the install icon to add both Flutter and Dart functionality.

Discussion

VS Code provides a low barrier to entry when working with Dart/Flutter. The maintainers of this extension assume you will want both Dart and Flutter, so the installation process invokes both.

Once the extensions are installed, you will be able to create a Dart application within the environment. At this point, there will be some additional elements added to the user interface. Specifically the ability to run Dart directly from within the editor. In addition, VS Code will show information based on the user context. When code is being developed, the IDE will check on the validity of that code automatically. The code editor

will also render run/debug icons dynamically when code compiles without errors. When you graduate to using Flutter, VS Code will seek to target a particular device (e.g. web, android, iOS etc) based on your operating system setup.

If you are already working within this environment, this will be a no brainer to add this extension. In terms of updates, the environment will automatically indicate when a new version is available and allow the update to take place.

To uninstall the extension, you would select it and then choose the uninstall option.

1.5 Installing the Dart SDK

Problem

You want to use the Dart software development kit (SDK) with another Editor e.g. Emacs/VIM.

Solution

To develop using the Dart SDK outside of an IDE, will typically require you to install the software. You can quickly gain access to by following the instructions at https://dart.dev/get-dart. The SDK supports Linux (e.g. Debian and Ubuntu), MacOS and Windows platforms. Up to date instructions on the installation of the Dart SDK are available via the dart.dev site.

Discussion

Performing an SDK installation directly to the operating system means the SDK can be made accessible from any software running on your device. In general performing an installation outside of an IDE is a more sophisticated process. If your favorite editor is Vim or Emacs and you want to develop with Dart, then this recipe is likely how you would want to proceed. If you

have another preference then you will need to investigate how best to integrate with the Dart SDK.

For the majority of folks, using Android Studio or VS Code with plugins may be preferable. Reference this recipe if you are wanting to directly control the dart and flutter packages on a device.

1.6 Running a Dart application

Problem

You want to run a dart application.

Solution

Once the SDK is correctly installed, Dart code can be run within your environment. Open a terminal session to allow the entry of commands. If you are using an IDE, the terminal needs to be opened within that application. Now confirm that Dart is installed on the device by checking the version as shown below:

```
dart --version
```

If the command responds successfully, you should see the version of the SDK installed as well as the platform you are running on. If the command is unsuccessful, you will need to check the installation and path for your device.

Now create a new file named `main.dart` and add the following contents:

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

Run your example code from the command line as below:

```
dart main.dart
```

The above command should output the ‘Hello, Dart World!’

Discussion

The dart command is available as part of the Dart SDK installation. In the above example, the command will run a file named main.dart. Dart applications have the extension `.dart` and can be run either from the command line or within an IDE (e.g. Android Studio or VS Code). Note: Neither Android Studio or VS Code are pre-configured to include Dart/Flutter functionality. You will need to install the relevant plugin prior to being able to run any code.

If you don't want to install Dart within your environment use the online editor available at https://dartpad.dev/.

If you are unable to run the dart command, it's likely that the SDK has not been installed correctly. Use the latest installation instructions available at https://dart.dev/get-dart to confirm the installation on your device.

1.7 Selecting a release channel

Problem

You need to run against a specific version of the SDK.

Solution

Install the Dart SDK, which is compatible with Windows, MacOS and Linux (e.g. Debian and Ubuntu). Up to date instructions on the installation of the Dart SDK are available via the dart.dev site.

Dart works with the operating system and enables code to be run via the command line or an editor plugin.

Discussion

Release channels provide a mechanism to build code against a specific version of the Dart SDK. The channels are:

Channel	Description

Stable	This channel is meant for production and is updated on a quarterly basis.
Beta	This channel is meant for working with leading edge updates on a monthly basis.
Dev	This channel is meant for bleeding edge with updates on a weekly basis.

Based on the above, I would suggest that you should be on the stable channel for the majority of use cases. Unless you have a very good reason to use another release channel, stable should be where you do the majority of your development.

Chapter 2. Learning Dart Variables

In this chapter, we focus on learning the basics of using variables in Dart. As you might expect Dart offers a rich set of variable data types. To quickly get up to speed in the language, it is vitally important to know the basic data types.

If you are familiar with the use of variables in other programming languages, understanding variables in Dart should not be too difficult to grasp. Use this chapter as a quick guide to cement your understanding, before moving on to move complex topics.

For beginners this chapter will introduce you to the fundamentals. Ultimately it should offer a quick technical guide as you progress in your journey to learn Dart/Flutter.

Across the chapter, the code examples are self contained and are focused on a typical use case. We start by discussing the four main variable types (i.e. int, double, bool and String) and how each is used. Finally we learn how to let Dart to know what we want to do with our variables (i.e. Final, const and null).

As of Dart 2.0 the language is type-safe, meaning that once a variable is declared the type cannot be changed. For example if a variable of type double is declared, it cannot then be used as an int without explicit casting.

2.1 Declaring an Integer variable

Problem

You want to store a number without a decimal point.

Solution

Use an integer variable to store a number without a decimal point.

If you want to store an integer value of 35, the declaration would be as follows:

```
int     demoIntegerValue = 35;
```

In the Dart language an integer uses the reference int In the above code example, the data type e.g. int is the first part of the declaration. Next a label is assigned to the data type to be used e.g. demoIntegerValue. Finally we assign a value to the data type, in this example the value of 35.

