O'REILLY®

Flutter & Dart Cookbook

Developing Full-Stack Applications for the Cloud

Richard 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

O'REILLY®

Beijing • Boston • Farnham • Sebastopol • Tokyo

Flutter and Dart Cookbook

by Rich Rose

Copyright © 2022 Richard Rose. All rights reserved.

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
- 2022-08-11 Fourth Release
- 2022-09-16 Fifth Release

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

The O’Reilly logo is a registered trademark of O’Reilly Media, Inc. *Flutter and Dart Cookbook*, the cover image, and related trade dress are trademarks of O’Reilly Media, Inc.

The views expressed in this work are those of the author and do not represent the publisher’s views. While the publisher and the author have

used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-11945-4

Chapter 1. Starting with Dart

> **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 1st 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 editor at *jbleiel@oreilly.com*.

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.

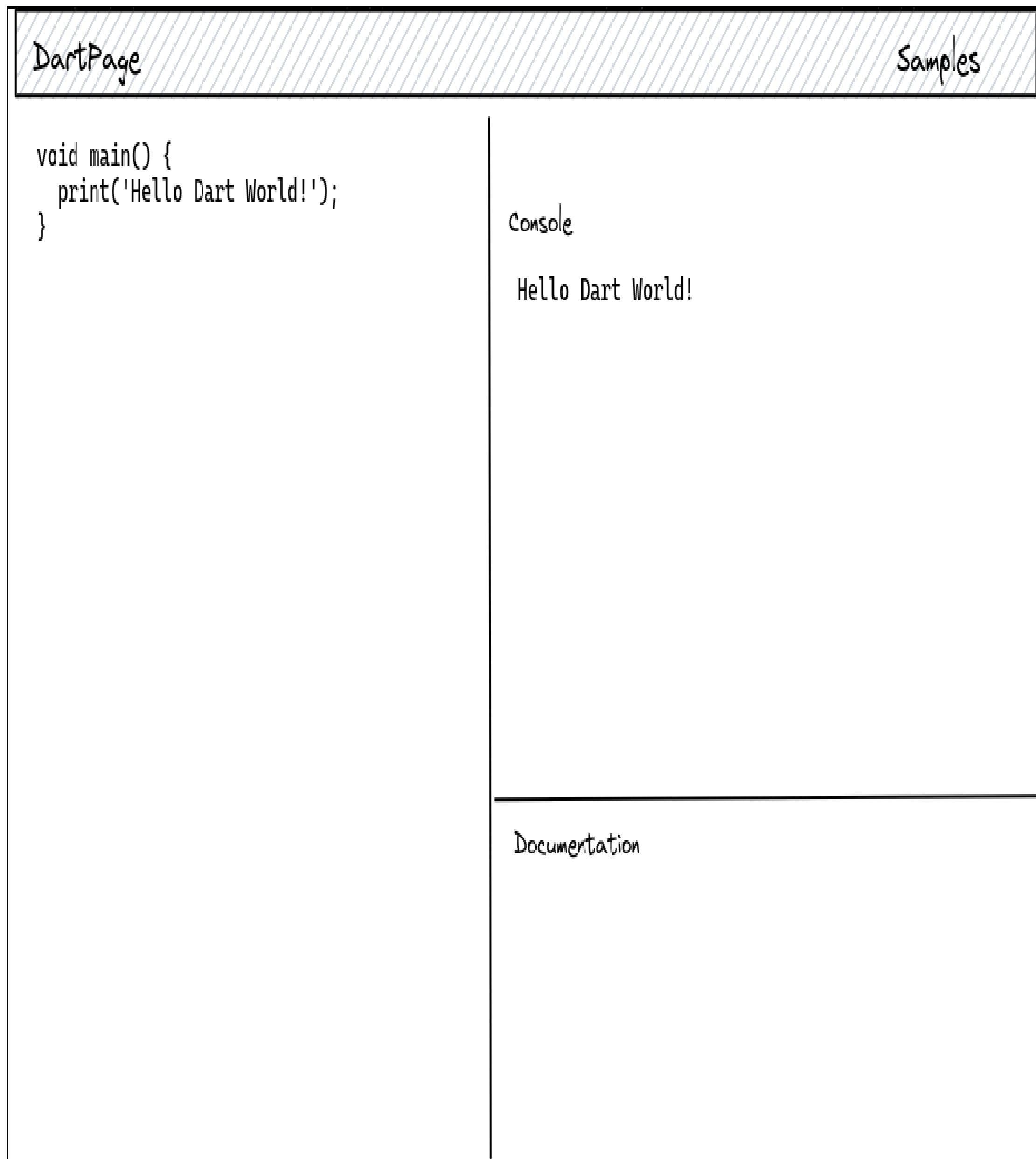

Figure 1-1. The DartPad interface

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

> **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 2nd 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 editor at *jbleiel@oreilly.com*.

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:

```dart
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

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

String interpolation with braces

```dart
  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.xhtml)

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 editor 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

> **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 4th 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 editor at *jbleiel@oreilly.com*.

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

> **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 5th 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 editor at *jbleiel@oreilly.com*.

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

> **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 6th 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 editor at *jbleiel@oreilly.com*.

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.

Chapter 7. Introducing the Flutter Framework

> **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 7th 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 editor at *jbleiel@oreilly.com*.

In this chapter, we begin our journey with the Flutter Framework, and focus on some of the fundamentals of Flutter.

When beginning with Flutter it is important to cover the fundamentals. For me the best place to start a Flutter application is a diagram of how your application will look and operate. The Flutter team has your back as they provide a wide range of templates to get you started coding your application. Once we have a starter code, it's time to understand the difference between stateful and stateless widgets, which will be a continual question as you build out your designs.

We take a brief moment to look at refactoring widgets. Adding complexity to your applications will mean this activity will be something you will iterate on multiple times. Refactoring your code is a skill that I highly recommend to avoid bugs and aid general performance. Thankfully Flutters ability to create ever more complex interfaces, means this investment in time really pays off.

7.1 Mocking an interface

Problem

You want a way to mock an interface to understand layout before creating a Flutter application.

Solution

Use a graphics package to design your application. Depending on your budget and use case, the following options may prove helpful.

Product	Link	Price	Description
Excalidraw	https://excalidraw.com/	Free	A general web based graphic design tool.
Figma	https://www.figma.com/	Free/Paid	A shared design and build solution for applications.
FlutterFlow	https://flutterflow.io/	Free/Subscription	Interactive UI templates and components that generate the Flutter code.

Discussion

Mocking an interface is an excellent way to get started with a visual framework like Flutter. There are many ways to design an interface ranging from free online tools to dedicated applications specifically created for Flutter.

When creating a mock of an application, I aim to capture the interface from the perspective of widgets to be used. Doing this makes it easier to build certain designs. If you are dealing with more complex designs, building an understanding of the application demands leads to a cleaner interface and design aesthetic.

Figure 7-1 is an example output using Excalidraw of the first Flutter application I created.

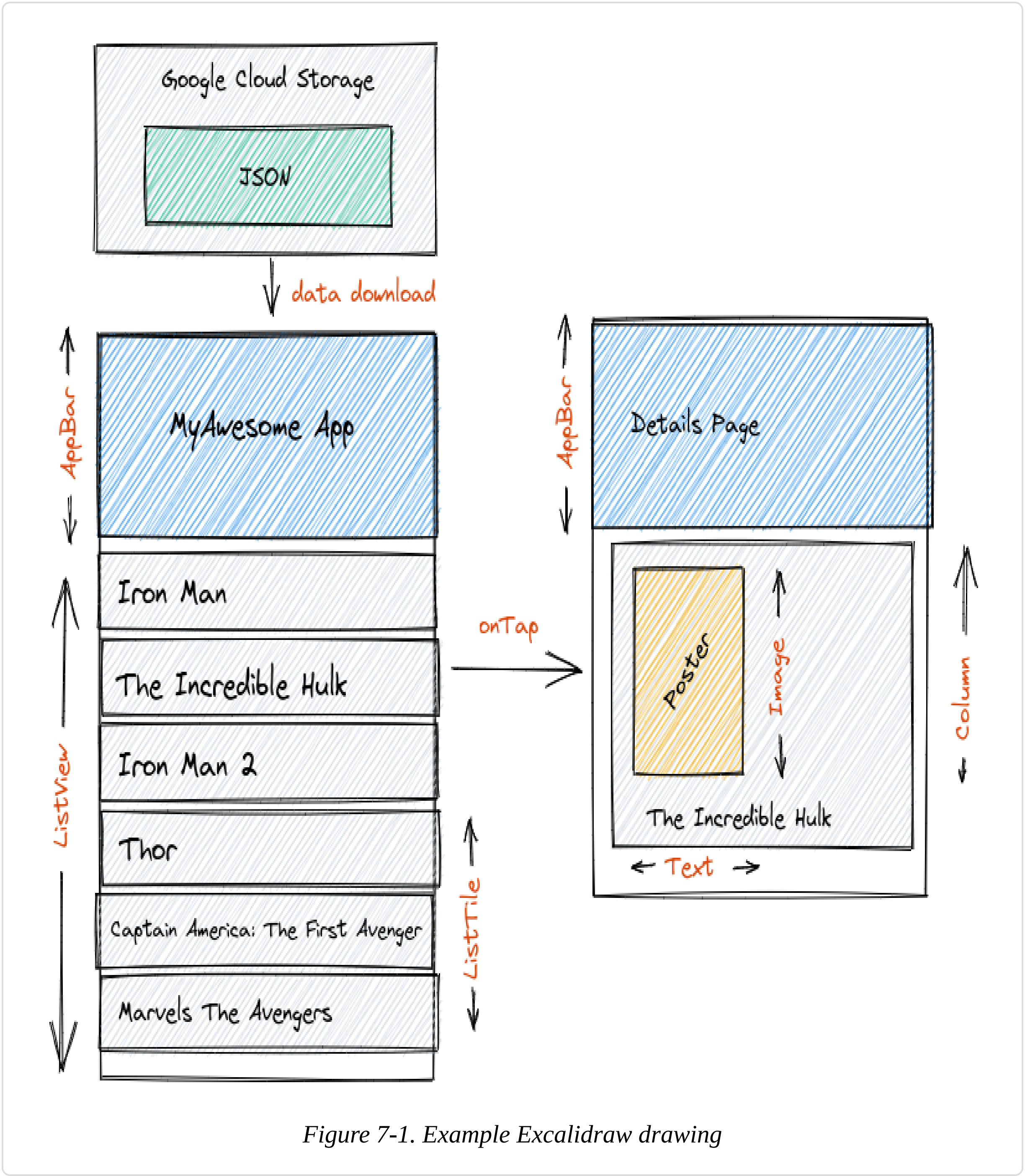

Figure 7-1. Example Excalidraw drawing

In the diagram, I include the functionality and screen transition required. Breaking down the interface is a good way to learn how the various widgets interact. Also learning the correct terminology for widgets, etc helps to find the appropriate solution. While the application is not very sophisticated, it did help me to learn the fundamentals of widget construction using Flutter.

