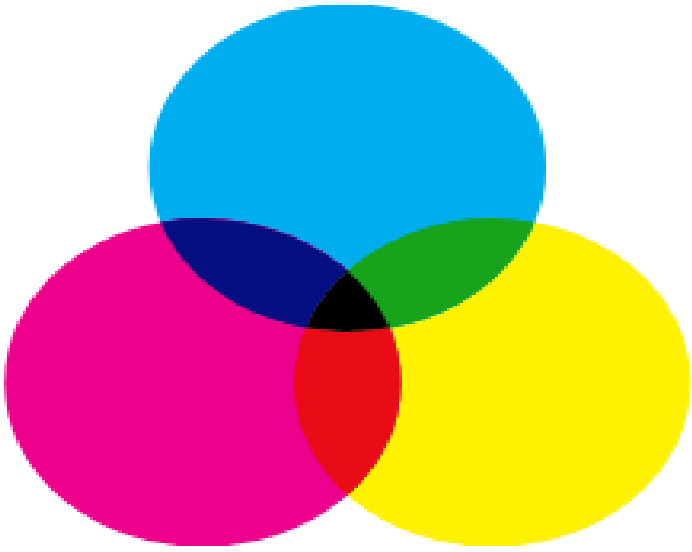
A STEP BY STEP GUIDE FOR BEGINNERS TO BUILD A BASIC ANDROID OR IOS MOBILE APPLICATION



BEGINNING FLUTTER WITH DART

LEARN TO CODE YOUR ASESOME UI DESIGN

SANJIB SINHA

Beginning Flutter with Dart

A Step by Step Guide for Beginners to Build a Basic Android or iOS Mobile Application

Sanjib Sinha

This book is for sale at http://leanpub.com/beginningflutterwithdart

This version was published on 2020-11-19

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

1. Getting Started

To start with, we need to download the Flutter framework.

That is our first task. We need to go to The installation page of Flutter[1] page from where we will download and install Flutter according to your operating system.

We will start with Windows, first.

Want to read more Flutter related Articles and resources? For more Flutter related Articles and Resources[2]

Before that we want to make one thing clear.

Who should read this book?

Are you an absolute beginner who without having any prior knowledge of programming language wants to build a mobile application? Well, then this book is for you. This book is not for intermediate or experienced learners or developers.

We will try to add two things to our knowledge so that we will start building your mobile application. First we will learn Flutter, a framework or tool that helps us to build the mobile application. Second, we will learn a programming language called Dart, with which Flutter works.

If we do not understand the basic syntax and semantics of Dart, we will not be able to understand the internal activities of Flutter.

We will learn both, Flutter and Dart side by side. For instance, if we find something like function and object or named parameters in a constructor, we will learn that concept in Dart.

If you have no knowledge of programming language, or you have not written a single line of code, you need not worry. We will go very smooth, we will have plenty of screenshots that will explain what we are going to do. We will also learn the basic concepts of programming language through Dart; it is important, because otherwise we will not be able to understand how Flutter framework works.

If you have any question, please do not hesitate to send me a mail at: <sanjib12sinha@gmail.com >

[1] https://flutter.dev/docs/get-started/install
[2] https://zerodotone.net

Flutter for Windows

Clicking the download button will automatically start downloading ziped Flutter in your Download folder. It would be around 700 MB in size. While extracting the file it would take around 1.30 GB place of your hard drive. You may copy that extracted file to elsewhere, or you may keep it there (figure 1.1).

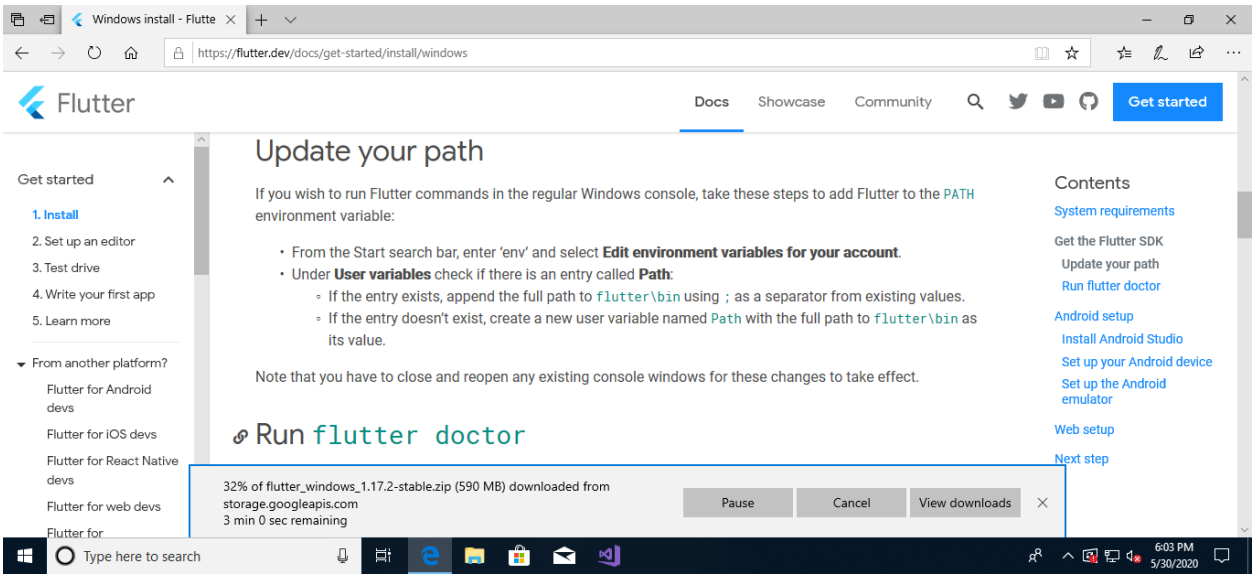


Figure 1.1 – Downloading Flutter for Windows

We have kept the extracted flutter folder there and created a new 'environment' path for the user. Because we want to work through the command prompt, in future, we have created this global environment path. Creating a new environment variable path in any Windows operating system is also easy. In the Windows 10 operating system, we type 'environment variable' in the search prompt, it will automatically open up the related window for us.

We can copy and paste the whole path there as the following:

```
1 "C:\Users\Downloads\flutter\bin".
```

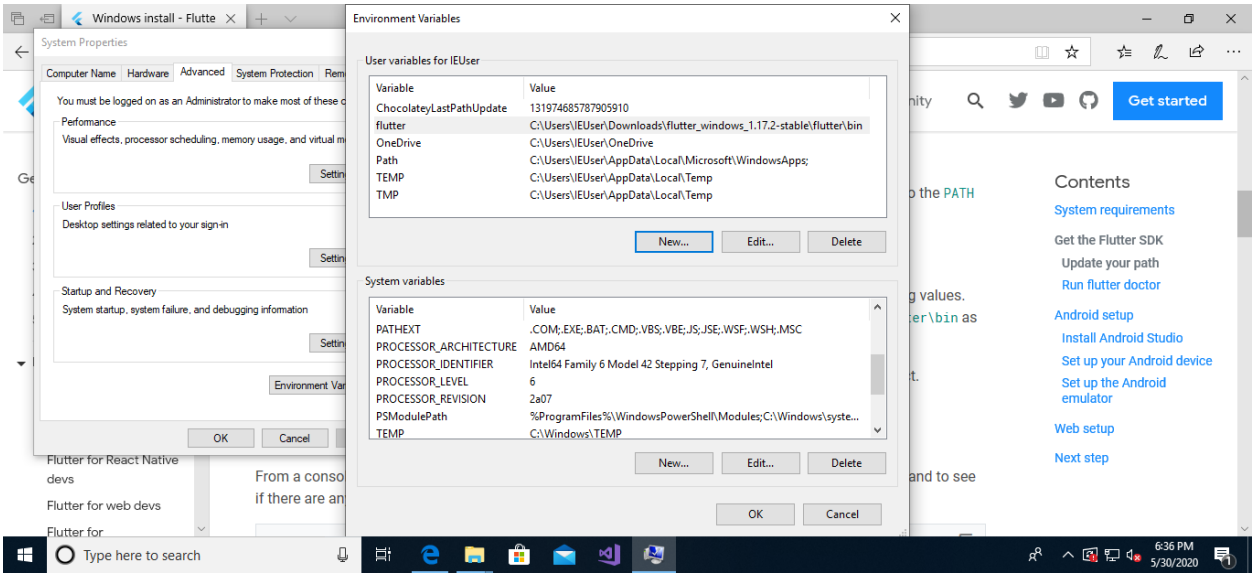


Figure 1.2 – Creating the new environment variable path in Windows 10

Now, we can open the command prompt and type 'flutter doctor' to see whether we have any Flutter related IDE installed already. It will also check whether we have any connected device or not.

We have not installed Android Studio or any other Flutter related IDE beforehand. The command 'flutter doctor' has detected that (Figure 1.3).

To work with Flutter, we need a good IDE. In fact, when we were downloading Flutter, it indicated that we should install Android Studio or any good IDE where we would have a connected device. The connected device is nothing but a virtual mobile device where we can see and test our mobile application.

Android Studio should be the best choice. It is widely used and Flutter home page also suggests to download and install that IDE.