Discussion

In the above example you begin by indicating the data type to be used, in this case. int. Following that, we provide a variable name for the data type, which in our example is demoIntegerValue. Finally we can optionally assign a value to the named variable.

If a value is not assigned to the int variable, then the value cannot be accessed. In this case, you would need to tell Dart how to handle this situation. Reference recipe 2.8 working with Null.

The typical use case is a number that doesn't require a decimal point (i.e. precision). An integer is defined as a 64 bit integer number. The integer data type is a subtype of num which includes basic operations e.g. +/- etc.

2.2 Declaring a Double variable

Problem

You want to store a number with a decimal point.

Solution

Use a double variable to store a number including a decimal point.

If we want to store a double value of 2.99, declare the following:

```
double demoDoubleValue  = 2.99;
```

Similar to other variables, prefix the variable with the desired data type e.g. double. The variable will then require a label to be assigned e.g. demoDoubleValue. Finally assign a value to the data type, in this example the value of 2.99.

Discussion

In the above example you begin by indicating the data type to be used, i.e. double. Following that we provide a variable name (in our example, demoDoubleValue) for double. The last part is optional, where we assign a value to the named variable.

The typical use case for a double data type is a number requiring a level of precision. A double data type is a 64 bit floating point number. Double is a subtype of num which includes basic operations e.g. +/- etc.

2.3 Declaring a Bool variable

Problem

You want to store a true/false value.

Solution

Use a bool variable to store a true/false state.

Declare a Boolean variable using the keyword bool. Following the data type declaration with a label for the variable name i.e. demoBoolValue. Finally assign a value to the variable of either true or false.

Here's an example of a how to declare a bool:

```
bool    demoBoolValue     = true;
```

Discussion

In the above example you begin by indicating the data type to be used, i.e. bool. Following that we provide a variable name for the defined data type. The last part is optional, where we assign a value to the named variable.

The use case for a bool is that of a true/false scenario. Note that true and false are reserved words in Dart. A boolean data type includes logic operations e.g. and/equality/inclusive or/exclusive or.

2.4 Declaring a String variable

Problem

You want to store a sequence of characters.

Solution

Use a String variable to store a series of text.

Here's an example of a how to declare a String:

```
String demoStringValue  = 'I am a string';
```

Discussion

In the above example you begin by indicating the data type to be used, i.e. String. Following that we provide a variable name for the defined data type. The last part is optional, where we assign a value to the named variable.

The typical use case for a String data type is collection of text. A String data type uses UTF-16 code units. String is used to represent text characters but due to encoding can also support an extended range of characters e.g. emojis.

When using a string, you can use either a matching single or double quotes to identify the text to be displayed. If you require a multiline text, this can be achieved using triple quotes.

2.5 Using a Print statement

Problem

You want to display programmatic output from a Dart application.

Solution

Use a print statement to display information from an application. The print statement can display both static (i.e. a string literal) and variable content.

Example 2-5. Example 1: Here's an example of a how to print static content:

```
void main() {

  print('Hello World!');
}
```

Example 2-6. Example 2: Here's an example of a how print the content of a variable:

```
void main() {
  int intVariable  = 10;
  var boolVariable = true;

  print(intVariable);
  print('$intVariable');
  print('The bool variable is $boolVariable');
}
```

Example 2-7. Example 3: Here's an example of a how print the complex data type:

```
import 'dart:convert';
void main() {
  // Create JSON value
  Map<String, dynamic> data = {
    jsonEncode('title'): json.encode('Star Wars'),
    jsonEncode('year'): json.encode(1979)
  };

  // Decode the JSON
  Map<String, dynamic>  items = json.decode(data.toString());

  print(items);
  print(items['title']);
  print("This is the title: $items['title']");
  print('This is the title: ${items['title']}');
}
```

Discussion

Use the `$` character to reference a variable in a print statement. Prefixing a variable with the `$` tells Dart that a variable is being used and it should replace this value.

The print statement is useful in a number of scenarios. Printing static content doesn't require any additional steps to display information. To use with static content enclose the value in quotes and the print statement will take care of the rest.

Printing a variable value will require the variable to be prefixed with the '$' sign. Where Dart is being used to print content, you can tell the language that you want the value to be displayed. In example 2 you see three common ways to reference a variable in a print statement.

Example 3 illustrates a use case, you will come across where the variable value needs a bit of help to be displayed correctly. Specifically in the last two print statements:

String interpolation without braces

```
  print("This is the title: $items['title']");
```

String interpolation with braces

```
  print('This is the title: ${items['title']}');
```

The above statements look equivalent, but they are not. In the first print statement Dart will interpret the variable to be displayed as items. To display the variable correctly, it actually requires that it be enclosed in braces, as per the second line. Doing this will ensure that the value associated with items [‘title’] is correctly interpreted by the print statement.

Dart will provide feedback on whether a brace is required, typically it is not. However, if you create a complex variable type, do check to ensure you are referencing the element desired.

2.6 Using a Const

Problem

You want to create a variable that cannot be changed

Solution

Use a const to create a variable whose value cannot be reassigned.

Here’s an example of using a const variable:

```
void main() {
  const daysInYear = 365;

  print ('There are $daysInYear days in a year');
}
```

Discussion

In Dart, Const represents a value that will not change. Where a value is not subject to change in an application, the Const keyword should be used. Const indicates that the variable represents immutable data.