From the diagram you should be aware that this type of interface is very common among Flutter applications. Learning to incorporate a ListView is essential as is handling gestures and navigation. Each of these patterns is covered in this chapter and can act as a reference for future implementations.

7.2 Creating a Flutter project

Problem

You want to create a new Flutter application based on a template.

Solution

Use a Flutter template to start your application. You don't have to start from scratch, because Flutter provides a range of application templates. There are a number of different templates available that provide a basic setup.

In more recent versions of the Flutter framework, templates have been improved to include the following templates:

Type	Description
app	This is the default for flutter create and is used to generate a Flutter application.
module	This option will enable you to create a module that can be integrated with other applications.
package	This option will enable a sharable Flutter project
plugin	This option provides an api base for use with Android and iOS
skeleton	This option provides a best practice application based on a Detail View

By appending the template command i.e. ***--template*** or ***-t***, you can indicate to Flutter that a template is to be applied at creation. Here's some examples of how the templates are used:

To create the default application type:

```
flutter create my_awesome_app
```

To create a module:

```
flutter create -t module my_awesome_module
```

To create a package:

```
flutter create -t package my_awesome_package
```

To create a plugin:

```
flutter create -t plugin my_awesome_plugin --platforms web --
platform android
```

> **NOTE**
>
> NOTE: When creating a plugin you must specify the platform to be supported. Each platform to be added requires the addition of the “--platform” prefix.

To create a package:

```
flutter create -t skeleton my_awesome_skeleton
```

Discussion

When you create a project based on a template, you must consider the device currently available on your machine. In addition to templates, you may also reference sample code from the API documentation website http://docs.flutter.dev/. To use the code from the site you need to reference the sample id located on the page for the widget to be used. In the example below, the code can be found on the webpage https://api.flutter.dev/flutter/widgets/GestureDetector-class.xhtml.

```
flutter create -s widgets.GestureDetector.1 my_awesome_sample
```

Samples are available to provide a quick way to access the multitude of content available online. As a developer you should aim to target the Web in addition to the desired host platform. Including the web makes sense as it includes a very effective method of enabling application testing within a browser. During the development phase, this approach can certainly improve developer velocity for both small and large enhancements.

7.3 Working with a Stateful Widget

Problem

You want to store a state (i.e. a value) associated with a Flutter widget.

Solution

Use the Flutter StatefulWidget to retain a value within an application. The declaration of a stateful widget indicates that a value is to be retained.

In the example below a widget is declared to hold the state.

```
import 'package:flutter/material.dart';
void main() {
  runApp(const MyApp());
}
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    const title = 'Stateless Widget demo';
    return MaterialApp(
      title: title,
      home: Scaffold(
        appBar: AppBar(
          title: const Text(title),
        ),
        body: const MyTextWidget(),
      ),
    );
  }
}
class MyTextWidget extends StatefulWidget {
  const MyTextWidget({Key? key}) : super(key: key);
  @override
  _MyTextWidget createState() => _MyTextWidget();
}
class _MyTextWidget extends State<MyTextWidget> {
  int count = 0;

  @override
  Widget build(BuildContext context){
    return GestureDetector(
      onTap: () {
        setState((){
          count++;
```

```
        });
      },
      child: Center(child: Text('Click Me: $count')),
    );
  }
}
```

Discussion

Storing state adds complexity to a Flutter application as the value needs to be tracked. A good pattern to observe when using a stateful widget is to reduce the number of widgets associated with state management.

State management in Flutter typically utilizes the pattern shown in Figure 7-2. A stateful widget requires the creation of a few methods that are used to retain information.

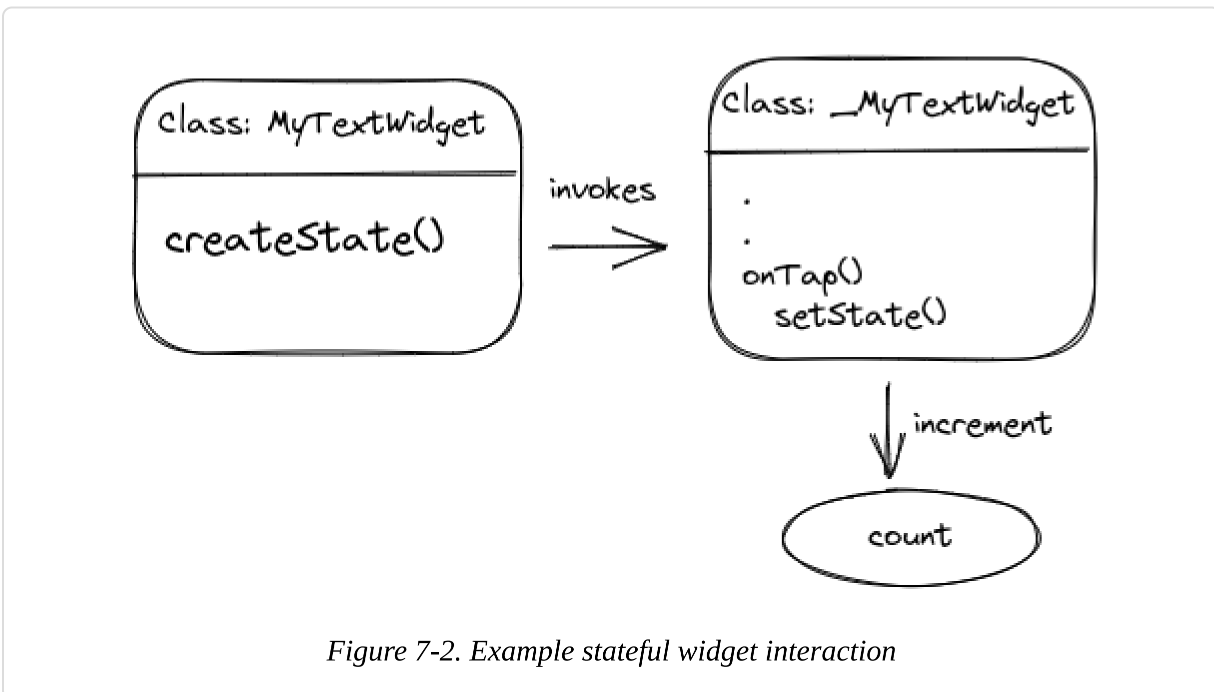

Figure 7-2. Example stateful widget interaction

The class MyTextWidget Stateful widget implements a createState method. The value returned from this method is assigned to a private variable i.e. _MyTextWidget. In Flutter private variables are prefixed with the underscore character. Observe how the private variable is constructed similar to the stateless widgets seen previously. The introduction of a new function setState is used to store a value based on an onTap event. In the

example, the count variable is incremented each time an onTap event is triggered.

The private class _MyTexWidget is then used to initiate state change. In the diagram we can see that the onTap() is used to increment the count variable. Now when a user of the application interacts and presses the button the variable will be incremented and the state change reflected in the application.

Working with stateful widgets is more of a challenge than working with stateless, but with some consideration of design, it can be just as effective. As a developer you should be in a position to design a minimal state application. Doing so will reduce the overall complexity and minimize the potential impact on performance associated with redraw to update on state changes.

Another consideration when using Stateful widgets is how to pass information. Reference recipe 11.5 for an overview of how to utilize keys to communicate parameters.

7.4 Working with a Stateless Widget

Problem

You want do not need to save state (i.e. not save a value) associated with on screen content

Solution

Use a stateless widget which will be just used to render on screen content. The following example shows how.

```
import 'package:flutter/material.dart';
void main() {
  runApp(const MyApp());
}
class MyApp e
xtends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);
```

```
  @override
  Widget build(BuildContext context) {
    const title = 'Stateless Widget demo';
    return MaterialApp(
      title: title,
      home: Scaffold(
        appBar: AppBar(
          title: const Text(title),
        ),
        body: const MyTextWidget(),
      ),
    );
  }
}
class MyTextWidget extends StatelessWidget {
  const MyTextWidget({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return const Center(
      child: Text('Hello'),
    );
  }
}
```

Discussion

In the example the StatelessWidget is used to render a Text widget. A stateless widget essentially means that value retention is not required. Often you will be in a situation where you may need to consider saving state. Consider how to best incorporate the demands of the widgets to be used. Flutter is super flexible on the interaction between stateless and stateful, don’t be under the impression it is one or the other.

In the example there is no value to store. Use this type of Widget where rendering a value on screen doesn’t require storing of information. Working without state is the easiest option as you have a lot less to consider in terms of how your application works. If you can, try and veer towards a stateless design to reduce the overall complexity of developing code where practical.

7.5 Refactoring Widgets

Problem

You want a way to improve the readability of code.

Solution

Use refactoring to improve the general reliability of your code. Refactoring allows you to simplify code.

The following code provides an example of code requiring refactoring based on the Build function.

```
        body: Container(
        width:  200,
        height: 180,
        color: Colors.black,
        child: Column(
          children: [
            Image.network(
                'https://images.unsplash.com/photo-1499028344343-
cd173ffc68a9'),
            const Text(
              'itemTitle',
              style: TextStyle(fontSize: 20, color: Colors.white),
            ),
            const Text(
              'itemSubTitle',
              style: TextStyle(fontSize: 16, color: Colors.grey),
            ),
          ],
        ),
      ),
```

The following example provides an example of code that has been refactored.

```
import 'package:flutter/material.dart';
void main() {
  runApp(const MyApp());
}
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    const title = 'Image Widget';
    return MaterialApp(
      title: title,
      home: Scaffold(
```

```
        appBar: AppBar(
          title: const Text(title),
        ),
        body: const MyContainerWidget(),
      ),
    );
  }
}
class ImageItem {
  final String title;
  final String subtitle;
  final String url;

  const ImageItem({
    required this.title,
    required this.subtitle,
    required this.url,
  });
}
class MyContainerWidget extends StatelessWidget {
  const MyContainerWidget();
  final ImageItem imageItem = const ImageItem(
          title: 'Hello',
          subtitle: 'subtitle',
          url: 'https://images.unsplash.com/photo-1499028344343-
cd173ffc68a9'
  );

  @override
  Widget build(BuildContext context) {
    return Container(
      width: 200,
      height: 180,
      color: Colors.black,
      child: Column(
        children: [
          Image.network(imageItem.url),
          Text(
            imageItem.title,
            style: const TextStyle(fontSize: 20, color:
Colors.white),
          ),
          Text(
            imageItem.subtitle,
            style: const TextStyle(fontSize: 16, color:
Colors.grey),
          ),
        ],
      ),
    );
```

```
  }
}
```

Discussion

Readability is a big subject and beyond the scope of this book. Put simply we are referring to whether the meaning of the code can be easily discerned.