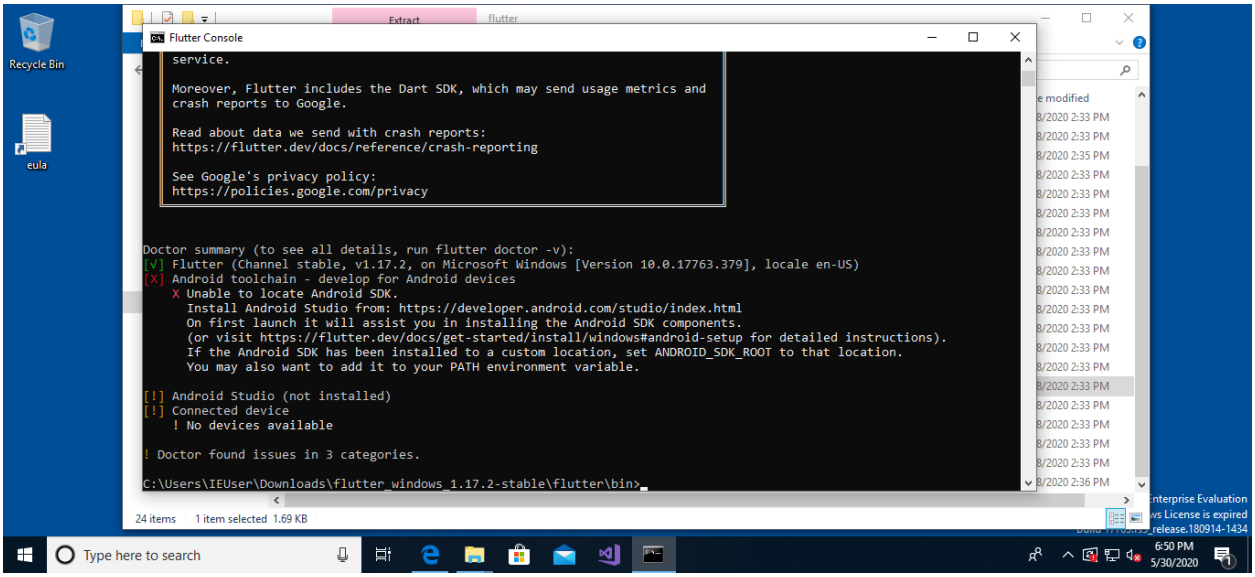


Figure 1.3 – Flutter Doctor Summary in Windows 10

In Flutter Doctor summary, we have found that Android Studio has not been installed and there is no device available.

We will do that in our macOS and Linux machines, because we will use any one of that operating system to learn Flutter and Dart together.

Flutter for macOS and Linux

Downloading Flutter for macOS and Linux is same. It will download the “flutter_linux_1.17.2-stable.tar.xz” file in your “Downloads” folder.

Next we will issue the following command to extract Flutter, on our terminal:

```
1  //code 1.1
2  tar xf flutter_linux_1.17.2-stable.tar.xz
```

Now we can copy this extracted ‘flutter’ directory to a suitable place, where we will build our first mobile application. In the ‘Documents’ directory, we have created another directory named ‘development’. We will keep the extracted ‘flutter’ directory there.

Just like Windows 10, we will now set the global path for ‘flutter’, so that we can use ‘flutter’ command, anywhere in our machine, in the future.

We will do that using ‘vim’ or ‘nano’ text editor, that works on the terminal. By the way, the commands are same for any macOS or Linux operating system.

If you type the following command, the nano text editor will open up the ‘bashrc’ file.

```
//code 1.2
nano ~/.bashrc
```

At the end of the 'bashrc' file we will add this line:

```
//code 1.3

export PATH=$PATH:/home/ss/Documents/development/flutter/bin:$PATH
```

We have to mention the full path as given above. We have kept our extracted 'flutter/bin' folder in the '/home/ss/Documents/development' directory.

Our next step will be to download the Android Studio. Download the zipped folder and extract it anywhere in the machine. We have kept it in our '/home/' directory. Next, issue this command:

```
//code 1.4
ss@ss-desktop:~$ cd android-studio/bin/
ss@ss-desktop:~/android-studio/bin$ ./studio.sh
```

It will open up the Android Studio for us (figure 1.4). Once the Android Studio opens up, you can go to the 'open folder' option and choose the flutter project we have created already. How we have created it, we will come to that point in a minute.

Before that, we need to see the Android Studio and our newly created virtual device.

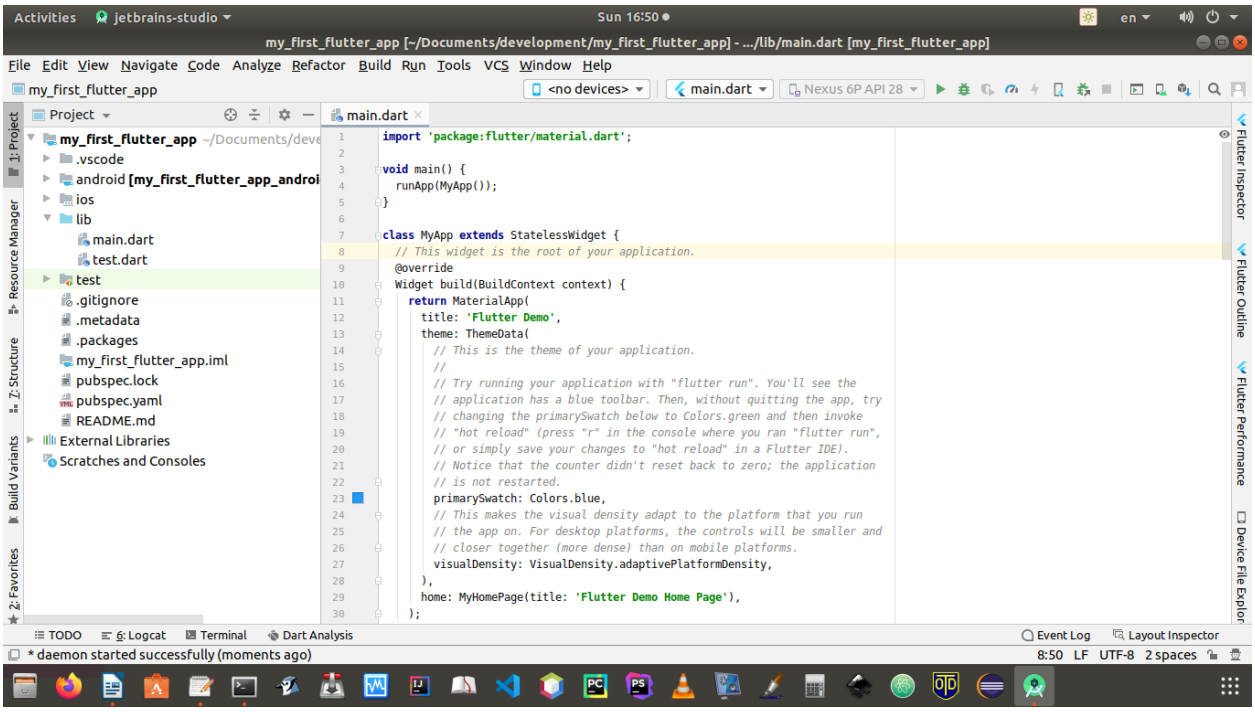


Figure 1.4 – Android Studio and our first flutter project

Before opening the Android Studio, we have opened up our terminal, and typed the following commands to reach to the newly installed 'flutter' directory.

```
//code 1.5
ss@ss-desktop:~$ cd Documents/development/flutter/
ss@ss-desktop:~/Documents/development/flutter$ flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel stable, v1.17.2, on Linux, locale en_IN)

[✓] Android toolchain - develop for Android devices (Android SDK version 29.0.3)
[✓] Android Studio (version 3.5)
[✓] Android Studio (version 4.0)
[✓] IntelliJ IDEA Community Edition (version 2019.3)
[✓] VS Code (version 1.43.2)
[!] Connected device
    ! No devices available

! Doctor found issues in 1 category.
ss@ss-desktop:~/Documents/development/flutter$
```

As you have seen in the above output, 'flutter doctor' has found only one issue. It has not found any connected device. Otherwise, we have already installed Android Studio (version 4.0), which is the latest at the time of writing this book. We have also installed IntelliJ IDEA Community Edition, and we have also Visual Studio Code IDE.

We can use the virtual device from Android Studio, but we can use the Visual Studio Code IDE or IntelliJ IDEA Community Edition IDE for writing our code. They will automatically synchronize with the connected device. However, before that we need to create our first flutter project with the help of flutter command as the following:

```
//code 1.6
flutter create my_first_flutter_app
```

Remember one thing. When we want to create a new flutter project, we should always create like this. The naming convention is important here. We can only use the underscore between the words. No hyphen or space is allowed.

Now the time has come to go back to the Android Studio. We will pick up the 'open folder' option and choose to open the newly created flutter project. We have named it as: 'my_first_flutter_app'.