2.7 Using Final

Problem

You want to create a variable that cannot be changed, but you will not know the value until runtime.

Solution

Use final to create a variable whose value cannot be reassigned. In contrast to a const variable, a final variable value is assigned at runtime.

Here's an example using a final variable:

```
void main() {
  final today = DateTime.now();
  print('Today is day ${today.weekday}');
}
```

Discussion

Final represents a value that needs to be determined at runtime and is not subject to change. The final keyword is used in situations where a value is derived at runtime (i.e. when the application is active). Again the value assigned is immutable, however unlike a const value it cannot be known at compile time.

In the code example, the day output by the print statement will be determined when the application is run, so it will display based on the actual weekday available on the host machine.

2.8 Working with Null

Problem

You want to assign a variable a default value of null.

Solution

Use null to apply a consistent value to a declared variable. Null is an interesting concept as it is meant to represent the absence of content. Typically a null value is used to initialize variables that do not have a

default value to be assigned. In this instance null can be used to represent a variable that has not explicitly been assigned a value.

Here's an example of how to declare a variable as null in Dart:

```
void main(){
  int? ten = null;
  print ('ten: $ten');

  ten = 10;
  print ('ten: $ten');
}
```

Discussion

Note: As of Dart v2.0 null type safety is now the default, meaning it is no longer possible to assign null to all data types.

In Dart, Null is also an object which means it can be used beyond the simple `no value` use case. To assign a null to a data type, it is expected that the ? type is appended to the data type to explicitly indicate a value can also be null.

More recent versions of the Dart SDK also require explicit acknowledgement of whether a data type is nullable or non nullable.

For further information consult the Null class reference in the dart API (i.e. https://api.dart.dev/stable/2.14.4/dart-core/Null-class.html)

Chapter 3. Exploring Control Flow

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at jbleiel@oreilly.com.

Control flow relates to the way instructions will be executed in an application. Typical logic flows exist such as conditional and looping flows used to determine the instructional processing order. Dart provides a number of methods to manage how the application operates and coordinates based on this decision flow.

If you have used other languages such as Python, JavaScript etc, then you will be very familiar with the content covered in this chapter. For those of you who are new to development, this chapter is super important! Control flow statements are common across most languages you will be exposed to. Part of learning a language is the ability to incorporate these types of statements.

In this chapter, you will learn how to use control flow to incorporate logic in your application. You'll also see use cases for each statement. Many of the flows include a condition statement which is used to dictate what actions are taken. Pay special attention to these conditions and look to efficiently use control flow within your application.

3.1 Using an If statement

Problem

You want to provide a logical check on a condition before executing an instruction.

Solution

Use an if statement, to provide a control statement for a binary option. An If statement provides a step to confirm that a logic statement is valid.

If there are multiple options consider using a `switch` statement. (reference recipe 3.4 Using a Switch statement)

This example shows how to use the if condition. The if statement is used to check the value of a bool variable. If the bool variable is set to `true` then the first message is displayed. If the bool variable is set to `false` an alternative message is displayed.

```
void main() {
  bool isFootball = true;
  if (isFootball) {
    print('Go Football!');
  } else {
    print('Go Sports!');
  }
}
```

Discussion

Working with an IF statement allows control over the logic progression within an application. Control flow of this type is essential to building applications and provides a simple mechanism to select between choices

Note: in the example, the `if` statement validation is implicit, meaning it is checking if the value assigned is true.

The typical use case for an `if` statement is to make a choice between two or more options. Ideally if you have two options, this type of control flow is ideal.

3.2 Using While/Do While

Problem

You want a method to loop until a condition is satisfied within an application.

Solution

Use a While loop when you need the entry condition to be validated at the start of the control flow. Note: the loop check is performed at the start of the loop condition. A While loop therefore has a minimum of zero iterations and a max iteration of N.

Here's an example of a while loop control flow:

```
void main() {
  bool isTrue = true;

  while (isTrue) {
    print ('Hello');
    isTrue = false;
  }
}
```

Use a Do While loop when you need the loop to be executed a minimum of one iteration. With this control structure the condition is validated at the end of each iteration.

Here's an example of a control flow: do while loop

```
void main() {
  bool isTrue = true;

  do {
    print ('Hello');
    isTrue = false;
  } while (isTrue) ;
}
```

Discussion

The key nuance to observe from these examples is the nature of execution and what that means for processing of the control flow.

In the while loop example, the demo application will only output a value when the bool variable is set to `true`. The do while loop example will output a print statement irrespective of the initial value of the isTrue variable.

A while loop will test a condition before executing the loop, meaning you can use this to perform 0..N iterations. A typical use case would be where a variable is used to control the number of iterations performed.

In a `do while` statement the typical use case would be where there is at least a single loop iteration. If the situation requires a single iteration then using this type of control flow is a good choice.

3.3 Using a For statement

Problem

You want a method to loop through a defined range of items.

Solution

Use a For statement to perform a defined number of iterations within a defined range. The specific range is determined as part of the initialisation of the `for` statement.

Here's an example of a for statement:

```
void main() {
  int maxIterations = 10;
  for (var i = 0; i < maxIterations; i++) {
    print ('Iteration: $i');
  }
}
```

In addition where you have an iterable object, you can also use forEach:

```
void main() {
  List daysOfWeek = ['Sunday', 'Monday', 'Tuesday'];
```

```
  daysOfWeek.forEach((print));
}
```

Discussion

A `for` statement can be used for a variety of use cases, such as performing an action an exact number of times (e.g. initializing variables).