There are two things to note in the code requiring refactoring. First the linked image data is embedded in the main codebase. Embedding data in code makes it hard to understand and subject to errors, when the code is changed. The code also embeds the Container widget within the body, which again makes this hard to change, but also makes it difficult to enhance.

A better practice is to isolate the code such that it can be enhanced independently without affecting the existing code. Moving the widget element to its own class introduces a significant advantage in that the class is both independent and can now be called by any method.

To manage the data requirement, we create a new data class to hold the required variables for the MyContainerWidget. The data class includes a constructor (e.g. required this.title, etc) that expects you to pass the values to be used. Abstracted the data requirement from the widget is a good practice to follow and gives your applications a lot of flexibility in terms of code reuse.

Observe the class definition of “MyContainerWidget” which defines the new widget as a StatelessWidget. The class constructor indicates that it expects our ImageItem data class to be provided. The ImageItem variable is declared as final, which means the value is to be determined at runtime.

Finally the build function is used to provide the widget functionality. In this function we return our new Container widget to the calling function. This type of code abstraction enables the “MyContainerWidget” to be functionally independent of “MyApp”. In taking this approach we have enhanced the overall readability of the application.

Overall the changes made ensure that when creating and using the MyContainerWidget the process is both consistent and repeatable. As a developer making reference to MyContainerWidget, you now only need to be aware of the data requirement and the class invocation. Ultimately based on the changes made, you now have two classes representing the data and implementation.

Now we know the basics of refactoring code, we incorporate these simple ideas in our thinking going forward.

7.6 Removing the Flutter Debug banner

Problem

You want a way to remove the debug banner from your Flutter application.

Solution

Use the debugShowCheckModeBanner to remove the debug banner applied to Flutter applications.

In the example below the debug property is turned off.

```
import 'package:flutter/material.dart';
void main() {
  runApp(MyApp());
}
class MyApp extends StatelessWidget {
  final appTitle = 'Cert In';
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: appTitle,
      theme: ThemeData(
        primarySwatch: Colors.blue,
        visualDensity: VisualDensity.adaptivePlatformDensity,
        textTheme: GoogleFonts.montserratTextTheme(),
      ),
      debugShowMaterialGrid: false,
      debugShowCheckedModeBanner: false,
      home: Professional(title: appTitle),
    );
```

```
  }
}
```

Discussion

The debugShowCheckedModeBanner accepts a boolean value to indicate whether the notification should be shown. In the example, the “Debug” message is turned off by setting the property to false.

Flutter has a default value of true set for the debugShowCheckedModeBanner. Developers are required to explicitly set this value as false to remove the temporary banner from applications. The table below outlines the various settings for the application states of Debug and Release.

Mode	Property	Discussion
Debug	`debugShowCheckedModeBanner`	The banner can be controlled via the boolean value. Setting the property to true will show the banner, this is the default for new applications. Amending the property to false, will remove the banner from your application.
Release	`debugShowCheckedModeBanner`	The banner is not displayed when in Release mode, irrespective of the property setting.

The debugShowMaterialGrid setting provides a grid overlay for your application. To use this setting your application needs to be in debug mode.

Chapter 8. Working with Widgets

> **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 8th 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 editor at *jbleiel@oreilly.com*.

In this chapter we continue our journey with the Flutter Framework. We now progress to a high level overview of the most common widgets.

Widgets are an essential concept in Flutter and provide the basis of most applications. Learning how to integrate the numerous widget types available will significantly enhance your development skills. Understanding how to use Scaffold, Row, and Container, together with other widgets, will allow you to develop a wide range of applications.

As part of this introduction you will hopefully also note how to combine widgets to deliver feature rich and beautiful interfaces. Knowing how to build beyond the basics building blocks of a Flutter application will steadily increase your confidence and provide the basis for more complex applications.

8.1 Using the Scaffold class

Problem

You want to work with the material design layout in your application.

Solution

Use the Scaffold to present a material interface within an application.

In the example we define the typical elements of a Scaffold layout.

```
import 'package:flutter/material.dart';
void main() => runApp(const MyApp());
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);
  static const String _title = 'Example';
  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      title: _title,
      home: MyStatelessWidget(),
    );
  }
}
class MyStatelessWidget extends StatelessWidget {
  const MyStatelessWidget({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Scaffold Example')),
      backgroundColor: Colors.blueGrey,
      bottomNavigationBar: const BottomAppBar(
        color: Colors.blueAccent,
        shape: CircularNotchedRectangle(),
        child: SizedBox(
          height: 300,
          child: Center(child: Text("bottomNavigationBar")),
        ),
      ),
      body: _buildCardWidget(),
    );
  }
  Widget _buildCardWidget() {
    return const SizedBox(
      height: 200,
      child: Card(
        child: Center(
          child: Text('Top Level Card'),
        ),
      ),
    );
```

```
  }
}
```

Discussion

When you use Scaffold it expands to fill the available space and it will dynamically adjust based on screen alterations. This behavior is desirable as you will mainly want your application to fill the screen. If an onscreen keyboard is present, the default Scaffold will adjust dynamically without additional logic being added to your application.

Scaffold provides the ability to enhance your application with AppBar (see Recipe 8.2) and Drawer widgets (Recipe TBC). If you wish to use Scaffold with a FloatingActionButton, use a Stateful widget to retain the associated button state. The Stateful widget (recipe TBC) can be followed to add this to your code.

In general, avoid nesting the Scaffold class as it is designed to be a top level container for a MaterialApp.

8.2 Using an AppBar

Problem

You want to show a toolbar header section at the top of your application.

Solution

Use an AppBar widget to control the header section of your application.

Here’s an example of how to work with an AppBar widget in Flutter:

```
import 'package:flutter/material.dart';
void main() {
  runApp(const MyApp());
}
class MyApp extends StatelessWidget {
  final String title = 'Container Widget';
  const MyApp({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
```

```
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      title: title,
      home: Scaffold(
        appBar: MyAppBar(title:title),
        body: const MyCenterWidget(),
      ),
    );
  }
}
class MyAppBar extends StatelessWidget implements
PreferredSizeWidget {
  final String title;
  final double sizeAppBar = 200.0;

  const MyAppBar({Key? key, required this.title}) : super(key:
key);

  @override
  Size get preferredSize => Size.fromHeight(sizeAppBar);

  @override build(BuildContext context) {

    return AppBar(
      title: Text(title),
      backgroundColor: Colors.black,
      elevation: 0.0,
      leading: IconButton(onPressed: (){}, icon: const
Icon(Icons.menu)),
      actions: [
        IconButton(
        onPressed: (){}, icon: const Icon(Icons.settings))
      ],
    );
  }
}
class MyCenterWidget extends StatelessWidget {
  const MyCenterWidget({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return const Center(
      child: Text('Hello Flutter!'),
    );
  }
}
```

Discussion

In the above example, the AppBar (https://api.flutter.dev/flutter/material/AppBar/AppBar.xhtml) Widget has been moved to its own class. The AppBar requires an awareness of the dimensions for the header to be used. To address this, we call a static AppBar and use this to provide the appropriate dimension for our widget. The preferredSize is dynamic and will resize based on the content to be displayed.

Using an AppBar provides access to a number of properties. Typically developers will use the main properties such as Title, backgroundColor and elevation. Set the backgroundColor property of the AppBar directly to use a range of available colors. Add an elevation property to display a flat (zero) or raised (>zero) graphical interface.

In addition there are many more options available. If you need a menu option, use the leading property to add Icons to the left hand side of the interface. Actions buttons can also be added to the interface to provide further interaction. The example code demonstrates how to incorporate both types of properties.

8.3 Using an Expanded widget

Problem

You want to automatically utilize any available onscreen space.

Solution

Use the Expanded widget to coordinate the available space visible to the user (i.e. the viewport).

In the example we use an expanded widget to define three onscreen elements that will coordinate to use the available onscreen dimensions.

```
import 'package:flutter/material.dart';
void main() {
  runApp(const MyApp());
}
```

```
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    const title = 'Expanded Widget';
    return MaterialApp(
      title: title,
      home: Scaffold(
        appBar: AppBar(
          title: const Text(title),
        ),
        body: const MyExpandedWidget(),
      ),
    );
  }
}
class MyExpandedWidget extends StatelessWidget {
  const MyExpandedWidget();
  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        Expanded(
          child: Container(
            color: Colors.red,
          ),
        ),
        Expanded(
          child: Container(
              ),
        ),
        RichText(
          text: const TextSpan(
            text: 'Luxembourg',
            style: TextStyle(
              fontWeight: FontWeight.bold,
              fontSize: 24,
              color: Colors.grey,
            ),
          ),
        ),
        Expanded(
          child: Container(
              ),
        ),
        Expanded(
          child: Container(
            color: Colors.blue,
          ),
        ),
```

```
      ],
    );
  }
}
```

Discussion

Despite the simplicity of the Expanded widget, it can be used in a variety of use cases. In the example, rather than perform a query on the dimensions on the screen, the Expanded widget is used to automatically fill the available space. Expanded (as the name suggests) will expand to the screen dimensions automatically. You do not have to specify the proportions, this will be calculated dynamically.

An Expanded widget's behavior is beneficial when working with Lists. Consider the above ListView widget, which dynamically populates a vertical list with the ListTile widgets at runtime. When using a ListView by itself, it will try to consume the available dimensions in the viewport.