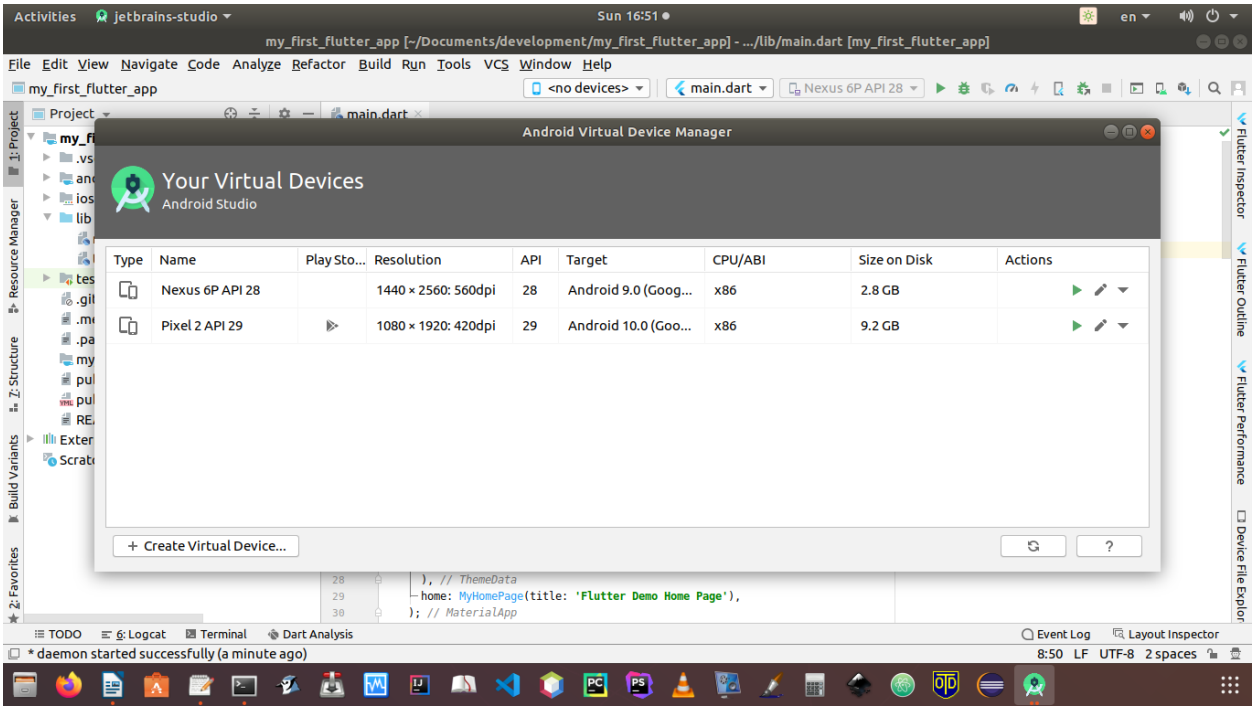


Figure 1.5 – Open the Android Virtual Device (AVD) manager from tools menu

To open up the connected device, we need to open the Android Virtual Device manager, or AVD manager in short.

You will get that from the 'tools' menu.

Select any one of them and click the 'green' play button on the far right hand side of any virtual device. It will automatically open up the 'connected device' (Figure 1.6).

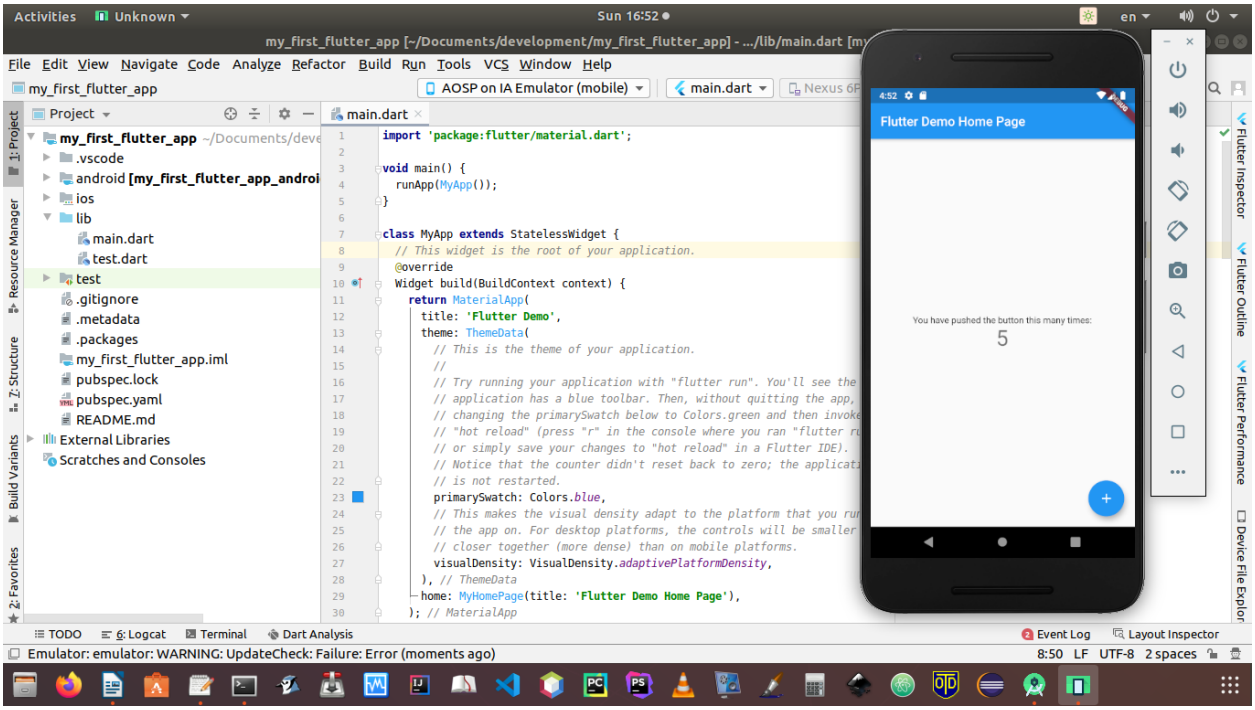


Figure 1.6 – We have the connected device on which we can test our first mobile application

Now everything is ready. We can start building our first mobile application from scratch using Flutter and Dart. Before closing down this section, we should know a few good tips. Usually, the beginners encounter a few errors when they try to run the command:

```
flutter doctor
```

If it gives any error, try this command:

```
flutter doctor --android-licenses
```

It will ask you to accept the license. Accept it, and it will not give any error anymore. Another problem often gives trouble to the new developers.

As a beginning Flutter developer, people often are stuck with this issue. They cannot launch the virtual mobile device while working with Android Studio.

We want that every code we write should reflect on the virtual device. It can be done by going to the 'AVD manager' from tools. But sometimes an ugly error pops up its head and tells that '/dev/kvm permission denied'. In Ubuntu 18 or Mac OS, you can give user the permission by issuing this command:

```
//code 1.7
sudo chmod 777 -R /dev/kvm
```

But it has a drawback. If someone else uses your machine, then the other user also gets the permission. The best remedy is – give permission to yourself only by the following commands:

```
//code 1.8
sudo apt install qemu-kvm
sudo adduser your-username kvm
sudo chown your-username /dev/kvm
```

It will solve the issue for ever. Now you can launch any virtual device you want. You can launch the device with your Android Studio, and work with any other IDE like IntelliJ or Visual Studio.

Relation between Flutter and Dart

We have found out that Flutter is a framework or tool that we need to create beautiful mobile applications. Flutter is written in Dart programming language. To understand how Flutter works, we need to understand Dart also.

Before digging deep to find out the relation between Flutter and Dart, let us try to understand one key concept of programming. There are two distinct parts of programming. One is abstraction and the other is concretion. We need to convert our abstract ideas into concretion, or a concrete shape or form.

Any mobile application is an abstract idea. We need a tool like Flutter to give it a concrete shape. Take a more real life example. Justice is an abstraction, but law is a tool. When we say that 'justice is done', the abstract idea of Justice gets a concrete shape. And it is 'done' by using the tool called 'law'.

We hope that now we get a more clear picture why we need a tool like Flutter. Because we want to convert our abstract idea of making a mobile application we need a tool like Flutter.

While we use Flutter, we will encounter many terms like function, class, constructor, positional parameter, named parameter, object, Widget, etc, etc.

As an absolute beginner if you search the Internet, you will find that before Flutter developers used either Ionic or React Native to build mobile applications. Android developers used Java also. You can use Java to build Android application. Therefore, Flutter is not doing anything new. People used to do that before using other tools. Reading until this point, we may ask, then why we should learn Flutter. We could have learned something else,some other tools. Some other languages.

This question is pertinent to our discussion.

Flutter has some benefits. You enjoy some more privileges not enjoyed by other developers. To use the Flutter tool, you need to learn one programming language – Dart, and Flutter has a single code-base that can be used to build Android and native iOS mobile application.

It is a specialty, special advantage not enjoyed by all who use other tools.

Therefore, as an absolute beginner, you need to remember that Dart is a programming language. And Flutter uses Dart language building mobile applications on top of Dart platform. Flutter has some more components, such as Software Development Kit or SDK, flutter engine, foundation libraries and Widgets that are design specific.

As a programming language, Dart has other functionalities.

We can build web or desktop applications with Dart. Feel free to differ, but Dart seems to be a mixture of C and Java. If you have already learned these two languages, Dart will appear to be less daunting. Flutter runs in the Dart virtual machine. Just like we have seen earlier in Java Virtual Machine or JVM.