As the second example shows, a forEach statement is a very useful technique to access information within an object. Where you have an iterable type (e.g. the List object), a forEach statement provides the ability to directly access the content. Appending the forEach to the List object, enables a shortcut in which a print statement can be directly attributed to each item in the list.

The typical use case for a `for` statement is to perform iterations where a range is defined. It can also be used to efficiently process a List or similar data type in an efficient manner.

3.4 Using a Switch statement

Problem

You want to perform multiple logical checks on a presented value.

Solution

Use a Switch statement where you have multiple logic statements. Typically where multiple logical checks are required the first control flow to come to mind might be an `if` statement (which we saw in Recipe 3.1). However it may be more efficient to use a `switch` statement.

Here's an example of a switch statement:

```
void main() {
  int myValue = 1;

  switch (myValue) {
    case 1: print('Monday');
```

```
      break;
    case 2: print('Tuesday');
      break;
    default:
      print('Error: Value not defined?');
      break;
  }
}
```

Discussion

A switch statement can present better readability than multiple `if` statements. In most cases where the requirements need a logical check, the switch statement may be a more efficient choice.

Note: In the above code, the switch statement has two valid choices i.e. 1 or 2. You can imagine expanding this code to incorporate more choices. In this instance, the switch statement will render the default statement where the relevant value has not been added. That is any other choice is sent to the default option which acts as a clean up section. Incorporating explicit statements is helpful to reduce errors in processing of information.

3.5 Using an Enum

Problem

You want to define a grouping of constant values to use within an application.

Solution

Use an enum (enumerator) to provide a grouping of information that is a consistent model for associated data.

Here's an example of declaring and printing the values associated with the enum:

```
enum Day { sun, mon, tues }
void main() {
```

```
  print('$Day.values');
}
```

Here's an example of declaring and printing the enum reference at index zero:

```
enum Day { sun, mon, tues }
void main() {
  print('${Day.values[0]}');
}
```

Here's an example of declaring and using a list assignment:

```
enum Day { sun, mon, tues }
void main() {
  List<Day> daysOfWeek = Day.values;
  print('${daysOfWeek[0]}');
}
```

Discussion

In the third example above, an enum is defined for the days of the week. When the print command is run, the debug output shows the values associated with the enum i.e. "sun", "mon", "tues". Enum is indexed, meaning each item declared has a value based on its position.

An enum (or enumeration) is used to define related items. Think of an enum as an ordered collection, for example days of the week, months of the year. In the examples the order can be transposed with the value e.g. the first month is January or the twelfth month is December.

The use of a list in the above example provides an opportunity to increase the readability of code.

3.6 Handling Exceptions

Problem

You want to provide a way to handle error processing within an application.

Solution

Use the `try`, `catch` and `finally` blocks to provide exception management in Dart.

Here's an example of how to handle exceptions in Dart:

```
void main(){
  String name = "Dart";

  try{
    print ('Name: $name');
    // The following line generates a RangeError
    name.indexOf(name[0], name.length - (name.length+2));
  } catch (exception) {
    print ('Exception: $exception');
  } finally {
    print ('Mission completed!');
  }
}
```

Discussion

The example code defines the relevant sections and sets up a String to hold the work `Dart`. To generate an exception the indexOf method is used with an invalid range (i.e. one greater than the length of the name String). An exception will then be displayed indicating a `RangeError`.

A try block is used for normal processing of code. The block of code will continue executing until an event indicates something abnormal is occurring.

A catch block is used to handle processing where an abnormal event occurs. Using a catch block provides an opportunity to safely recover or handle the event that took place.

A finally block is used to perform an action that should take place irrespective of whether code is successfully executed or generates an exception. Typically a finally block is used for clean up e.g. to close any open files etc. In addition it will output a message to indicate the processing has been completed irrespective of the exception occurring.

Exception management is certainly a type of control flow although not in the traditional sense. Adding exception management will become ever more important to your application as it increases in complexity.

Chapter 4. Implementing Functions

In this chapter, we will move beyond the fundamentals of Dart, and introduce functions. As you may have noticed we have already used a number of functions already (e.g. main and print). During this chapter we will explore the main use cases for using functions.

Building more complex applications will certainly require developers to progress beyond simple constructs. At a minimum an awareness of some essential concepts and algorithms is desirable. Over the course of this chapter learn the foundations of code isolation.

The main use cases for functions is to group instructions. The chapter begins by illustrating how to define a basic function without parameters or a return value. In most situations, this is not the pattern you will want to use. However for learning purposes it has been included. Beyond this you will be introduced to parameters and return values. At this point, hopefully it will become clearer why adding parameters and return values is so powerful and the desired pattern to follow.

Towards the end of the chapter you will see examples of other ways to use functions. You'll discover that, as your skills grow as a developer, so will your use of functions.

4.1 Declaring Functions

Problem

You want a common group name for instructions that perform a specific task.

Solution

Declare a function. In the following example, the getCurrentDateTime function is used to print out a date/time value.

```
void main() {
  getCurrentDateTime();
}
void getCurrentDateTime() {
  var timeLondon  = DateTime.now();
  print('London:    $timeLondon');
}
```

Discussion

In the above example a function named getCurrentDateTime is defined. Note that the function is declared as requiring no parameters, and a void return.

Here the function's only job is to print out the current date and time.