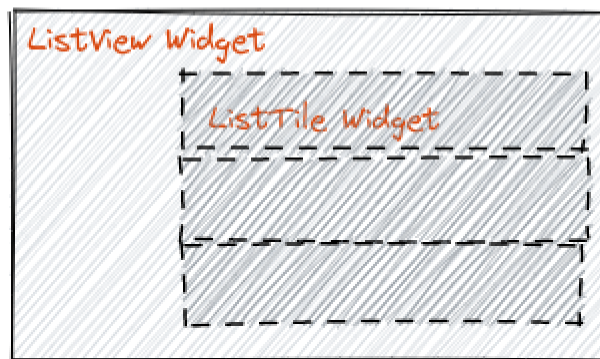

Figure 8-1. A ListView widget

Here the Expanded widget is used to tell the ListView it should consume the remaining viewport (i.e. the user's visible area on screen).

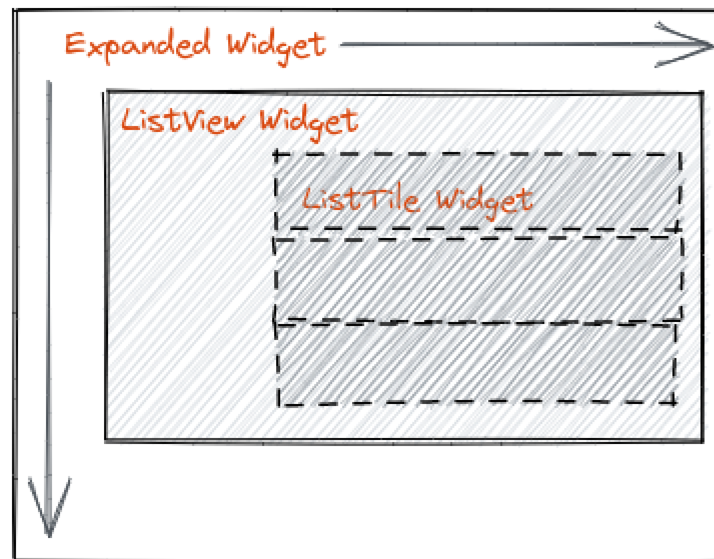

Figure 8-2. A ListView widget enclosed within an Expanded widget

If you choose to incorporate other on screen widgets, your application will want to know how to ratio the viewport allocation between each item. In this instance the Expanded widget can be used to automatically provide the correct dimensions available within the viewport.

8.4 Building with a Container

Problem

You want a way to isolate settings for a child widget or series of widgets.

Solution

Use a Container widget to provide configuration (e.g. padding, border, colors) for other child widgets. The container widget provides a defined structure in which to place other widgets.

In the example below a container widget is used to define an area that can be used to define additional widgets.

```
import 'package:flutter/material.dart';
void main() {
  runApp(const MyApp());
}
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    const title = 'Container Widget';
    return MaterialApp(
      title: title,
      home: Scaffold(
        appBar: AppBar(
          title: const Text(title),
        ),
        body: const MyCenterContainerWidget(),
      ),
    );
  }
}
class MyCenterContainerWidget extends StatelessWidget {
  const MyCenterContainerWidget({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return Center(
      child: Container(
        alignment: Alignment.center,
        height: 200,
        width: 200,
        color: Colors.red[300],
        transform: Matrix4.rotationZ(0.5),
        child: Container(
          color: Colors.blue[300],
          transform: Matrix4.rotationZ(0.5),
          child: Container(
            color: Colors.yellow[300],
            transform: Matrix4.rotationZ(0.5),
            child: Container(
              color: Colors.grey[300],
              transform: Matrix4.rotationZ(0.5),
              child: Container(
                color: Colors.orange[300],
                transform: Matrix4.rotationZ(0.5),
                child: Container(
                  color: Colors.black,
                  transform: Matrix4.rotationZ(0.5),
                  child: Container(
                    color: Colors.green[300],
                    transform: Matrix4.rotationZ(0.5),
                    child: Container(
```

```
                      color: Colors.indigo[300],
                      transform: Matrix4.rotationZ(0.5),
                      child: Container(
                        color: Colors.purple[300],
                        transform: Matrix4.rotationZ(0.5),
                        child: Container(
                          color: Colors.lime[300],
                          transform: Matrix4.rotationZ(0.5),
                          child: Container(
                            color: Colors.brown[300],
                            transform: Matrix4.rotationZ(0.5),
                            child: Container(
                              color: Colors.teal[300],
                              transform: Matrix4.rotationZ(0.5),
                            ),
                          ),
                        ),
                      ),
                    ),
                  ),
                ),
              ),
            ),
          ),
        ),
      ),
    );
  }
}
```

Discussion

Containers with no children will attempt to fill the space available to them. Containers support width and height values, but these can be omitted. Typically a container will be sized based on the dimensions of its children. When specifying the size and height of the child widget, it is constrained to the dimensions of the parent. To override this behavior use the width and height properties on the parent Container.

Note child element is constrained by the parent Container widget. You can still specify the size and height, but it is relative to the parent. If you want to provide whitespace in an application look to use SizedBox (Recipe 8.6).

8.5 Using a Center widget

Problem

You want to ensure that content is centered on screen.

Solution

Use the Center widget to align a child element on screen.

In the example the Center widget is used to center on the horizontal and vertical axis.

```
import 'package:flutter/material.dart';
void main() => runApp(const MyApp());
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);
  static const String _title = 'Example';
  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      title: _title,
      home: MyStatelessWidget(),
    );
  }
}
class MyStatelessWidget extends StatelessWidget {
  const MyStatelessWidget({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Center Example')),
      body: _buildCenterWidget(),
    );
  }
  Widget _buildCenterWidget() {
    return Column(
      children: [
        const Center(
          child: Text("Center Text"),
        ),
        Row(children: [
          Container(
              color: Colors.green,
              child: const Center(child: Text("Container
Center"))),
          Container(
              color: Colors.teal,
              child: const Center(child: Text("Container
Center"))),
          Expanded(
```

```
            child: Container(
                color: Colors.yellow,
                child: const Center(child: Text("Container
Center"))),
          ),
        ]),
        Container(
            color: Colors.blue,
            child: const Center(child: Text("Container Center"))),
        Container(
            color: Colors.red,
            child: const Center(child: Text("Container Center"))),
        Expanded(
          child: Container(
              color: Colors.yellow,
              child: const Center(child: Text("Container
Center"))),
        ),
      ],
    );
  }
}
```

Discussion

The Center widget provides a great way to center on screen objects. While it is relatively straightforward to use in an application, do not underestimate the power it provides. It can be used in conjunction with a wide range of widgets and will save you a fair bit of time in the development process.

8.6 Using a SizedBox

Problem

You want to add whitespace to a user interface.

Solution

Use the SizedBox widget to apply a defined space to the onscreen interface.

In the example we apply the SizedBox widget to the user interface.

```
import 'package:flutter/material.dart';
void main() => runApp(const MyApp());
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);
  static const String _title = 'Example';
  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      title: _title,
      home: MyStatelessWidget(),
    );
  }
}
class MyStatelessWidget extends StatelessWidget {
  const MyStatelessWidget({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('SizedBox Example')),
      body: _buildSizedBoxWidget(),
    );
  }
  Widget _buildSizedBoxWidget() {
    return Column(
      children: [
        Container(
          width: 200,
          height: 200,
          color: Colors.red,
        ),
        const SizedBox(
          width: 100,
          height: 100,
        ),
        const SizedBox(
          width: 400,
          height: 300,
          child: Card(
            child: Center(
              child: Text('Hello World'),
            )
          ),
        ),
      ],
    );
  }
}
```

Discussion

The SizedBox widget provides a simple method to specify a box size to be used in your application. By applying a size, you can apply a constraint on child widgets.

If a size is not provided, the widget can be sized appropriately based on the dimension of the child widget presented. The typical use case for SizedBox is to provide whitespace in an application, but it can also be used in a similar manner to a Container Widget (Recipe 8.4).

8.7 Using a Column

Problem

You need a flexible widget to present as a vertical layout on screen.

Solution

Use a Column widget to allow information to be displayed as a vertical array.

Here’s an example of how to use the Column widget in Flutter:

```
import 'package:flutter/material.dart';
void main() => runApp(const MyApp());
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);
  static const String _title = 'Example';
  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      title: _title,
      home: MyStatelessWidget(),
    );
  }
}
class MyStatelessWidget extends StatelessWidget {
  const MyStatelessWidget({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Column Example')),
      body: _buildColumnWidget(),
    );
```

```
  }
  Widget _buildColumnWidget() {
    return Column(
      mainAxisAlignment: MainAxisAlignment.center,
      crossAxisAlignment: CrossAxisAlignment.start,
      children: <Widget> [
        Container(
          height: 200,
          width: 50,
          color: Colors.red,
          child: const Center(child: Text("50"),),
        ),
        Container(
          height: 200,
          width: 100,
          color: Colors.green,
          child: const Center(child: Text("100"),),
        ),
        Container(
          height: 200,
          width: 200,
          color: Colors.orange,
          child: const Center(child: Text("200"),),
        ),
        Container(
          height: 200,
          width: 500,
          color: Colors.blue,
          child: const Center(child: Text("1000"),),
        ),
      ]
    );
  }
}
```

Discussion

In the example code we have three columns created on the vertical (Y) axis. The area defined for the column is non-scrolling. Allocation of height and width properties can be used to constrain the area available to the Column widget.

If a constraint is applied, then the widget will attempt to handle the changes presented. In some situations, the changes cannot be correctly interpreted, which leads to an overflow error. Overflows are a very common error and are displayed as a series of yellow and black stripes on screen. The error is

accompanied by an overflow warning indicating why the error occurred. To understand the options applicable to constraints, consult the Flutter documentation at https://docs.flutter.dev/development/ui/layout/constraints.

When using a Column widget, you can align content on the X and Y axis. Alignment on the horizontal (X) axis is enabled with crossAxisAlignment. Use this to set start, center or end alignment.

To incorporate alignment, use the mainAxisAlignment property for the Y axis. Alignment ensures that the free space is evenly distributed between children, including before, and after the first/last child in the array. The mainAxisAlignment supports start, center and end properties to shift the Y axis anchor.

Column and Row widgets share a lot of commonality. Reference Recipe 8.8 for a direct comparative with a Row widget.

8.8 Using a Row

Problem

You need a flexible widget to present as a horizontal layout on screen.

Solution

Use a Row widget to allow its children widgets to be displayed as a horizontal array.