We do not want to be more specific on Flutter internals, as this book is aimed for the absolute beginners.

To be more specific on how Flutter and Dart work together, we will create another Dart project in our IntelliJ IDEA Community Edition IDE. In that Dart project we will learn Dart simultaneously as we progress with Flutter in a different project. We hope that will make sense.

To create a separate Dart project, we will open our IntelliJ IDE and add the Dart and Flutter plugins first. After that, we will open up our IntelliJ IDE to create a console based Dart application (Figure 1.7).

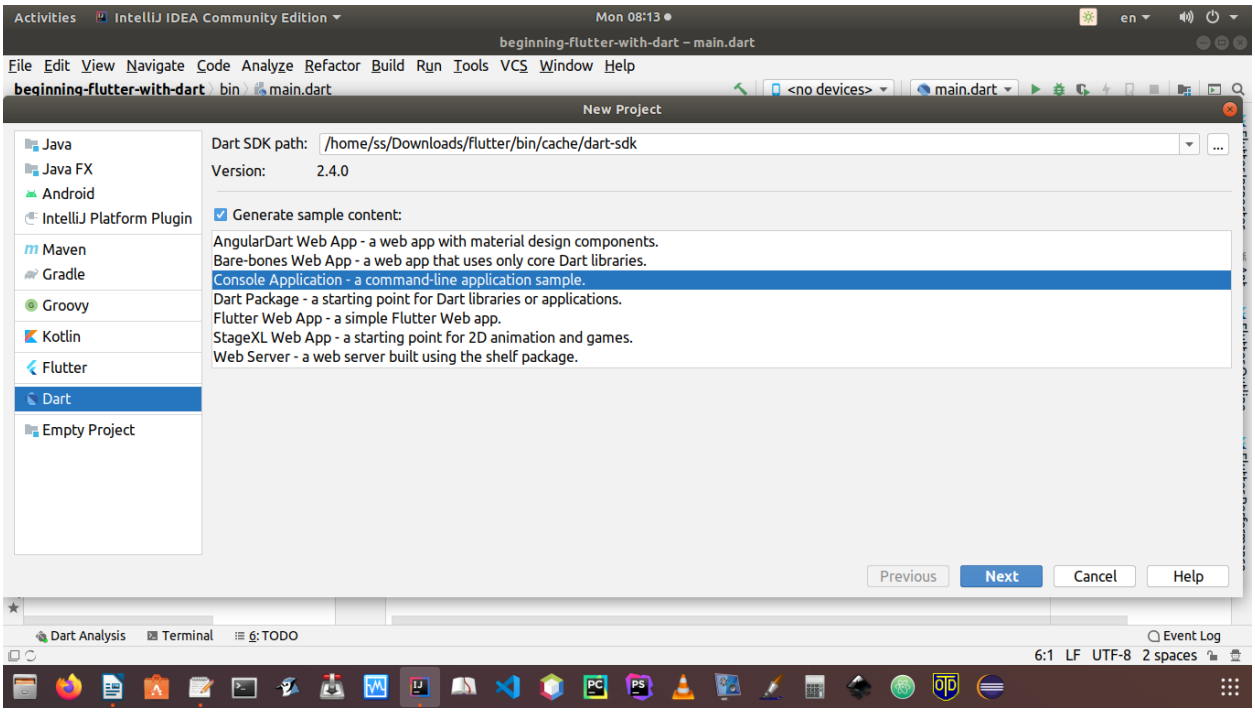


Figure 1.7 – A Dart Console project in IntelliJ IDE

As in the above image, we are going to create a Dart Console application. In this command line application sample, we will write different type of Dart code to learn the language basics.

The language basics is enough to give us a brief and primary idea about how Flutter works and builds a mobile application.

We are keeping these two projects separate. For Dart we have a console sample application named 'beginning_flutter_with_dart' and for the Flutter project we have 'my_first_flutter_app'.

We have saved these two separate projects in two separate folders.

When we have created the Dart project, it comes up with two '.dart' files. The 'main.dart' is in the 'bin' folder, and the 'beginning_flutter_with_dart.dart' file is in the 'lib' folder.

Like 'C', 'C++' or 'Java', Dart application runs through the 'main()' function. Therefore, the 'main.dart' file in the 'bin' folder is the main file through which our Dart console sample application will run.

What is the role of the 'lib' folder? We will place other dart files in the 'lib' folder, just like the 'beginning_flutter_with_dart.dart' file.

When we open up the 'beginning_flutter_with_dart.dart' file, we find a function inside.

```
// code 1.9
int calculate() {
return 6 * 7;
}
```

And the 'main.dart' file has this code inside:

```
// code 1.10
import 'package:beginning_flutter_with_dart/beginning_flutter_with_dart.dart'
as beginning_flutter_with_dart;

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

If we run this code we get this output:

```
//output of code 1.10
Hello world: 42!
```

If you are an absolute beginner, it really makes no sense. Let us tolerate this for a moment and try to understand what is actually happening.

As a beginner, try to understand programming language from the standpoint of natural language. In a natural language, we start with letters or alphabets. Then we form words by arranging those alphabets. After that we need to learn grammar, which is a set of rules that teaches us to make

meaningful sentences. Only after learning to create sentences, we can think of writing a paragraph, an essay, a story, even a novel.

With the help of a programming language we also try to write an application, a software. A natural language works with words, and a programming language deals with data. This Dart console application gives us two types of data. One is 'String' data that gives the output: Hello World; and, it also gives us an 'integer' data, which gives us an output of 42.

Watch the code 1.9 carefully, it is a function of integer data type, and it has a name 'calculate()', and it returns an integer value of multiplication between two numbers 6 and 7. Quite predictably it has returned 42.

Therefore, we have learned one important concept, a function returns a value. If we mention what type of data type it will return, it will return that data type and in the main() function, we can call this function and get the desired output.

However, in the main() function, there are many more things that we should also discuss. If we take a look at the code 1.10, we see that in the 'main.dart' file we have imported the 'beginning_flutter_-with_dart.dart' file from the 'lib' folder as a package. When we have created the Dart application, it automatically gives it a name, the same name that we used while creating the console sample application.

Now, inside the print() function, that usually prints an output, Dart uses that name (beginning_-flutter_with_dart) and adds a '.' symbol to call the 'calculate()' function. Why this is happening? It is because Dart is an object-oriented programming language, and in Dart treats everything as an object. Through that object Dart calls any function as it has called here.

Now let us change the code 1.10 to this:

```
// code 1.11
import 'package:beginning_flutter_with_dart/beginning_flutter_with_dart.dart'
as an_object;

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

Simultaneously, we have changed the code 1.9 to this, where we have changed the inside value of calculate() function.

```
// code 1.12
int calculate() {
return 6 * 12;
}
```

If we run this program, it works and gives us this output:

```
//output of code 1.11
Hello world: 72!
```

We have learned a few important concepts. Dart converts everything into an object. We can call that object by any name. At the time of creation, the object was named 'beginning_flutter_with_dart'; later we have changed the name to a more generic name, such as 'an_object'. After changing the name, we have called the same function that was written into the Dart file, inside the 'lib' folder.

Can we create another function in the 'lib' folder?

Let us try. Creating a new function is very simple process. We will click the second mouse on the 'lib' folder in our IntelliJ IDE, it will automatically ask for creating different types of file. We have chosen a Dart file and named it 'a_new_function'. A new Dart file is generated inside the 'lib' folder.

We are trying to add two numbers through that function and returns its value. The code snippet looks like this:

```
// code 1.13
int addingTwoNumbers(var x, var y){
return x + y;
}
```

Now we will call this function inside the main() function just like before.

```
// code 1.14
import 'package:beginning_flutter_with_dart/beginning_flutter_with_dart.dart'
as an_object;
import 'package:beginning_flutter_with_dart/a_new_function.dart'
as a_new_function;

main(List<String> arguments) {
print('Hello world: ${an_object.calculate()}!');
print('Adding 10 and 20: ${a_new_function.addingTwoNumbers(10, 20)}');
}
```

The output is quite predictable.

```
//output of code 1.14
Hello world: 72!
Adding 10 and 20: 30
```

We have passed two variables 'x' and 'y' as parameters through the function addingTwoNumbers(var x, var y); instead of using the word 'var' we could have written 'int'. Dart is strongly typed programming language, so we can mention what data type we are passing. Otherwise, we can use

only ‘var’, that stands for variable, and Dart will automatically infer it as integers; in this case we wanted to return an integer data type.

If you look at the meaning of the word in natural language,the word ‘function’ is used as noun as well as verb. When you use it as noun, one of the meanings tells us something like this: the actions and activities assigned to or required or expected of a person or group. And as a verb, its meaning is quite straight forward: perform as expected when applied.

In the programming paradigm, a function() does not always return something. It could be void. That means it does not return anything. Take a look at the main() function. Before the main() function, have seen any data type like ‘int’ or ‘String’? The main() function always calls other functions and gives us the output.

Can we create a void function and call it inside a main function?