In the real world, it doesn't have input parameters or a return value, which means this type of function has limited use.

4.2 Adding parameters to Functions

Problem

You want to pass variable information to a function.

Solution

Use a parameter to pass information to a function. In the following example, a parameter is provided to a function as used as part of the control flow

```
void main() {
  getCurrentDateTime(-7);
}
void getCurrentDateTime(int hourDifference) {
  var timeNow  = DateTime.now();
  var timeDifference = timeNow.add(Duration(hours:
hourDifference));

  print('London:    $timeNow');
  print('New York:  $timeDifference');
}
```

Discussion

In the above example, the parameter provided to the function is used to determine an action. The function is used to determine the time in New York by using the current time in London.

Given we have added a parameter value to the function, we can now state the number of hours difference required. Doing this has made the function more applicable to a wider series of use cases. However because the function doesn't return the value, the function isn't as flexible as it could be.

Using parameters enhances the flexibility of a function by adding a variable. The addition of variables to the function signature makes the function more general in nature. Creating generalized functions in this way is a good approach to reduce the amount of code that needs to be created for a task.

4.3 Returning values from Functions

Problem

You want a common group name for instructions that returns a computed value.

Solution

Use a named function that computes a value and return this to the calling method. is a common mechanism for grouping instructions together.

Here's an example of declaring a function that returns a value:

```
void main() {
  DateTime timeLondon  = getCurrentDateTime(0);
  DateTime timeNewYork = getCurrentDateTime(-7);

  print('London:    $timeLondon');
  print('New York:  $timeNewYork');
}
DateTime getCurrentDateTime(int hourDifference) {
  DateTime timeNow  = DateTime.now();
  DateTime timeDifference = timeNow.add(Duration(hours:
hourDifference));

  return timeDifference;
}
```

Discussion

In the above example the function named getCurrentDateTime is enhanced to return a value. The function is declared to accept parameters and return a value. Now we have a more generic function that can be utilized in a wider series of settings.

In this instance, the function accepting parameters means you are able to provide different hour values. The function only knows that it should accept an int value representing the number of hours to be used. From the example we see we make two calls to the function to initialize the London time with a zero hours difference, followed by New York with negative 7 hours.

The return value from the getCurrentDateTime function presents a Datetime object. By capturing the return value, you can output the relative date time combination.

Note how we have reused the function to be invoked with an integer parameter and then return a DateTime object. Dart provides the opportunity to create simple functionality like this to help with your development. Having a rich set of methods associated with the class objects like DateTime saves an enormous amount of development time.

4.4 Declaring Anonymous functions

Problem

You want to enclose an expression within a function.

Solution

Declare an anonymous function to perform a simple expression. Often a function only requires a single expression, in which case an anonymous function can provide an elegant solution.

Here's an example of how to use an anonymous function:

```
void main() {
  int value = 5;
  int intSquared(value) => value * value;
  int intCubed(value) => value * value * value;
  print('$value squared is ${intSquared(value)}');
  print('$value cubed is ${intCubed(value)}');
}
```

Discussion

In the example, a function to square a number is required. The algorithm requires the input to be multiplied without any additional steps. In this case, an anonymous function provides a good way to simplify the declaration.

Anonymous functions use the `=>` to indicate a function. Prior to the function, a variable is declared to hold the result from the function. Note: the variable can include parameters by adding these within the bracket declaration. In the example, the anonymous function accepts an integer and this is used within the function declaration. An explicit return is used to

assign the value of the function back to the variable (i.e. the result of multiplier * multiplier is stored in the variable intSquared).

4.5 Using optional parameters

Problem

You want to vary the number of parameters to a function.

Solution

Provide an optional parameter. Dart supports optional parameters that enable values to be omitted. Two distinct types of optional parameters are available i.e. Named and Positional.

Here's an example of how to use named parameters:

```
void main() {
  printGreetingNamed();
  printGreetingNamed(personName: "Rich");
  printGreetingNamed(personName: "Mary", clientId: 001);
}
void printGreetingNamed({String personName = 'Stranger',
                         int clientId = 999}){
  if (personName.contains('Stranger')) {
    print('Employee: $clientId Stranger danger ');
  } else {
    print('Employee: $clientId $personName ');
  }
}
```

Here's an example of how to use positional parameters:

```
void main() {
  printGreetingPositional("Rich");
  printGreetingPositional("Rich", "Rose");
}
void printGreetingPositional(String personName, [String?
personSurname]){
  print(personName);
  if (personSurname != null){
    print(personSurname);
  }
}
```

Discussion

Dart provides additional flexibility for the use of parameters to functions. Where parameters can be omitted it can be useful to consider the use of optional parameters.

Named parameters provide the ability to include named variables within the function declaration. To use this type of parameter, include braces to define the necessary values to be presented. In the first example optional parameters are used to pass across a name and clientId. If the function is not supplied with the information it will still operate as expected by defaulting to a value. Default values can be supplied if it is necessary to provide a value and perform specific logic e.g. “int clientId = 999”.

Positional parameters perform similar to normal parameters with the flexibility to be omitted as necessary. In the example the second parameter is defined as a positional parameter using the square brackets. Additionally Dart allows the variable to be defined as a potentially null value by the inclusion of the `?` character.

Both named and positional parameters offer increased flexibility to your functions. You can use them in a variety of scenarios where parameters are needed (for example, in a person object where firstname and surname are mandatory, but the middle name is optional).