Here’s an example of how to use the Row widget in Flutter:

```
import 'package:flutter/material.dart';
void main() => runApp(const MyApp());
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);
  static const String _title = 'Example';
  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      title: _title,
      home: MyStatelessWidget(),
    );
```

```
  }
}
class MyStatelessWidget extends StatelessWidget {
  const MyStatelessWidget({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Row Example')),
      body: _buildRowWidget(),
    );
  }
  Widget _buildRowWidget() {
    return Row(
        mainAxisAlignment: MainAxisAlignment.start,
        crossAxisAlignment: CrossAxisAlignment.end,
        children: [
          Container(
              width: 5,
              color: Colors.transparent,
          ),
          Expanded(
            child: Container(
              height: 50,
              width: 200,
              color: Colors.red,
              child: const Center(
                child: Text("50"),
              ),
            ),
          ),
          Expanded(
            child: Container(
              height: 100,
              width: 200,
              color: Colors.green,
              child: const Center(
                child: Text("100"),
              ),
            ),
          ),
          Expanded(
            child: Container(
              height: 200,
              width: 200,
              color: Colors.orange,
              child: const Center(
                child: Text("200"),
              ),
            ),
          ),
```

```
          Container(
              width: 5,
              color: Colors.transparent,
          ),
        ]);
  }
}
```

Discussion

In the example code data rows are created on the horizontal (X) axis. Row Widgets are static, meaning they do not enable scrolling.

If data displayed overflows beyond the screen dimensions, it is displayed as a series of yellow and black stripes on screen. The example applies a constraint to the widget displayed, so it will dynamically adjust to the screen size. To understand the options applicable to constraints, consult the Flutter documentation at https://docs.flutter.dev/development/ui/layout/constraints.

The alignment of the vertical (Y) axis uses the mainAxisAlignment property to use the start of the screen as an anchor. The mainAxisAlignment supports start, center and end properties to shift the Y axis anchor.

Alignment on the horizontal (X) axis is performed by crossAxisAlignment. Again this method supports start, center and end properties to shift the X axis anchor.

Column and Row widgets share a lot of commonality. Reference Recipe 8.7 for a direct comparative with a Column widget.

Chapter 9. Developing User Interfaces

> **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 9th 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 editor at *jbleiel@oreilly.com*.

In this chapter, we move on to the topic of building user interfaces. The discussion focuses on the key technical elements of designing a beautiful interface. You will learn how to utilize Fonts to enhance the text interface, define the onscreen layout for better placement and address identification of the host platform.

Leverage the features of Flutter to fundamentally improve your applications. Understand how to address platform specific areas of functionality through the Dart SDK. Construct code that works with Flutter to present information in the most performant manner. The recipes shown in this chapter will be key to building extensible applications to delight your users.

9.1 Incorporating Rich Text

Problem

You want to have more control over the text displayed on screen.

Solution

In the example the Rich Text widget is used to customize the text rendered on screen.

```
import 'package:flutter/material.dart';
void main() {
  runApp(const MyApp());
}
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    const title = 'Rich Text';
    return MaterialApp(
      title: title,
      home: Scaffold(
        appBar: AppBar(
          title: const Text(title),
        ),
        body: const MyRichText(),
      ),
    );
  }
}
double screenHeight = 0.0;
class MyRichText extends StatelessWidget {
  const MyRichText();
  @override
  Widget build(BuildContext context) {
    screenHeight = MediaQuery.of(context).size.height/3;

    return SingleChildScrollView(
      child: Column(
        children: [
          Container(
            height: screenHeight,
            color: Colors.red,
          ),
          Row(
            mainAxisAlignment: MainAxisAlignment.center,
            children: [
            SizedBox(height:  screenHeight),
            RichText(
              text: const TextSpan(
                children: [
                  TextSpan(
                    text: 'Hello',
                    style: TextStyle(fontWeight: FontWeight.bold,
fontSize: 24),
```

```
                  ),
                  TextSpan(
                    text: 'Luxembourg',
                    style: TextStyle(
                        fontWeight: FontWeight.bold,
                        fontSize: 32,
                        color: Colors.grey),
                  ),
                ],
              ),
            ),
          ]),
          Container(
            height:  screenHeight,
            color: Colors.blue,
          ),
        ],
      ),
    );
  }
}
```

Discussion

In the example the flag of Luxembourg is shown on screen. Use Rich Text when you need more control over the text to be displayed in an application.

The RichText widget provides greater control over the placement and styling associated with application text.

9.2 Incorporating the Google fonts package

Problem

You want to use a package to use external fonts in a Flutter application.

Solution

Flutter allows you to incorporate external fonts as part of your application. The following example demonstrates how to use Google Fonts[1].

pubspec.yaml

```
.
.
.
dependencies:
  flutter:
    sdk: flutter
  cupertino_icon: ^1.0.2
  google_fonts: 2.2.0
```

Main.dart

```
import 'package:flutter/material.dart';
import 'package:google_fonts/google_fonts.dart';
void main() => runApp(MyApp());
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Cookbook Demo',
      debugShowCheckedModeBanner: false,
      theme: ThemeData(
        primarySwatch: Colors.blue,
        textTheme: TextTheme(
          bodyText1: GoogleFonts.aBeeZee(fontSize: 30, color:
Colors.deepOrange),
          bodyText2: GoogleFonts.aBeeZee(fontSize: 30, color:
Colors.white60))
      ),
      home: const MyHomePage(title: 'Flutter Cookbook'),
    );
  }
}
class MyHomePage extends StatelessWidget {
  final String title;
  const MyHomePage({
    Key? key,
    required this.title,
  }) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: Colors.black,
      body: Column(children: [
        const Text('Yo MTV Raps'),
        Text('Yo MTV Raps', style: GoogleFonts.coiny(fontSize: 30,
color: Colors.blueGrey),),
        Text('Yo MTV Raps', style: GoogleFonts.actor(fontSize: 30,
color: Colors.indigo),),
      ]),
    );
```

```
  }
}
```

Discussion

If you are getting a package from the pub.dev, the instructions typically provide most of the information required. There are some general assumptions regarding placement and updates made, so it is worthwhile becoming familiar with how to integrate an external package with your application.

One of the places folks become confused is the addition to the pubspec dependencies section. The section to update is based on what entry you intend to make. For the addition of fonts, the addition already has an entry for Cupertino Fonts, so just add the Google Fonts entry below this setting.

In the application two approaches are used to set the Google Font. First the textTheme is set as part of the general application theme. Use this approach if you want to set a default for your application. A common question is why bodyText2 takes precedence. The default text style for Material is bodyText2.

The second approach applies the Google Font directly to the text widget. Here we can individually set the font as required.

9.3 Identifying the host platform

Problem

You want to verify which platform the application is being run on.

Solution

In some instances you may wish to know the specific host platform the application is running on. This can be useful, if you need to observe the user criteria to be applied within an application such as Android or iOS.

In the example below, the settings are assigned to variables that can be checked to identify which host platform the application is running on.

```
import 'package:flutter/material.dart';
import 'dart:io' show Platform;
import 'package:flutter/foundation.dart' show kIsWeb;
void main() {
  runApp(const MyApp());
}
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    const title = 'Platform demo';
    return MaterialApp(
      title: title,
      home: Scaffold(
        appBar: AppBar(
          title: const Text(title),
        ),
        body: const MyPlatformWidget(),
      ),
    );
  }
}
class MyPlatformWidget extends StatelessWidget {
  const MyPlatformWidget({Key? key}) : super(key: key);
  bool get isMobileDevice => !kIsWeb && (Platform.isIOS ||
Platform.isAndroid);
  bool get isDesktopDevice =>
      !kIsWeb && (Platform.isMacOS || Platform.isWindows ||
Platform.isLinux);
  bool get isMobileDeviceOrWeb => kIsWeb || isMobileDevice;
  bool get isDesktopDeviceOrWeb => kIsWeb || isDesktopDevice;
  bool get isAndroid => !kIsWeb && Platform.isAndroid;
  bool get isFuchsia => !kIsWeb && Platform.isFuchsia;
  bool get isIOS => !kIsWeb && Platform.isIOS;
  bool get isLinux => !kIsWeb && Platform.isLinux;
  bool get isMacOS => !kIsWeb && Platform.isMacOS;
  bool get isWindows => !kIsWeb && Platform.isWindows;
  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        const Text(
          'Web: $kIsWeb',
          style: TextStyle(fontSize: 20, color: Colors.grey),
        ),
        Text(
          'Android: $isAndroid',
```

```
            style: const TextStyle(fontSize: 20, color: Colors.grey),
          ),
          Text(
            'Fuchsia: $isFuchsia',
            style: const TextStyle(fontSize: 20, color: Colors.grey),
          ),
          Text(
            'IOS: $isIOS',
            style: const TextStyle(fontSize: 20, color: Colors.grey),
          ),
          Text(
            'Linux: $isLinux',
            style: const TextStyle(fontSize: 20, color: Colors.grey),
          ),
          Text(
            'MacOS: $isMacOS',
            style: const TextStyle(fontSize: 20, color: Colors.grey),
          ),
          Text(
            'Windows: $isWindows',
            style: const TextStyle(fontSize: 20, color: Colors.grey),
          ),
          Text(
            'isMobileDevice: $isMobileDevice',
            style: const TextStyle(fontSize: 20, color: Colors.grey),
          ),
          Text(
            'isDesktopDevice: $isDesktopDevice',
            style: const TextStyle(fontSize: 20, color: Colors.grey),
          ),
          Text(
            'isMobileDeviceOrWeb: $isMobileDeviceOrWeb',
            style: const TextStyle(fontSize: 20, color: Colors.grey),
          ),
          Text(
            'isDesktopDeviceOrWeb: $isDesktopDeviceOrWeb',
            style: const TextStyle(fontSize: 20, color: Colors.grey),
          ),
        ],
      );
  }
}
```

Discussion

Typically when testing it is useful to understand which host platform the application is running on. Flutter enables you to detect the host platform using pre-defined Platform constants. In the documentation the information

relating to the host platform is located under device segmentation and referenced in the Platform API.

The Platform API supports the main platforms (i.e. Android, Fuchsia, IOS, Linux, MacOS and Windows). As part of the example each test for a specific configuration can be used to identify the host platform. In addition there is a separate setting (i.e. kIsWeb) for Web based applications.

9.4 Using a Placeholder widget

Problem

You want to build a user interface when not all graphical assets are available.