Yes, we can do. Let us add a void function inside the ‘a_new_function.dart’ file and the code 1.13 looks like this:

```
// code 1.15
int addingTwoNumbers(var x, var y){
return x + y;
}

void doNothing(){
print('Do nothing');
}
```

Now we can call this void function just like any other regular function, inside the man() function.

```
// code 1.16
import 'package:beginning_flutter_with_dart/beginning_flutter_with_dart.dart'
as an_object;
import 'package:beginning_flutter_with_dart/a_new_function.dart'
as a_new_function;

main(List<String> arguments) {
print('Hello world: ${an_object.calculate()}!');
print('Adding 10 and 20: ${a_new_function.addingTwoNumbers(10, 20)}');
a_new_function.doNothing();
}

//output
Hello world: 72!
Adding 10 and 20: 30
Do nothing
```

Since inside the void function we have used a print() function and passed a String object - 'Do nothing'. We get the same output.

We may ask, what is the function of this functions inside the Flutter project? Do this functions will have to do anything with our first mobile application?

To get that answer, we need to close our Dart project for a time being and move to the Flutter project 'my_first_flutter_app'. Let us open the Android Studio.

When the Flutter project was created it came with a main() file, just like we have just seen in the Dart project.

The code snippet is quite long, but we want to see the whole code here, because we want to see if we can find something familiar as an absolute beginner. So far, we have learned to create functions, and we have heard that everything in Dart is object, but we still do not understand it very much.

Let us see the whole code snippets, without the comments, first.

```
// code 1.17
import 'package:flutter/material.dart';

void main() {
runApp(MyApp());
}

class MyApp extends StatelessWidget {
// This widget is the root of your application.
@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: 'Flutter Demo',
    theme: ThemeData(

        primarySwatch: Colors.blue,

        visualDensity: VisualDensity.adaptivePlatformDensity,
    ),
    home: MyHomePage(title: 'Flutter Demo Home Page'),
    );
}
}

class MyHomePage extends StatefulWidget {
MyHomePage({Key key, this.title}) : super(key: key);

final String title;
```

```
29
30  @override
31  _MyHomePageState createState() => _MyHomePageState();
32  }
33
34  class _MyHomePageState extends State<MyHomePage> {
35  int _counter = 0;
36
37  void _incrementCounter() {
38      setState(() {
39
40      _counter++;
41      });
42  }
43
44  @override
45  Widget build(BuildContext context) {
46
47      return Scaffold(
48      appBar: AppBar(
49
50          title: Text(widget.title),
51      ),
52      body: Center(
53
54          child: Column(
55
56          mainAxisAlignment: MainAxisAlignment.center,
57          children: <Widget>[
58              Text(
59              'You have pushed the button this many times:',
60              ),
61              Text(
62              '$_counter',
63              style: Theme.of(context).textTheme.headline4,
64              ),
65          ],
66          ),
67      ),
68      floatingActionButton: FloatingActionButton(
69          onPressed: _incrementCounter,
70          tooltip: 'Increment',
71          child: Icon(Icons.add),
```

```
72      ),
73      );
74  }
75  }
```

Watching the above code, we can say that, yes, we have found one familiar thing, a main() function. And the main()function here directly calls a function runApp(); inside that runApp() function, Flutter has passed a parameter MyApp(), which also looks like a function. But it is not a regular function. It is an object that has been instantiated from the class MyApp().

Because of this code our virtual device looks like the figure 1.6.

We will remove all the code and build our mobile application from the scratch so that we can follow the building process step by step.

But before that, we will try to understand the code. We have already found one familiar function main() inside the Flutter 'main.dart' file. The second most important thing we have noticed a comment that tells us that '// This widget is the root of your application'.

Basically, in Flutter, Widget plays the key role. We can summarize that it is all about Widget. Our mobile application is a collection of many parent and child Widgets. The Widget tree contains different child Widgets, they draw the image on the mobile screen pixel by pixel.

At the top there is a header section, after that just below of the header starts the body part. Again, the body part contains many other Widgets. Things go on like this. In Flutter you cannot drag and drop your Widgets; we have to code them in different folders and after that we can import them as packages.

At the top of the Flutter 'main.dart' file we have seen this line:

```
1  import 'package:flutter/material.dart';
```

This 'material.dart' file has been supplied by Flutter. This file has many core functionalities that we can call in the 'main.dart' file.

We have also noticed a line like this:

```
1  class MyApp extends StatelessWidget {}
```

As an absolute beginner, we have learned function(), but we have not found out what class is. Moreover, we also do not know how 'class' and 'object' are connected; yet it is said that in Dart everything is object. Therefore, we will again, go back to our Dart console sample application; we will learn this core concepts and after that we will again come back to the Flutter project and build our mobile application from the scratch.

However, we feel that we need to understand the concepts of function in detail, especially in the context of object-oriented programming paradigm.

Functions and Objects

When we say: functions are objects in Dart, it seems confusing to the absolute beginners. The seasoned programmers may get the hint: Dart is an out and out object-oriented language. So even functions are objects and have a type called – Function.

It means many things. One of the key things is you can assign a function to a variable, and even you can pass a function as arguments to other functions. We have seen it in the Flutter 'main.dart' file as the following:

```
void main() {
runApp(MyApp());
}
```

To understand objects, you need to have an introduction to object-oriented programming. In this section, we will have an introduction to object-oriented programming. Otherwise, we cannot follow how Flutter works with its objects.

We have already seen how functions work. Although that was a basic introduction; we will cover this topic later in detail when we will discuss 'methods' in object-oriented programming.

Before writing a function, we need to remember a few major points:

```
1. It is a good practice to define a type of function. So type annotation is recomme\
nded.
2. Although Dart recommends type annotation, a function still works without any "typ\
e declaration". So you can omit the type and write it straight.
3. However, the most important thing to remember in Dart is: whatever value you want\
 to 'return', from a function, you need to change the 'type' of that function accord\
ingly. If you want an 'integer' value to 'return', you should change the 'type' of t\
he function to 'integer'.
4. For 'void', nothing is returned from a function. So whenever you use the keyword \
'void' before the function, you need to use the 'print(object)' option.
```

So far we have seen integer and String data types that return numbers and texts respectively. However, there is another major data type in every programming language, Dart is no exception. It is called 'boolean'. It returns either 'true' or 'false'. In our Flutter project, we will need this data type, especially while building the app logic.

Let us see some examples, it will give us a clear picture of how boolean data work. In our Dart project, in the 'lib' folder, we will create a new file 'boolean_function.dart'. We have written this following code inside that file:

```
// code 1.18
bool isTrue(){
return true;
}
bool isFalse(){
return false;
}
```

We should also change the ‘main.dart’ file accordingly.

```
// code 1.19
import 'package:beginning_flutter_with_dart/boolean_function.dart'
as boolean_object;

dynamic main(List<String> arguments) {

print('It is true: ${boolean_object.isTrue()}');
print('It is false: ${boolean_object.isFalse()}');

}
//output

It is true: true
It is true: false
```

If we did not mention the boolean type in the code 1.18, it would still work. But Dart strongly recommends to define the type. It makes your code more clear. By the way,we should know about a few other good practices.

According to the naming convention, it is recommended that a function name should always be like this: ‘aFunction()’; it is called camel case. The initial word will start with lower case, and the next word should start with an upper case. If we want to mean a single verb action, we can use one word like ‘build()’, which Flutter has done in its ‘main.dart’ file.

In the Flutter code, it means, the Widget control is building the first application.

When we name a class, we will use Pascal case, such as ‘MyApp’ used in the Flutter ‘main.dart’ file; in this case,both words start with upper case.

The names of function and class should always be meaningful and should be synchronized with your application. If we take a close look at the Flutter ‘main.dart’ file, we will understand that every name of class, function is meaningful.

To understand this naming convention, we should take a look at the Flutter ‘main.dart’ code again. Watch this code snippets from code 1.17:

```
class MyApp extends StatelessWidget {
// This widget is the root of your application.
@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: 'Flutter Demo',
    theme: ThemeData(

        primarySwatch: Colors.blue,

        visualDensity: VisualDensity.adaptivePlatformDensity,
    ),
    home: MyHomePage(title: 'Flutter Demo Home Page'),
    );
}
}
```

We can clearly see that the MaterialApp() calls another function ThemeData() inside it. Therefore, we can call another function inside a function.

We will go back to the Flutter project again, but before that, let us open our IntelliJ IDE and try to pass the 'ThemeData()' function inside the function 'MaterialApp() '. To do that we need to create 'passing_a_function_inside_function.dart'. Inside that file, we write a simple function that passes a function as its parameter.

```
// code 1.20
String MaterialApp(page()){
return page();
}
```

Now in the 'main.dart' file we call that 'MaterialApp(page())' function and try to pass another function as its parameter.

```
// code 1.21
import 'package:beginning_flutter_with_dart/passing_a_function_inside_function.dart'
as passing_function;

// ignore: always_declare_return_types
main(List<String> arguments) {

// ignore: always_declare_return_types
themeData(){
    return 'Home Page';
```

```
}

print('Passing function inside a function: ${passing_function.MaterialApp(themeData)\
}');

}

//output
Passing function inside a function: Home Page
```

We have just done what we have seen in the Flutter 'main.dart' file. Of course, we have done in a microscopic form. Flutter uses the same concepts in a much bigger way. But we have at least started understanding what is happening inside. Yet, we need to understand class and objects. When we use a function inside a class, it is usually called a method.

A class may extend functionalities of other classes. Flutter tool uses hundreds of such classes, and extend many more classes to use their functionalities and give the application a concrete shape.