Chapter 5. Handling Maps and Lists

In this section the fundamentals of data handling is outlined. The aim of this chapter is to cover Lists and Maps that are used to provide foundational data structures for information handling in Dart.

If you are lucky enough to be familiar with other languages, then many of the concepts presented should be familiar. However if this is the first time seeing these techniques, the example pieces of code are self contained. It would be helpful to run and experiment with the examples to gain a feel of the workings of the language. Over the course of this chapter learn how to utilize maps and lists within your application.

5.1 Using a Map to handle objects

Problem

You want to handle a key/value pair in a Dart application.

Solution

Use a Map to handle key/value objects of any type. Map keys are required to be unique as they act as the index to access Map values. Map values are not required to be unique and can be duplicated as required.

Here's an example of how to declare a Map in Dart:

```
void main() {
  Map<int, dynamic> movieInformation = {0: 'Star wars'};

  // Add element to Map
  movieInformation[1] = 'Empire Strikes Back';
  movieInformation[2] = 'Return of the Jedi';

  // Loop through Map
  movieInformation.keys.forEach(
    (k) => print('Key: $k, Value: ${movieInformation[k]}')
  );

  // Show the keys
  print('Keys: ${movieInformation.keys}');

  // Show the values
  print('Values: ${movieInformation.values}');

  // Remove element from map using the key
  movieInformation.remove(0);

  // Loop through Map
  movieInformation.keys.forEach(
    (k) => print('Key: $k, Value: ${movieInformation[k]}')
  );

  // Check if a key exists
  print('Key 1 exists: ${movieInformation.containsKey(1)}');
  print('Value Star Wars exists:
${movieInformation.containsValue('Star Wars')}');
}
```

Discussion

In the code example, a Map is used to define a collection of data based on series order and film name. A construct of this type is very useful when combining pieces of information together.

Declaration of the Map follows a standard variable declaration. Note: Map is actually a function call so requires braces. To populate the Map, define a

key (e.g. movieInformation[0]) and then assign a value to the key (e.g. Star Wars). The assignment of values can be made in any order, just be careful not to duplicate the keys.

To access the information within the Map, use the Map function. Map has a number of methods available that can be used to access the associated data items held within the Map. In this instance to loop through each item, the key is used to access the forEach method. An anonymous function is then used to print the details of each movie stored in the map.

5.2 Retrieving Map content

Problem

You want to assign a Map value to a variable

Solution

Use a variable in conjunction with a Map to reference an indexed item. Map values are referenced as key/value combinations which can be assigned to a variable for easier access.

Here's an example of retrieving a Map value and assigning it to a variable with Dart:

```
void main() {
  Map starWars = {"title": "Star Wars", "year": 1977};
  Map theEmpireStrikesBack = {"title": "The Empire Strikes Back",
"year": 1980};
  Map listFilms = {"first": starWars, "second":
theEmpireStrikesBack};
  Map currentFilm = listFilms['first'];
  String title  = currentFilm['title'];
  int    year   = currentFilm['year'];
  print (title);
  print (year);
}
```

Discussion

In the code example, the value within the Map structure is accessed via its key,and the key is ‘title’. When a value is required, we provide the key to index the map to retrieve the desired value.

You can see in the above code that where we have a more complex data construct being able to dereference a variable reduces complexity. In the example code, listFilms is a complex data type in which we assign a key e.g. ‘first’ and the value is represented by another Map. To uniquely reference a Map value we again use the key. A more convenient method to access a Map value is shown with the title and year variables. Now rather than accessing the Map object, you can use the direct data type to perform an additional action e.g. print the variable value.

5.3 Validating key existence within a Map

Problem

You want to confirm a key exists in a Map

Solution

Use the indexing functionality of a Map to identify if a key explicitly exists.

Here’s an example of validating that a key exists in a Map with Dart:

```
void main() {
  Map starWars = {"title": "Star Wars", "year": 1977};
  Map theEmpireStrikesBack = {"title": "The Empire Strikes Back",
"year": 1980};
  Map listFilms = {"first": starWars, "second":
theEmpireStrikesBack};
  if (listFilms['first']==null) {
    print ('Key does not exist');
  } else {
    print ('Key exists!');
  }
}
```

Discussion

Maps are indexed using key values, so validating the existence of a key can quickly be performed. To find a key, use the required key to index the Map.

In the example the Map will return a null value where the key is not present in the Map. If the key exists in the Map, the information returned will be the value associated with the key.

5.4 Working with Lists

Problem

You want a way to use an array of values within a Dart application.

Solution

Use a list to organize objects as an ordered collection. A List represents an array object that can hold information. It provides a simple construct that uses a zero indexed grouping of elements.

Here's an example of how to use a List in Dart:

```
void main() {
  List listMonths = ['January', 'February', 'March'];
  listMonths.forEach(print);
}
```

Discussion

Lists are very versatile and can be used in a variety of circumstances. In the above example, a list is used to hold the months of the year. The List declaration is used to hold a String, but it can actually hold a variety of data types making this object extremely flexible.

A List is denoted by using square start and end brackets that indicate the values are defined within the square braces. The length of the List is available as a method which identifies how many elements have been declared. Note the list is indexed from zero so if you intend to manually access elements, you will need to use zero if you want the first element.

Another nice feature of Lists is that it includes a range of methods to handle processing information. In the example, the forEach method is used to perform a print of the elements contained in the list.