Solution

Use the Placeholder widget to represent interface resources that have yet to be added to an application

In the example we revisit our flag example from Recipe 9.1.

```
import 'package:flutter/material.dart';
void main() => runApp(const MyApp());
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);
  static const String _title = 'Example';
  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      title: _title,
      home: MyStatelessWidget(),
    );
  }
}
class MyStatelessWidget extends StatelessWidget {
  const MyStatelessWidget({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Placeholder Example')),
      body: _buildPlaceholderWidget(),
    );
```

```
  }
  Widget _buildPlaceholderWidget() {
    return Column(
      children: const <Widget> [
        Placeholder(fallbackHeight: 400, strokeWidth: 10, color: 
Colors.red),
        Expanded(
          child: Text("Expanded Text")
        ),
        Placeholder(fallbackHeight: 200, strokeWidth: 5, color: 
Colors.green),
        Expanded(
          child: Text("Expanded Text")
        ),
        Placeholder(fallbackHeight: 100, strokeWidth: 1, color: 
Colors.blue),
      ]
    );
  }
}
```

Discussion

The example illustrates how a placeholder can be used to fill space that would otherwise be used. An expanded widget would utilize the space specified automatically unless we add a placeholder. If you need to allocate visual space without necessarily needing to add the supporting resource, a Placeholder widget can be super helpful.

The Placeholder supports additional properties such as widget height (fallbackHeight) and width (fallbackWidth). In addition the widget also supports color (color) and line width (strokeWidth) to provide additional flexibility in the design stage of building an application.

9.5 Using a Layout Builder

Problem

You want your layout to dynamically resize using an adaptive layout.

Solution

Use a LayoutBuilder widget to handle the screen adaptive layout requirements automatically.

In this example we revisit our flag example from Rich Text.

```
import 'package:flutter/material.dart';
void main() => runApp(const MyApp());
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);
  static const String _title = 'LayoutBuilder';
  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      title: _title,
      home: MyStatelessWidget(),
    );
  }
}
class MyStatelessWidget extends StatelessWidget {
  const MyStatelessWidget({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('LayoutBuilder Example')),
      body: LayoutBuilder(
        builder: (BuildContext context, BoxConstraints constraints)
{
          // Restrict based on Width
          if (constraints.maxWidth > 800) {
            return _buildTripleContainers();
          } else
          if (constraints.maxWidth > 600 &&
constraints.maxWidth<=800) {
            return _buildDoubleContainers();
          } else {
            return _buildSingleContainer();
          }
        },
      ),
    );
  }
  Widget _buildSingleContainer() {
    return Center(
      child: Container(
        height: 400.0,
        width: 100.0,
        color: Colors.red,
      ),
    );
  }
```

```
  Widget _buildDoubleContainers() {
    return Center(
      child: Row(
        mainAxisAlignment: MainAxisAlignment.spaceEvenly,
        children: <Widget>[
          Container(
            height: 400.0,
            width: 100.0,
            color: Colors.yellow,
          ),
          Container(
            height: 400.0,
            width: 100.0,
            color: Colors.yellow,
          ),
        ],
      ),
    );
  }

    Widget _buildTripleContainers() {
    return Center(
      child: Row(
        mainAxisAlignment: MainAxisAlignment.spaceEvenly,
        children: <Widget>[
          Container(
            height: 400.0,
            width: 100.0,
            color: Colors.green,
          ),
          Container(
            height: 400.0,
            width: 100.0,
            color: Colors.green,
          ),
          Container(
            height: 400.0,
            width: 100.0,
            color: Colors.green,
          ),
        ],
      ),
    );
  }
}
```

Discussion

LayoutBuilder can be used in situations where you need to understand the available onscreen constraints.

The LayoutBuilder widget provides an adaptive interface which is defined as an app running on different devices. The scope associated with this moves beyond screen dimensions and includes the hardware to be interrogated such as type of input, visual density and selection type.

Return a LayoutBuilder to provide the current context with constraints that can be queried dynamically. The typical use case for this functionality is to use the constraints structure to define thresholds. These thresholds can be used to marshal the available dimensions in an efficient manner. Use LayoutBuilder to enable smart composition of screens to be displayed based on interrogation of the device.

Contrast LayoutBuilder with the Media Query (Recipe 9.6) which takes into account the application size and orientation.

9.6 Getting screen dimensions with Media Query

Problem

You want to access the dimensions of a device.

Solution

Use the Media Query class to find the dimensions of a device. It will return properties such as the current width and height.

In the example we use the MediaQuery class to provide the dimensions on which the screen ratio is determined.

```
import 'package:flutter/material.dart';
void main() {
  runApp(const MyApp());
}
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);
```

```
  @override
  Widget build(BuildContext context) {
    const title = 'MediaQuery demo';
    return MaterialApp(
      title: title,
      home: Scaffold(
        appBar: AppBar(
          title: const Text(title),
        ),
        body: const MyMediaQueryWidget(),
      ),
    );
  }
}
class MyMediaQueryWidget extends StatelessWidget {
  const MyMediaQueryWidget({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        Text('Screen Width: ${MediaQuery.of(context).size.width}',
          style: const TextStyle(fontSize: 40, color: Colors.grey),
        ),
        Text('Screen Height:
${MediaQuery.of(context).size.height}',
          style: const TextStyle(fontSize: 40, color: Colors.grey),
        ),
        Text('Aspect Ratio:
${MediaQuery.of(context).size.aspectRatio}',
          style: const TextStyle(fontSize: 40, color: Colors.grey),
        ),
        Text('Orientation: ${MediaQuery.of(context).orientation}',
          style: const TextStyle(fontSize: 40, color: Colors.grey),
        ),
      ],
    );
  }
}
```

Discussion

A media query provides the ability to learn information about the device and the application. If you have performed web development previously, you will most likely be familiar with the meida query.Typically this class can be used to help with providing properties used to assist with having a responsive interface.

In the Flutter documentation, a key point raised is the difference between Responsiveness vs Adaptability[2]. The Bottom line is that responsiveness is attuned to the available screen size. If you intend to consider different device types then you are more likely to want to incorporate adaptability (Reference recipe 9.5) e.g. mouse, keyboard, component selection strategy.

MediaQuery also provides access to the aspectRatio and Orientation making it useful for tracking the device context. The supplied information is not without cost. When invoking MediaQuery.of, as per the example, will mean a rebuild of the widget tree if the properties change. For example if the viewport is enlarged or the orientation is changed.

---

[1] https://fonts.google.com/

[2] https://docs.flutter.dev/development/ui/layout/adaptive-responsive

Chapter 10. Organizing onscreen data

> **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 10th 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 editor at *jbleiel@oreilly.com*.

In this section we continue our journey with the Flutter Framework. Here we learn how to introduce some complementary techniques to take your application to the next level.

An important factor of developing Flutter applications is starting with the correct foundation. In many instances how the code is laid out will bring forward both strengths and weaknesses. Understanding when to use particular techniques or data structures will most definitely increase your enjoyment and efficiency when building with Flutter.

When creating more complex applications, always try to make Dart + Flutter do the hard work. Each iteration of the language and framework delivers better efficiencies. Incorporating these features will help you to avoid certain errors and encourage better coding practice.

Within this chapter you will learn about many of the fundamentals and tips that can be in your applications. Applications often have a number of moving parts and taking advantage of the existing patterns and approaches of Flutter will be highly beneficial to you as a developer.

10.1 Implementing a vertical list

Problem

You want to incorporate a list of items in a Flutter application.

Solution

In Flutter, a list of text items is typically rendered using the ListView widget. The ListView widget provides a simple mechanism for capturing data.

```
import 'package:flutter/material.dart';
class ListTileItem {
  final String monthItem;
  const ListTileItem({
    required this.monthItem,
  });
}
class ListDataItems {
    final List<ListTileItem> monthItems = [
    const ListTileItem(monthItem: 'January'),
    const ListTileItem(monthItem: 'February'),
    const ListTileItem(monthItem: 'March'),
    const ListTileItem(monthItem: 'April'),
    const ListTileItem(monthItem: 'May'),
    const ListTileItem(monthItem: 'June'),
    const ListTileItem(monthItem: 'July'),
    const ListTileItem(monthItem: 'August'),
    const ListTileItem(monthItem: 'September'),
    const ListTileItem(monthItem: 'October'),
    const ListTileItem(monthItem: 'November'),
    const ListTileItem(monthItem: 'December'),
  ];

  ListDataItems();
}
void main() {
  runApp(const MyApp());
}
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    const title = 'MyAwesome App';
```

```
    return MaterialApp(
      title: title,
      home: Scaffold(
        appBar: AppBar(
          title: const Text(title),
        ),
        body: MyListView(),
      ),
    );
  }
}
class MyListView extends StatelessWidget {
  MyListView();
  final ListDataItems item = ListDataItems();
  @override
  Widget build(BuildContext context) {
    return ListView.builder(
      itemCount: item.monthItems.length,
      itemBuilder: (context, index) {
        return MyListTile(item.monthItems[index]);
      },
    );
  }
}
class MyListTile extends StatelessWidget {
  const MyListTile(this.item);
  final ListTileItem item;
  @override
  Widget build(BuildContext context) {
    return ListTile(
      title: Text(item.monthItem),
      onTap: () {
        ScaffoldMessenger.of(context).showSnackBar(
          SnackBar(
            content: Text('You selected ${item.monthItem}'),
          ),
        );
      },
    );
  }
}
```

Discussion

For the majority of use cases where an on screen list is required, a ListView can be partnered with a builder. A combination of ListView and builder provides a memory efficient method for managing large volumes of data that are to be rendered on screen.

In the example you have a class defined for the data to be shown in the ListView. The class ListDataItems holds the information to be rendered on screen. Observe how the data is separate to the ListView definition, meaning the values held can be changed without impacting the functionality used to populate the ListView.

The ListView function defines a local variable that instantiates the data class. Once initiated the ListView can then use the Builder design pattern to populate the list with the contents of the data class.

A ListTile widget function is created to isolate the onscreen rendering of the data. The widget uses a ListTile and is passed an index data item that will be shown on screen. Finally an onTap method is declared so that each ListView item will display a notification when the user taps the onscreen value.

The data element to data declaration from the rendering processing required for a list.

In the example code the ListView is declared to use an existing list defined in a separate class. To render items, the ListView expects its data to be presented as a List. Reference recipe 2.7 for information on how to declare a list. The ListTileItem provides a group of information that will be displayed as the contents for the ListTile widget used by ListView.