Before going to understand classes and objects, let us go back to our Flutter project again. This time, we will remove the in-built code and after that, we will try to write a small piece of code, to change the appearance of the virtual device.

Building the mobile application from scratch

Let us open the Android Studio and start our virtual device. To start building our first mobile application, we need to remove all the in-built code from the 'main.dart' file.

Let us name our application "MyFirstApp". The very first thing we need to do is, write a main() function inside the 'main.dart' file. Any Flutter project will always launch through the main() function.

We are not going to do anything special. Displaying a text,such as 'My First Flutter app from scratch...' will be enough at the beginning. Let us write our code, the following way:

```
// code 1.22
import 'package:flutter/material.dart';

void main() {
runApp(MyFirstApp());
}

class MyFirstApp extends StatelessWidget {
Widget build(BuildContext context) {
    return MaterialApp(home: Text('My First Flutter app from scratch...'),
```

```
11      );
12  }
13  }
```

Before running our code we will take a look at the virtual device that comes up with the creation of the Flutter project. Think about the code 1.17, which was created on the original 'main.dart' file. It created a virtual device,which displays a counter. Clicking the button raise the counter number by 1.

The virtual device initially looked like the following image (Figure 1.8).

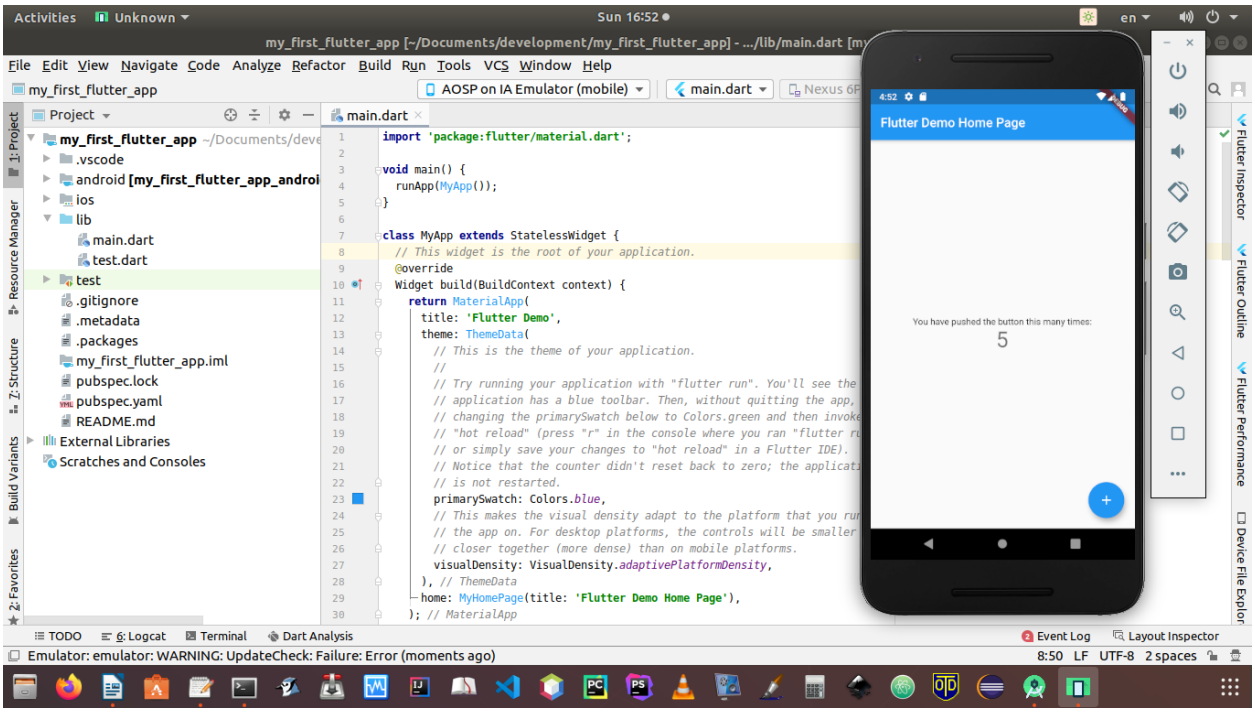


Figure 1.8 – The virtual device at the time of creation of Flutter App

Now we have changed the original 'main.dart' file and building our first flutter application from scratch. Let us run the new 'main.dart' file, and we get this output in our virtual device (Figure 1.9). It is not a good looking application, at present, but to add more functionalities we need to understand more core concepts, such as how class and object work in Dart.

We have kept the main() file as it was. Only we have changed our application's name from 'MyApp' to 'MyFirstApp'.

```
1  void main() {
2  runApp(MyFirstApp());
3  }
```

We have also changed the original class configuration. Now 'MyFirstApp(){}' class extends another class 'StatelessWidget(){}', which also passes the Widget method build(). The Widget build() method

also passes another 'BuildContext' object called 'context' as its parameter. This build() method returns another class constructor 'MaterialApp()' that returns a 'named parameter' Text() class constructor. Inside the Text() class constructor we have passed a String parameter 'My First Flutter app from scratch...'.

As an absolute beginner, it appears to be very difficult if you do not know anything about class and objects. However, our first code works and change the appearance of the virtual device.

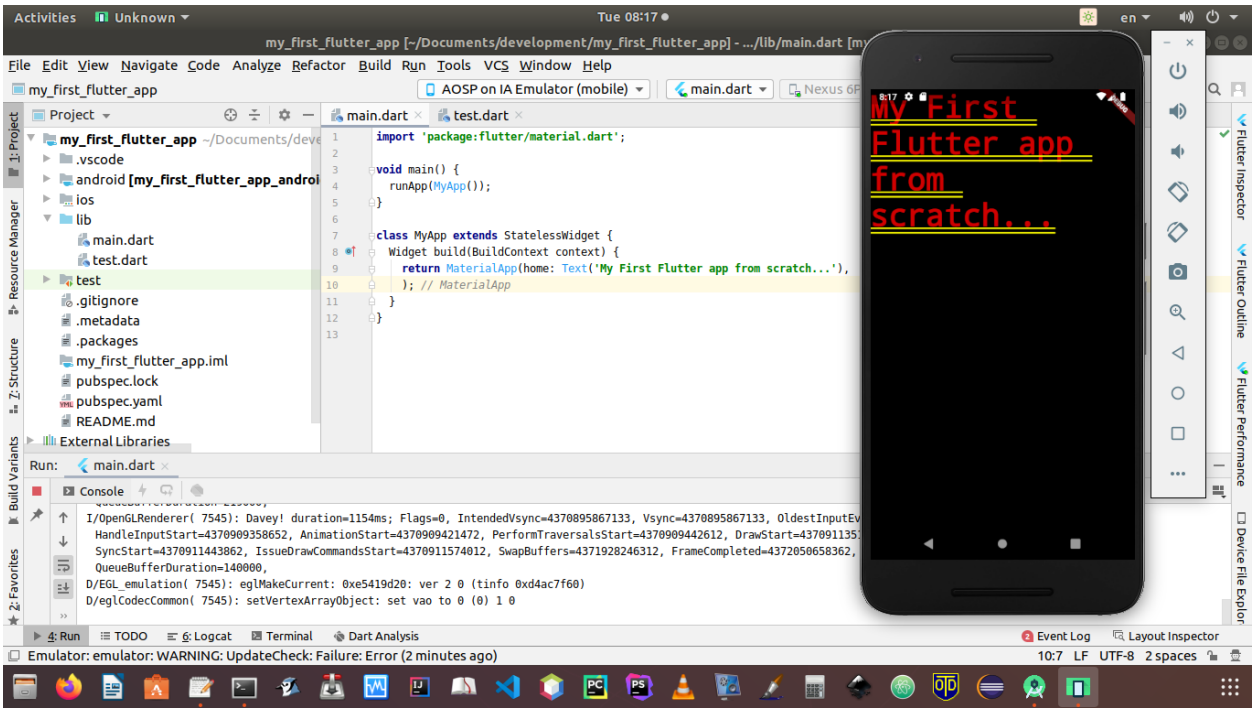


Figure 1.9 – Building our first mobile application from scratch

It does not look good. But we have just started building the application. It will look better as the application progresses. However, before progressing any further with our first Flutter application, we have to understand how class and object work. To understand that we need to go back to our Dart project again.

2. Flutter and Dart Architecture: Understanding Class and Object

So far, we have understood one key concept. Flutter has many in-built classes and objects that interacting with each other and build application. This is a complex process and it runs on Dart platform.

We have also seen that in any Dart code, 'package' or libraries play the key role. In a Flutter project, everything runs through the 'main.dart' file, or through the main() function.

Now it is impossible to write hundreds of classes and creating thousands of objects inside one 'main.dart' file. The key concept of object-oriented programming is modularity. Breaking the application into several different parts will enhance the ability to organize our code.

Therefore, we need to understand the key concepts of class and objects.

A Short Introduction to Class and Objects

As we know, Dart is an object-oriented language with classes and objects. Every object is an instance of a class and all classes descend from Object.

For absolute beginners, it is a conflicting statement. Class is behind every object. And it says, behind classes there are objects. What does it actually mean? It means a class could have many classes inside. It can inherit properties and methods of other classes, when that class has an instance or object, we can say that the object is behind many classes.