5.5 Adding List content

Problem

You want to add new content to an existing List

Solution

Use the list add method to incorporate new content into a List. Lists support the dynamic addition of new elements and can be expanded as required.

Here's an example of how to add a List element in Dart:

```
void main() {
  List listMonths = ['January', 'February', 'March'];
  listMonths.add('April');
  listMonths.forEach(print);
}
```

Discussion

In the above example, a List is defined with three elements. If you want to expand the number of elements, this can be done by using the add method. The add method will append the new element at the end of the List. Therefore in the example you would see the month's output as 'January', 'February', 'March', 'April'.

The dynamic nature of a List makes it perfect for multiple situations where data structure manipulation is required. You will see Lists used across a number of situations to handle a variety of data types.

5.6 Using Lists with complex types

Problem

You want to make an array of complex data types.

Solution

Use Lists to organize other data. Lists can be especially useful for handling other data structures such as Maps.

Here's an example of how to use a List with complex data types in Dart:

```
void main() {
  Map<String, dynamic> filmStarWars = {"title": "Star Wars",
                                       "year": 1977};
  Map<String, dynamic> filmEmpire   = {"title": "The Empire Strikes
Back",
                                       "year": 1980};
  Map<String, dynamic> filmJedi     = {"title": "The Return of the
Jedi",
                                       "year": 1983};
  List listFilms = [filmStarWars, filmEmpire, filmJedi];
  Map<String, dynamic> currentFilm = listFilms[0];
  print(currentFilm);
  print(currentFilm['title']);
}
```

Discussion

In the example, film data is added to a Map that encloses title and year information. Here we use a List to manage the individual Maps, so the individual elements can be combined. The resultant List provides a convenient data structure for accessing the information to be stored.

To access the information you need to dereference the variable. The listFilms[0] means to access the first element in the list. As each element is a Map, you now have the data associated with this. Use the dereferenced value to store in a new variable currentFilm, which can be accessed directly or with a key.

Using Lists can provide an elegant method to access complex data types in a consistent manner. If you need to coordinate data types, consider using a List to make this process more manageable.

Chapter 6. Leveraging Classes

In this chapter, we introduce Classes and how these can be used together with Dart. Over the course of the chapter you will explore both declaration and extension of objects. These techniques are important and will provide a good reference as your Dart skills increase over time.

The chapter covers how to incorporate a Class which represents a basic requirement for object oriented programming. In addition we also cover the need for constructors, extending classes and using multiple classes together.

As your development becomes more sophisticated, you will be able to utilize custom classes to achieve your requirements. Becoming efficient with Classes is a steep learning curve, so take small steps. Overtime, you will naturally improve and be able to incorporate very complex subject matter into your general solutions.

6.1 Defining Classes

Problem

You want to create an object that allows both functions and variables to be invoked via a common method.

Solution

Use a class to collate information into a new object providing both variable storage and functionality to process information. Here's an example of how to declare a class in Dart:

```
const numDays = 7;
class DaysLeftInWeek {
  int currentDay = DateTime.now().weekday.toInt();
  int howManyDaysLeft(){
    return numDays - currentDay;
  }
}
void main() {
  var currentDay = DaysLeftInWeek();

  print ('Today is day ${currentDay.currentDay}');
  print ('We have ${currentDay.howManyDaysLeft()} day(s) left in
the week');
}
```

Discussion

Dart is an object oriented language and has the Object class for all Dart Objects except null. The result is that a non-nullable object is a subclass of Object.

In the example the class is used to determine how many days are left in the week. The declaration uses `class` to denote the definition that follows including elements for both variables and functions.

To use the class, declare a variable e.g. currentDay to instantiate the class. Now the variable currentDay is able to access both the variable and functions associated with the class. The print statements demonstrate how to access a variable and a function to access the underlying data.

6.2 Using Class Constructors

Problem

You want to initialize defaults within the class.

Solution

Use a class constructor to perform initialization of the object instance. The initialization can be used to set sensible defaults to class values.

Here's an example[1] of how to declare and use a Class constructor:

```
const numDays = 7;
class DaysLeftInWeek {
  int currentDay = 0;

  DaysLeftInWeek(){
    currentDay = DateTime.now().weekday.toInt();
  }

  int howManyDaysLeft(){
    return numDays - currentDay;
  }
}
void main() {
  var currentDay = DaysLeftInWeek();

  print ('Today is day ${currentDay.currentDay}');
  print ('We have ${currentDay.howManyDaysLeft()} day(s) left in
the week');
}
```

Discussion

In the example the class is used to determine how many months are left in the year. The declaration uses `class` to denote the definition that follows includes elements for both variables and functions.

Within the class DaysLeftInWeek, note there is a function defined with the same name as the class. A constructor takes the same name as the class and will be called on instantiation of the class. In this instance the constructor sets the class variable currentDay with the value of today's date.

To instantiate the class, in the main function, a variable currentDay is declared. The variable is now set to the class construct and has access to both the variables and associated methods of the DaysLeftInweek class.

6.3 Extending Classes

Problem

You want to enhance an existing class by introducing additional functionality.

Solution

Use a class with extends to incorporate new functionalities. When using extends, this creates a subclass for existing class functionality. As an object oriented language Dart provides extensive class support in each new release.