10.2 Implementing a horizontal list

Problem

You want to create a horizontal list of items

Solution

Creating a horizontal list can be performed in a similar manner to theway we create a vertical list (as done in recipe 10.1).. However in most use cases, you will want the list to react dynamically (i.e. height and width) based on the contents to be displayed.

```
import 'package:flutter/material.dart';
class MenuItem {
  final String title;
  final String subtitle;
  final String url;
  const MenuItem({
    required this.title,
    required this.subtitle,
    required this.url,
  });
}
class ListDataItems {
  final List<MenuItem> menuItems = [
    const MenuItem(
      title: 'Burger #1',
      subtitle: 'House Special',
      url:
          'https://images.unsplash.com/photo-1499028344343-
cd173ffc68a9?ixlib=rb-
1.2.1&ixid=MnwxMjA3fDB8MHxleHBsb3JlLWZlZWR8MTV8fHxlbnwwfHx8fA%3D%3D
&auto=format&fit=crop&w=500&q=60',
    ),
    const MenuItem(
      title: 'Burger #2',
      subtitle: 'Tall Boy Special',
      url:
          'https://images.unsplash.com/photo-1542574271-
7f3b92e6c821?ixlib=rb-
1.2.1&ixid=MnwxMjA3fDB8MHxzZWFyY2h8MTN8fGJ1cmdlcnxlbnwwfHwwfHw%3D&a
uto=format&fit=crop&w=500&q=60',
    ),
    const MenuItem(
      title: 'Burger #3',
      subtitle: 'Gastro Special',
      url:
          'https://images.unsplash.com/photo-1596662951482-
0c4ba74a6df6?ixlib=rb-
1.2.1&ixid=MnwxMjA3fDB8MHxzZWFyY2h8N3x8YnVyZ2VyfGVufDB8fDB8fA%3D%3D
&auto=format&fit=crop&w=500&q=60',
    ),
    const MenuItem(
      title: 'Burger #4',
      subtitle: 'Chicken Special',
      url:
          'https://images.unsplash.com/photo-1551782450-
17144efb9c50?ixlib=rb-
1.2.1&ixid=MnwxMjA3fDB8MHxzZWFyY2h8MTB8fGJ1cmdlcnxlbnwwfHwwfHw%3D&a
uto=format&fit=crop&w=500&q=60',
    ),
  ];
```

```
  ListDataItems();
}
void main() {
  runApp(const MyApp());
}
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    const title = 'Horizontal List';
    return MaterialApp(
      title: title,
      home: Scaffold(
        appBar: AppBar(
          title: const Text(title),
        ),
        body: const MyHorizontalListView(),
      ),
    );
  }
}
class MyHorizontalListView extends StatelessWidget {
  const MyHorizontalListView({Key? key}) : super(key: key);
  //final index = listMonthItems.length;
  @override
  Widget build(BuildContext context) {
    return MySizedBox();
  }
}
class MySizedBox extends StatelessWidget {
  MySizedBox({Key? key}) : super(key: key);

  final ListDataItems item = ListDataItems();

  final itemWidth = 12.0;
  @override
  Widget build(BuildContext context) {
    return SizedBox(
      height: 250,
      child: ListView.separated(
        scrollDirection: Axis.horizontal,
        itemCount: item.menuItems.length,
        separatorBuilder: (context, _) => SizedBox(width:
itemWidth),
        itemBuilder: (context, index) =>
MyListViewItem(item.menuItems[index]),
      ),
    );
  }
}
```

```
class MyListViewItem extends StatelessWidget {
  const MyListViewItem(this.item);
  final MenuItem item;
  @override
  Widget build(BuildContext context) {
    return SizedBox(
      width: 200,
      child: Column(
        children: [
          Expanded(
            child: AspectRatio(
              aspectRatio: 4 / 3,
              child: Image.network(
                item.url,
                fit: BoxFit.cover,
              ),
            ),
          ),
          Text(
            item.title,
            style: const TextStyle(fontSize: 20, color:
Colors.black),
          ),
          Text(
            item.subtitle,
            style: const TextStyle(fontSize: 16, color:
Colors.grey),
          ),
        ],
      ),
    );
  }
}
```

Discussion

In the solution you create a row object which will pass information to a widget. To ensure the list presented in the screen is respectful of position and content use a ‘SingleChildScrollView’ together with a Row widget.

The SingleChildScrollView is a scrollable widget for multi-axis directional content. In this instance it is useful for a horizontal scrollable list as content to be displayed will require a degree of flexibility.

To simplify the code, the imageItem displayed has been extracted to its own function. Readability can often be increased by refactoring code to simple

widget functions.

10.3 Adding a SliverAppBar

Problem

You want to create a responsive header area for an application based on user scrolling activity.

Solution

Add SliverAppBar(floating: true,) to make the AppBar reappear when the user screens up. The default is to not show the app bar until the top of the list is present. Here’s how to add the SilverAppBar:

```
import 'package:flutter/material.dart';
class CarItem {
  final String title;
  final String subtitle;
  final String url;
  CarItem({
    required this.title,
    required this.subtitle,
    required this.url,
  });
}
class ListDataItems {
    final List<CarItem> carItems = [
    CarItem(
        title: '911 Cabriolet',
        subtitle: '911 Carrera Cabriolet Porsche',
        url:

'https://files.porsche.com/filestore/image/multimedia/none/992-
c2cab-modelimage-sideshot/thumbwhite/9806daa1-d97f-11eb-80d9-
005056bbdc38;sD;twebp/porsche-thumbwhite.webp'),
    CarItem(
        title: '718 Spyder',
        subtitle: '718 Spyder Porsche',
        url:

'https://files.porsche.com/filestore/image/multimedia/none/982-
718spyder-modelimage-sideshot/thumbwhite/e9f11134-fa4e-11e9-80c6-
005056bbdc38;sD;twebp/porsche-thumbwhite.webp'),
```

```
    CarItem(
        title: '718 Boxster T',
        subtitle: '718 Boxster T Porsche',
        url:

'https://files.porsche.com/filestore/image/multimedia/none/982-718-
bo-t-modelimage-sideshot/thumbwhite/70f6828b-ac0d-11eb-80d5-
005056bbdc38;sO;twebp/porsche-thumbwhite.webp'),
    CarItem(
        title: 'Cayenne',
        subtitle: 'Cayenne S Porsche',
        url:

'https://files.porsche.com/filestore/image/multimedia/none/9ya-e3-
s-modelimage-sideshot/thumbwhite/e814b368-a8d3-11eb-80d5-
005056bbdc38;sO;twebp/porsche-thumbwhite.webp'),
  ];

  ListDataItems();
}
void main() => runApp(MyApp());
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Cookbook Demo',
      debugShowCheckedModeBanner: false,
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: const MyHomePage(title: 'Flutter Cookbook'),
    );
  }
}
class MyHomePage extends StatelessWidget {
  final String title;
  const MyHomePage({
    Key? key,
    required this.title,
  }) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: Colors.grey[300],
      body: CustomScrollView(
        slivers: [
          const SliverAppBar(
            leading: Icon(Icons.menu),
            title: Text('Sliver App Bar + List'),
            expandedHeight: 300,
```

```
            collapsedHeight: 150,
          ),
          // Next, create a SliverList
          MySliverList(),
        ], // End
      ),
    );
  }
}
class MySliverList extends StatelessWidget {
  MySliverList();

  final ListDataItems item = ListDataItems();
  @override
  Widget build(BuildContext context) {
    return // Next, create a SliverList
        SliverList(
      // Use a delegate to build items as they're scrolled on
screen.
      delegate: SliverChildBuilderDelegate(
        (context, index) => MyListTile(item.carItems[index]),
        // Builds 1000 ListTiles
        childCount: item.carItems.length,
      ),
    );
  }
}
class MyListTile extends StatelessWidget {
  const MyListTile(this.carItem);
  final CarItem carItem;
  @override
  Widget build(BuildContext context) {
    return ListTile(
      leading: CircleAvatar(
        backgroundImage: NetworkImage(carItem.url),
      ),
      title: Text(carItem.title),
      subtitle: Text(carItem.subtitle),
    );
  }
}
```

Discussion

If you are familiar with a ListView/Builder pattern, using a SliverAppBar with a List should be a familiar approach. The SliverAppBar widget provides a nice way to add a scroll context to an application.

When using a SliverAppBar, you as the developer take responsibility for the on screen widget management. In this instance, to establish a scrollable interface, a CustomScrollView is added to handle this processing.

The usual AppBar widget is removed from the example and replaced by SliverAppBar. Initially we use a SliverAppBar that will shrink as the user scrolls up and down the application viewport. The SliverAppBar header is attached to the screen and reacts to user interaction.

From the example, note how a SliverList rather than a ListView is used for the processing of the information. The SliverList is used in conjunction with a SliverAppBar and will dynamically adjust based on the content to be shown on screen. In addition to this list processing a special SliverChildBuilderDelegate function is used to process the items to be viewed.

10.4 Adding a grid of items

Problem

You want a way to display a grid of items.

Solution

In the example below there are two types of GridView rendered

```
import 'package:flutter/material.dart';
void main() {
  runApp(const MyApp());
}
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    const title = 'Drawer demo';
    return MaterialApp(
      title: title,
      home: Scaffold(
        appBar: AppBar(
          title: const Text(title),
        ),
```

```
        body: const MyGridViewBuilderWidget(),
      ),
    );
  }
}
class MyGridViewWidget extends StatelessWidget {
  const MyGridViewWidget({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return GridView.count(crossAxisCount: 2, children: [
      (Padding(
        padding: const EdgeInsets.all(8.0),
        child: Container(
          height: 50,
          width: 50,
          color: Colors.blue,
        ),
      )),
      (Padding(
        padding: const EdgeInsets.all(8.0),
        child: Container(
          height: 50,
          width: 50,
          color: Colors.blue,
        ),
      )),
      (Padding(
        padding: const EdgeInsets.all(8.0),
        child: Container(
          height: 50,
          width: 50,
          color: Colors.blue,
        ),
      )),
      (Padding(
        padding: const EdgeInsets.all(8.0),
        child: Container(
          height: 50,
          width: 50,
          color: Colors.blue,
        ),
      )),
    ]);
  }
}
class MyGridViewBuilderWidget extends StatelessWidget {
  const MyGridViewBuilderWidget({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return GridView.builder(
```

```
        physics: const NeverScrollableScrollPhysics(),
        itemCount: 10,
        gridDelegate:
            const
SliverGridDelegateWithFixedCrossAxisCount(crossAxisCount: 5),
        itemBuilder: (context, index) {
          return Padding(
              padding: const EdgeInsets.all(8.0),
              child: Container(
                height: 50,
                width: 50,
                color: Colors.blue,
              ));
        });
  }
}
```

Discussion

If you have used a ListView/Builder combination previously, hopefully using a GridView will be familiar to you. The GridView.builder follows the same principles as a ListView Builder. .