In Dart, there is a concept called 'Mixin-based' inheritance. It means every class has exactly one superclass, and a class body can be reused in multiple hierarchies. This concept is a little bit advanced for absolute beginners. So we will cover 'Mixin-based' inheritance at the end of this book.

To begin with, we start with a simple class and an object. So far we have seen variables and functions. We have seen how we can pass variables as parameters.

Let us think about something that will hold variables and functions inside. We call it a class. You should think this way: an object has states, like a person object has 'colorOfEyes'. It is a state. A state can always be changed. Therefore, an object can also change its state. So, we can conclude that an object has some states and it can also change its states. The way or method that are used in changing object-states are called 'methods'. You can also call it 'function'. Many languages use the keyword function, and some other use the term method. Extending our person object, which has state like 'colorOfEyes', we can say that the person object has one method called 'wearContactLens()'. Now, using that method, the person object can change its state 'colorOfEyes' .

A class always acts like a blueprint of an object. What will be the states, and how those states can be changed, all are predefined in a class.

In the next section we will see a long list of code, which will create mainly two objects; a person from a Person class, and a robot object from a Robot class. If you do not understand it at the first glance, do not worry. It takes time.

How two objects interact

Before starting to learn how two objects interact, we must remember that Dart is an object-oriented programming language. It means, in Dart, everything is an object.

Behind every object, there is a blueprint or class that decides how an object will change its states using different methods.

We will discuss about objects in more detail, as we progress. For the sake of simplicity, you only know that we are all objects. Actually, any object-oriented language simulates the real world.

Here, in the coming application mobile game, we assume that one person object owns one robot object. The person object and the robot object have names, and they can do some actions. An object ideally should have a state and it can also change its state by using methods.

Consider a simple example. Besides having names, the robot object has state like ‘number of bullets’ that it can fire at something. We can change the state of the number of bullets by using a method called ‘canFire(number of bullets)’, passing the number of bullets as its parameter.

When a person owns a robot, he or she can use it to fire at some other person objects. And the person object that has been fired at, can strike back to the person object who fires at him.

By now, you understand that we are talking about four different objects. They are interacting with one another. In any application, be it a software, or a mobile application, or a web application, these interactions between various objects keep going on.

Our, programmer’s job is to design the application most efficiently.

Okay, enough talking, let us watch some code now. Here is the first code snippet, where we have written two classes, Person and Robot. We have also created two objects, belonging to each of them.

In the bottom section, we have commented out some actions that we are going to write in the next code snippet.

```
// code 2.1
class Person{
String name;
Person(this.name);
}

class Robot{
String name;
Robot(this.name);
}

// ignore: always_declare_return_types
main(List<String> arguments) {
var personOne = Person('John');
var robotOne = Robot('ROBO_COP');
print('The first person object has a name : ${personOne.name}');
print('The first robot object has a name : ${robotOne.name}');

// robot.canFire(any number of bullets)
// person.getsARobot(robotOne)
// personOne.robotOne.fireAt(personTwo)
// personTwo.getsARobot(robotTwo)
// personTwo.robotOne.fireAt(personOne)
}

//output
The first person object has a name : John
The first robot object has a name : ROBO_COP
```

So, we have successfully created two objects, belonging to Person and Robot. Now the person must have this robot, which is called ROBO_COP.

A Person class is written like this:

```
class Person{
String name;
Person(this.name);
}
```

Inside the Person class, we have two things – one is a state (name), which is represented by a data type String. Therefore, it will be text. Next we have a special method called constructor. The name of the constructor should always be name of the class. Here the name of constructor is Person(). There could be many constructors, we will see to that later.

In this case, we have one constructor that passes a parameter 'this.name'. It means, through that special method or function called constructor we can change the state of the 'person' object. Now we are able to create different person objects with different names. A person can now own a Robot object. We have followed the same procedure for the Robot class.

By the way, the person constructor can also be written like this also.

```
class Person{
String name;

Person(String name){
    this.name = name;
}
}
```

We have commented out the parts of our code in the below, which we are going to implement in the future course of our program.

After owning this robot he can do some actions using that. Therefore, in our next code snippet we have added a few lines.

```
// code 2.2
class Person{
String name;

Person(String name){
    this.name = name;
}
}

class Robot{
String name;
int numberOfBullets;

Robot(String name){
    this.name = name;
}

int canFire(int numberOfBullets){
    this.numberOfBullets = numberOfBullets;
    return numberOfBullets;
}


```

```
24
25  }
26
27  void main(){
28  var personOne = Person("John");
29  var robotOne = Robot("ROBO_COP");
30  print("The first person object has a name : ${personOne.name}");
31  print("The first robot object has a name : ${robotOne.name}");
32
33  // robot.canFire(number of bullets)
34  // person.getsARobot(robotOne)
35  // personOne.robotOne.fireAt(personTwo)
36  // personTwo.getsARobot(robotTwo)
37  // personTwo.robotOne.fireAt(personOne)
38
39  print("${robotOne.name} can fire ${robotOne.canFire(100)} bullets");
40  }
41
42  // output:
43
44  The first person object has a name : John
45  The first robot object has a name : ROBO_COP
46  ROBO_COP can fire 100 bullets
```

So, every robot can fire and fire up to 100 bullets. Now, this person object John can use this robot to fire at someone else. To make that happen, we need to create another Person object and Robot object, but before that John must own this ROBO_COP.

In the next code snippet we will try that, first.

```
1  // code 2.3
2  class Person{
3  String name;
4  Robot robot;
5
6  Person(String name){
7      this.name = name;
8  }
9
10 String getsARobot(Robot robot){
11     this.robot = robot;
12     return robot.name;
13 }
```

```
}

class Robot{
String name;
int numberOfBullets;

Robot(String name){
    this.name = name;
}

int canFire(int numberOfBullets){
    this.numberOfBullets = numberOfBullets;
    return numberOfBullets;
}



}

void main(){
var personOne = Person("John");
var robotOne = Robot("ROBO_COP");
print("The first person object has a name : ${personOne.name}");
print("The first robot object has a name : ${robotOne.name}");

// robot.canFire(number of bullets)
// person.getsARobot(robotOne)
// personOne.robotOne.fireAt(personTwo)
// personTwo.getsARobot(robotTwo)
// personTwo.robotOne.fireAt(personOne)

print("${robotOne.name} can fire ${robotOne.canFire(100)} bullets");
print("${personOne.name} has a robot called ${personOne.getsARobot(robotOne)}");
}
```

Let us watch the output to check whether this person John has owned this robot object or not.

```
The first person object has a name : John
The first robot object has a name : ROBO_COP
ROBO_COP can fire 100 bullets
John has a robot called ROBO_COP
```

Therefore, John has owned ROBO_COP. Now, he can use this robot to fire at someone. Right? But,

to do that we need some more person and robot objects.

The next code snippet shows us the same thing.

```
// code 2.4
class Person{
String name;
Robot robot;

Person(String name){
    this.name = name;
}

String getsARobot(Robot robot){
    this.robot = robot;
    return robot.name;
}
}

class Robot{
String name;
int numberOfBullets;
Person person;

Robot(String name){
    this.name = name;
}

int canFire(int numberOfBullets){
    this.numberOfBullets = numberOfBullets;
    return numberOfBullets;
}

String fireAt(Person person){
    this.person = person;
    return person.name;
}


}

void main(){
var personOne = Person("John");
```

```
var robotOne = Robot("ROBO_COP");
print("The first person object has a name : ${personOne.name}");
print("The first robot object has a name : ${robotOne.name}");
var personTwo = Person("Hicky");
print("The second person object has a name : ${personTwo.name}");

// personOne.robotOne.fireAt(personTwo)
// personTwo.getsARobot(robotTwo)
// personTwo.robotOne.fireAt(personOne)

print("${robotOne.name} can fire ${robotOne.canFire(100)} bullets");
print("${personOne.name} has a robot called ${personOne.getsARobot(robotOne)}");
print("${personOne.name} uses ${personOne.getsARobot(robotOne)} "
    "to fire at ${robotOne.fireAt(personTwo)}");

}
```

Once a new person object Hicky comes into picture, John uses his robot to fire at the new person, Hicky. Here is the output:

```
//output
The first person object has a name : John
The first robot object has a name : ROBO_COP
The second person object has a name : Hicky
ROBO_COP can fire 100 bullets
John has a robot called ROBO_COP
John uses ROBO_COP to fire at Hicky
```

We can also manipulate the number of bullets in the run-time. Let us change the last line of the above code snippet.

```
print("${personOne.name} uses ${personOne.getsARobot(robotOne)} "
    "to fire ${personOne.robot.canFire(50)} bullets at ${robotOne.fireAt(personTwo)}\
");
```

It changes the output, if we re-run the code.

```
//output
The first person object has a name : John
The first robot object has a name : ROBO_COP
The second person object has a name : Hicky
ROBO_COP can fire 100 bullets
John has a robot called ROBO_COP
John uses ROBO_COP to fire 50 bullets at Hicky
```