Here's an example of how to extend a class in Dart:

```
class Media {
  String title = "";
  String type = "";

  Media(){ type = "Class"; }

  void setMediaTitle(String mediaTitle){ title = mediaTitle; }

  String getMediaTitle(){ return title; }

  String getMediaType(){ return type; }
}
class Book extends Media {
  String author = "";
  String isbn = "";

  Book(){ type = "Subclass"; }

  void setBookTitle(String bookTitle){ title = bookTitle; }

  void setBookAuthor(String bookAuthor){ author = bookAuthor; }

  void setBookISBN(String bookISBN){ isbn = bookISBN; }

  String getBookTitle(){ return title; }

  String getBookAuthor(){ return author; }

  String getBookISBN(){ return isbn; }
}
void main() {
  var myMedia = Media();
```

```
  myMedia.setMediaTitle('Tron');
  print ('Title: ${myMedia.getMediaTitle()}');
  print ('Type: ${myMedia.getMediaType()}');


  var myBook = Book();
  myBook.setBookTitle("Jungle Book");
  myBook.setBookAuthor("R Kipling");
  print ('Title: ${myBook.getMediaTitle()}');
  print ('Author: ${myBook.getBookAuthor()}');
  print ('Type: ${myBook.getMediaType()}');
}
```

Discussion

When using a subclass it is possible to override existing class functionality e.g. methods, etc.

In the code example, the Media class is extended through the Book subclass. The book subclass extends the Media class meaning it can access the methods and variables instantiated within it.

Using extends is a useful approach where there are similar data structures available that potentially need slightly different methods. In the example the Media class is a generic abstraction that is set up to hold base information. The Book class is a specialization of the Media class offering the ability to add book specific information.

6.4 Extending Classes with Mixins

Problem

You want to extend an existing class with functionality from multiple class hierarchies.

Solution

Use `mixins` when requiring functionality from multiple classes. Mixins are a powerful tool when working with classes and allow information to be incorporated from multiple classes.

Here’s an example of how to use a Mixin:

```
abstract class SnickersOriginal {
  bool hasHazelnut = true;
  bool hasRice = false;
  bool hasAlmond = false;
}
abstract class SnickersCrisp {
  bool hasHazelnut = true;
  bool hasRice = true;
  bool hasAlmond = false;
}
class ChocolateBar {
  bool hasChocolate = true;
}
class CandyBar extends ChocolateBar with SnickersOriginal {
  List<String> ingredients = [];

  CandyBar(){
    if (hasChocolate){
      ingredients.add('Chocolate');
    }
    if (hasHazelnut){
      ingredients.add('Hazelnut');
    }
    if (hasRice){
      ingredients.add('Hazelnut');
    }
    if (hasAlmond){
      ingredients.add('Almonds');
    }
  }

  List<String> getIngredients(){
    return ingredients;
  }
}
void main() {
  var snickersOriginal = CandyBar();
  print ('Ingredients:');
  snickersOriginal.getIngredients().forEach((ingredient) =>
print(ingredient));
}
```

Discussion

Mixins require the use of the `with` keyword. The base class should not override the default constructor.

Define an abstract class to include the relevant markers for the object to be created. In the example, the class denotes the key ingredients of a candy bar. In addition a chocolate bar class is created that can be used to hold the specifics of the object.

Using a mixin allows the subclass object to incorporate a lot more functionality without having to write specific code. In the example, a combination of multiple classes is used to define the nature of the subclass. The general ingredients can then be listed by validating the general ingredients for the candy bar.

6.5 Importing a package

Problem

You want to incorporate functionality derived from a library.

Solution

Use a package to incorporate pre-existing functionality into a Dart application.

Import statements enable external packages to be used within a Dart application. To utilize a package within an application, use the `import` statement to include the library.

Dart has a feature-rich set of libraries, which are packages of code for a particular task. These libraries are published on sites such as https://pub.dev/. Use the package repository to find and import packages for a specific task to reduce development time.

Here's an example to use an import in Dart:

```
import 'dart:math';
void main() {
  // Generate random number between 0-9
  int seed = Random().nextInt(10);

  print ('Seed: $seed');
}
```

Discussion

In the above example code, we are importing the dart:math library. Dart uses the pub package manager to handle dependencies. The command line supports downloading of the appropriate package and some IDEs also provide the ability to load information. In the example shown, `dart:math` is bundled with the SDK, so it does not need additional dependencies to be added.

Where an external package is used, a pubspec.yaml file will need to be defined to indicate information about the package to be used. A pubspec.yaml file is metadata about the library being used. The file uses yaml format and enables dependencies to be listed in a consistent manner. For example the `google_fonts` package would use the following declaration in the pubspec.yaml definition.

```
dependencies:
  google_fonts: ^2.1.0
```

In the Dart source code, the import statement can then reference the package.

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

---

[1] In the above example, the keyword `this` has been omitted from the currentDay variable assignment. Dart best practice indicates the keyword `this` should be omitted unless required.

About the Author

Rich loves building things in the cloud and tinkering with different technologies. Lately this involves either Kubernetes or Serverless. Based in the UK, he enjoys attending (Ya remember that!) technical conferences and speaking to other people about new technologies. When he's not working, he likes spending time with his family, playing the guitar and riding his mountain bike. To improve his development skills he has also started writing smaller utility applications to simplify the more repetitive tasks (e.g. image manipulation, text manipulation, studying for certifications). Rich is also the author of *Hands-On Serverless Computing with Google Cloud*.