GridView.count

As with a ListView, you need to indicate how many items are to be displayed as part of the GridView. The count property is used to indicate the quantity to be displayed and should be set accordingly. In the example, we indicated that X items should be displayed.

The addition of NeverScrollableScrollPhysics prevents a scrollable area for the user.

The display of the widget can be constructed as you wish. In the example code Padding and a Container widget are used for the data to be displayed.

10.5 Adding a Snackbar (Popup notification)

Problem

You want a way to display a short lived notification to the user.

Solution

Brief notifications to the user are facilitated by a snackbar. A snackbar will momentarily be presented on screen with the designated text.

```
import 'package:flutter/material.dart';
void main() {
  runApp(const MyApp());
}
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    const title = 'MyAwesome App';

    return MaterialApp(
      title: title,
      home: Scaffold(
        appBar: AppBar(
          title: const Text(title),
        ),
        body: MyListView(),
      ),
    );
  }
}
class ListTileItem {
  final String monthItem;
  const ListTileItem({
    required this.monthItem,
  });
}
class MyListView extends StatelessWidget {
  MyListView();
  final List<ListTileItem> monthItems = [
    const ListTileItem(monthItem: 'January'),
    const ListTileItem(monthItem: 'February'),
    const ListTileItem(monthItem: 'March'),
    const ListTileItem(monthItem: 'April'),
    const ListTileItem(monthItem: 'May'),
    const ListTileItem(monthItem: 'June'),
    const ListTileItem(monthItem: 'July'),
    const ListTileItem(monthItem: 'August'),
    const ListTileItem(monthItem: 'September'),
    const ListTileItem(monthItem: 'October'),
    const ListTileItem(monthItem: 'November'),
    const ListTileItem(monthItem: 'December'),
  ];
  @override
  Widget build(BuildContext context) {
```

```
    return ListView.builder(
      itemCount: monthItems.length,
      itemBuilder: (context, index) {
        return MyListTile(monthItems[index]);
      },
    );
  }
}
class MyListTile extends StatelessWidget {
  const MyListTile(this.item);
  final ListTileItem item;
  @override
  Widget build(BuildContext context) {
    return ListTile(
      title: Text(item.monthItem),
      onTap: () {
        ScaffoldMessenger.of(context).showSnackBar(
          SnackBar(
            content: Text('You selected ${item.monthItem}'),
          ),
        );
      },
    );
  }
}
```

Discussion

A snackbar is useful where you need to indicate an activity has been performed. For example, indicating a button has been tapped or a file has been downloaded.

The SnackBar widget provides a notification back to the user. You can override the default settings of the snackbar as desired. For example the default duration of 4 secs can be overridden, by introducing the following code:

```
            SnackBar(
              duration: const Duration(seconds: 10, milliseconds:
500),
              content: Text('You selected $listTitle'),
            ),
```

In addition to providing feedback to the user, the snackbar can also perform additional actions. Extend the snackbar definition to include a label and a gesture to initiate a complementary action.

```
SnackBar(
  action: SnackBarAction(
    label: 'action',
    onPressed: () {},

  ),
  duration: const Duration(seconds: 10, milliseconds:
500),
  content: Text('You selected $listTitle'),
),
```

You can utilize this feature to invoke additional activities similar to that seen with the more common menu based options.

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*.

Table of Contents

- Cover
- Flutter and Dart Cookbook
- Copyright
  - Revision History for the Early Release
- Chapter 1. Starting with Dart
  - A Note for Early Release Readers
  - 1.1 Determining which Dart installation to use
    - Problem
    - Solution
    - Discussion
  - 1.2 Running Dart in DartPad
    - Problem
    - Solution
    - Discussion
      - Figure 1-1. The DartPad interface
  - 1.3 Extending Android Studio to support Dart
    - Problem
    - Solution
    - Discussion
  - 1.4 Developing with VS Code
    - Problem
    - Solution
    - Discussion
  - 1.5 Installing the Dart SDK
    - Problem
    - Solution
    - Discussion
  - 1.6 Running a Dart application
    - Problem
    - Solution
    - Discussion
  - 1.7 Selecting a release channel
    - Problem

- Solution
- Discussion

Chapter 2. Learning Dart Variables
- A Note for Early Release Readers
- 2.1 Declaring an Integer variable
  - Problem
  - Solution
  - Discussion
- 2.2 Declaring a Double variable
  - Problem
  - Solution
  - Discussion
- 2.3 Declaring a Bool variable
  - Problem
  - Solution
  - Discussion
- 2.4 Declaring a String variable
  - Problem
  - Solution
  - Discussion
- 2.5 Using a Print statement
  - Problem
  - Solution
    - Example 2-5. Example 1: Here’s an example of a how to print static content:
    - Example 2-6. Example 2: Here’s an example of a how print the content of a variable:
    - Example 2-7. Example 3: Here’s an example of a how print the complex data type:
  - Discussion
- 2.6 Using a Const
  - Problem
  - Solution
  - Discussion
- 2.7 Using Final
  - Problem
  - Solution

- Discussion
- 2.8 Working with Null
  - Problem
  - Solution
  - Discussion
- Chapter 3. Exploring Control Flow
  - A Note for Early Release Readers
  - 3.1 Using an If statement
    - Problem
    - Solution
    - Discussion
  - 3.2 Using While/Do While
    - Problem
    - Solution
    - Discussion
  - 3.3 Using a For statement
    - Problem
    - Solution
    - Discussion
  - 3.4 Using a Switch statement
    - Problem
    - Solution
    - Discussion
  - 3.5 Using an Enum
    - Problem
    - Solution
    - Discussion
  - 3.6 Handling Exceptions
    - Problem
    - Solution
    - Discussion
- Chapter 4. Implementing Functions
  - A Note for Early Release Readers
  - 4.1 Declaring Functions
    - Problem
    - Solution
    - Discussion

- 4.2 Adding parameters to Functions
  - Problem
  - Solution
  - Discussion
- 4.3 Returning values from Functions
  - Problem
  - Solution
  - Discussion
- 4.4 Declaring Anonymous functions
  - Problem
  - Solution
  - Discussion
- 4.5 Using optional parameters
  - Problem
  - Solution
  - Discussion

Chapter 5. Handling Maps and Lists

- A Note for Early Release Readers
- 5.1 Using a Map to handle objects
  - Problem
  - Solution
  - Discussion
- 5.2 Retrieving Map content
  - Problem
  - Solution
  - Discussion
- 5.3 Validating key existence within a Map
  - Problem
  - Solution
  - Discussion
- 5.4 Working with Lists
  - Problem
  - Solution
  - Discussion
- 5.5 Adding List content
  - Problem
  - Solution

- Discussion
- 5.6 Using Lists with complex types
  - Problem
  - Solution
  - Discussion
- Chapter 6. Leveraging Classes
  - A Note for Early Release Readers
  - 6.1 Defining Classes
    - Problem
    - Solution
    - Discussion
  - 6.2 Using Class Constructors
    - Problem
    - Solution
    - Discussion
  - 6.3 Extending Classes
    - Problem
    - Solution
    - Discussion
  - 6.4 Extending Classes with Mixins
    - Problem
    - Solution
    - Discussion
  - 6.5 Importing a package
    - Problem
    - Solution
    - Discussion
- Chapter 7. Introducing the Flutter Framework
  - A Note for Early Release Readers
  - 7.1 Mocking an interface
    - Problem
    - Solution
    - Discussion
      - Figure 7-1. Example Excalidraw drawing
  - 7.2 Creating a Flutter project
    - Problem
    - Solution

- Note
- Discussion
- 7.3 Working with a Stateful Widget
  - Problem
  - Solution
  - Discussion
    - Figure 7-2. Example stateful widget interaction
- 7.4 Working with a Stateless Widget
  - Problem
  - Solution
  - Discussion
- 7.5 Refactoring Widgets
  - Problem
  - Solution
  - Discussion
- 7.6 Removing the Flutter Debug banner
  - Problem
  - Solution
  - Discussion

Chapter 8. Working with Widgets
- A Note for Early Release Readers
- 8.1 Using the Scaffold class
  - Problem
  - Solution
  - Discussion
- 8.2 Using an AppBar
  - Problem
  - Solution
  - Discussion
- 8.3 Using an Expanded widget
  - Problem
  - Solution
  - Discussion
    - Figure 8-1. A ListView widget
    - Figure 8-2. A ListView widget enclosed within an Expanded widget
- 8.4 Building with a Container

- Problem
- Solution
- Discussion
- 8.5 Using a Center widget
  - Problem
  - Solution
  - Discussion
- 8.6 Using a SizedBox
  - Problem
  - Solution
  - Discussion
- 8.7 Using a Column
  - Problem
  - Solution
  - Discussion
- 8.8 Using a Row
  - Problem
  - Solution
  - Discussion

Chapter 9. Developing User Interfaces
- A Note for Early Release Readers
- 9.1 Incorporating Rich Text
  - Problem
  - Solution
  - Discussion
- 9.2 Incorporating the Google fonts package
  - Problem
  - Solution
  - Discussion
- 9.3 Identifying the host platform
  - Problem
  - Solution
  - Discussion
- 9.4 Using a Placeholder widget
  - Problem
  - Solution
  - Discussion

- 9.5 Using a Layout Builder
  - Problem
  - Solution
  - Discussion
- 9.6 Getting screen dimensions with Media Query
  - Problem
  - Solution
  - Discussion

Chapter 10. Organizing onscreen data
- A Note for Early Release Readers
- 10.1 Implementing a vertical list
  - Problem
  - Solution
  - Discussion
- 10.2 Implementing a horizontal list
  - Problem
  - Solution
  - Discussion
- 10.3 Adding a SliverAppBar
  - Problem
  - Solution
  - Discussion
- 10.4 Adding a grid of items
  - Problem
  - Solution
  - Discussion
- 10.5 Adding a Snackbar (Popup notification)
  - Problem
  - Solution
  - Discussion

About the Author