Now, Hicky should be able to retaliate. Otherwise, how we can make our futuristic mobile application look interesting? Therefore, in the next code snippet, we have managed to solve that problem.

```
// code 2.5
class Person{
String name;
Robot robot;
Person person;

Person(String name){
    this.name = name;
}

String getsARobot(Robot robot){
    this.robot = robot;
    return robot.name;
}

String strikeBack(Person person){
    this.person = person;
    return person.name;
}
}

class Robot{
String name;
int numberOfBullets;
Person person;

Robot(String name){
    this.name = name;
}

int canFire(int numberOfBullets){
    this.numberOfBullets = numberOfBullets;
```

```
33      return numberOfBullets;
34  }
35
36  String fireAt(Person person){
37      this.person = person;
38      return person.name;
39  }
40
41
42  }
43
44  void main(){
45  var personOne = Person("John");
46  var robotOne = Robot("ROBO_COP");
47  print("The first person object has a name : ${personOne.name}");
48  print("The first robot object has a name : ${robotOne.name}");
49  var personTwo = Person("Hicky");
50  print("The second person object has a name : ${personTwo.name}");
51  var robotTwo = Robot("ROBO_MACHINE");
52  print("The second robot object has a name : ${robotTwo.name}");
53
54  print("${robotOne.name} can fire ${robotOne.canFire(100)} bullets");
55  print("${personOne.name} has a robot called ${personOne.getsARobot(robotOne)}");
56  print("${personOne.name} uses ${personOne.getsARobot(robotOne)} "
57      "to fire ${personOne.robot.canFire(50)} bullets at ${robotOne.fireAt(personTwo)}\
58  ");
59  print("${personTwo.name} has a robot called ${personTwo.getsARobot(robotTwo)}");
60  print("${personTwo.name} uses ${personTwo.getsARobot(robotTwo)} "
61      "to fire ${personTwo.robot.canFire(100)} bullets at ${robotTwo.fireAt(personOne)}\
62  }");
63  print("${personTwo.name} strikes back at ${personTwo.strikeBack(personOne)}");
64
65  }
```

Look, in the above code snippet, we don't have any commented out sections anymore, because we have implemented everything that we have wanted to do. Therefore, the output changes as follows:

```
//output
The first person object has a name : John
The first robot object has a name : ROBO_COP
The second person object has a name : Hicky
The second robot object has a name : ROBO_MACHINE
ROBO_COP can fire 100 bullets
John has a robot called ROBO_COP
John uses ROBO_COP to fire 50 bullets at Hicky
Hicky has a robot called ROBO_MACHINE
Hicky uses ROBO_MACHINE to fire 100 bullets at John
Hicky strikes back at John
```

So, we can make this battle more interesting; moreover, we can add more features. However, to do that, we need to design the software, data structures, and algorithm in the most efficient manner.

All we have done is, we have created a few objects, building relationship between them by passing objects through various classes. After that they start interacting with each other. Of course, we can do this job more efficiently. But, to do that, we need to learn the language basics, first. Only after learning that, we can design our first Flutter application more efficiently.

More about classes and objects

At the very beginning, we have seen Person and Robot class. We have also seen how we have created various objects that interacted with each other. Now, in every programming language, it is a customary that when we explain class and object, we give examples of Car class. Some also use Dog and Cat class. Thinking that animal right could be violated, we have restricted ourselves to Car class.

Suppose we have a car class. It has two properties: name, and model number. It has also a method (outside object-oriented paradigm we call it function) called ‘isTurnedOn(bool)’ we have passed a ‘boolean type’ argument through that function or method. Consider it as the “action part” of the class ‘Car’. When we pass ‘boolean value true’, the car starts and when we pass ‘boolean value false’, the car stops.

Now imagine a manufacturer company wants to build many cars that have separate names, model numbers but each one has one method ‘isTurnedOn(bool)’. In this scenario, each car is an object or instance of ‘Car’ class. Consider the code below.

```
//code 2.6
main(List<String> arguments) {
var newCar = new Car();
newCar.carName = "Red Angel";
newCar.carModel = 256;
if(newCar.isTurnedOn(true)){
    print("${newCar.carName} starts. It has model number ${newCar.carModel}");
} else print("${newCar.carName} stops. It has model number ${newCar.carModel}");
}
class Car {
int carModel = 123;
String carName = "Blue Angel";
bool isTurnedOn(bool){
    return false;
}
}
```

It gives us this output:

```
//output
Red Angel stops. It has model number 256
```

Watch the 'Car' class. It has two properties or attributes (or states): 'carName' and 'carModel'. Treat them as variables, but since they are inside a class, we will call them properties, members, or attributes, or states. These values can be changed when we will create an instance. In fact, we have done the same, inside the 'main()' function.

The default values were '123' and 'Blue Angel'. But we have an output where the name changes to 'Red Angel'. And the model has been changed to '256'. We have created an instance or object of the 'Car' class, by simply writing this line:

```
var newCar = new Car();
```

Next , we have defined the name and the model number as:

```
newCar.carName = "Red Angel";
newCar.carModel = 256;
```

The next step is vital, because we have declared the method 'isTurnedOn(bool)' as 'true'.

```
if(newCar.isTurnedOn(true)){
print("${newCar.carName} starts. It has model number ${newCar.carModel}");
} else print("${newCar.carName} stops. It has model number ${newCar.carModel}");
```

Now according to our logic, if the method 'isTurnedOn(bbol)' is 'true', it should start. But in the output, we have seen that it 'stops'.

Why it happens?

It happens because in our 'Car' class, we have already set that value 'false.'

Let us change it to 'true' and see the output again:

```
//code 2.7
main(List<String> arguments) {
var newCar = new Car();
newCar.carName = "Red Angel";
newCar.carModel = 256;
if(newCar.isTurnedOn(true)){
    print("${newCar.carName} starts. It has model number ${newCar.carModel}");
} else print("${newCar.carName} stops. It has model number ${newCar.carModel}");
}
class Car {
int carModel = 123;
String carName = "Blue Angel";
bool isTurnedOn(bool){
    return true;
}
}

//Watch the output again:

//output
Red Angel starts. It has model number 256
```

From this example, we can conclude one thing: a class is a blueprint of an object. An object or an instance of a class is extremely powerful, it is not like simple variables, holding one reference to a spot in the memory where we can only store a value. Through an 'app' object we can run a large complicated application, moreover, we can make a series of complex layers of logic behind an object.

How Flutter and Dart work together

Now we have an introduction to class and object-oriented programming paradigms. Now in the light of new insights we have just learned, we can define Flutter more precisely. Flutter is a tool

that builds native cross-platform (here cross-platform means for iOS and Android platforms, both) mobile applications with one programming language Dart and one code-base.

Flutter has its own SDK or Software Development Kit that converts our code to native machine code. SDK also helps us to develop our application more eloquently.

Due to the presence of the SDK Flutter works as a framework or Widget library that produces reusable User Interface or UI that builds different types of blocks, utility functions and packages.

Although Dart is an object-oriented programming language, for Flutter its role is more focused on front-end specific. It means with the help of Dart we will build different types of User Interfaces.

As we progress we will understand this core concepts more and more while building our first Flutter application. Understanding Dart language basic is important as it helps Flutter to build UI as code. Since Flutter is mainly different types of Widget trees, we have to write different types of code in Dart.

The advantage of Flutter is that it provides iOS specific code, as well as Android specific code from a single code-base. When we run a Flutter application on our smartphone, we actually see a bunch of Widgets. From the top to the bottom of the mobile screen Flutter divides its Widgets accordingly.

We may think these Widgets as controls that we create by writing code in Dart. In fact, we do not have to do the low level plumbing each time. Flutter framework and Widget libraries provide us the required assistance.

All we need to do is to memorize common rules of building basic Widgets, and moreover, we need to understand the core Dart language basic like function, class and object-oriented style of programming, different types of parameters and their roles in Flutter Widgets, etc.

As we have said, any Flutter application is a bunch of Widgets, we also mean that what we see on the mobile application is Widget trees. However complex application it appears to be, it is actually a bunch of Widget trees. Since there is no Visual Editor code assistance, there is no drag-and-drop facility. We have to code the entire application. Although it sounds daunting at the beginning, in reality, it is not. Because Flutter SDK has come up with almost every kind of solutions, we just need to add those functionalities.

We will find different types of buttons, text boxes, text decorations available. The Flutter API comes up with two distinct facilities; one is Utility functions, and the other is Widget libraries. They are written in Dart, and Dart complies every code we write with the help of SDK, and finally we get the native code for iOS and Android. The iOS platform has different types of buttons, so the Android. Flutter tackles this problem in its own way, it has a custom implementation. Every pixel is drawn on the mobile screen. For that reason, the platform specific limitations are tackled without any hitch.

If we think of a minimal Flutter app, it calls the runApp()function inside the main() function. We could have called the runApp() function with a Widget. Instead we passed a parameter MyFirstApp(), which is a class that extends ‘StatelessWidget’ class (code 1.22).

Now we are going to change the code 1.22 to get an idea of how Flutter can run minimally based only on runApp() function.

```dart
// code 2.8
import 'package:flutter/material.dart';

void main() {
runApp(
    Center(
    child: Text(
        'My First Flutter App is running!',
        textDirection: TextDirection.ltr,
    ),
    ),
);
}
```

We have run the Flutter default Center() class constructor inside the runApp() function. It automatically changes the look of the virtual device (Figure 2.1).

Of course, this is not the way one should build a Flutter app. We will also do not take this way. However, getting an idea of how a Flutter app runs will not hurt the learning process.

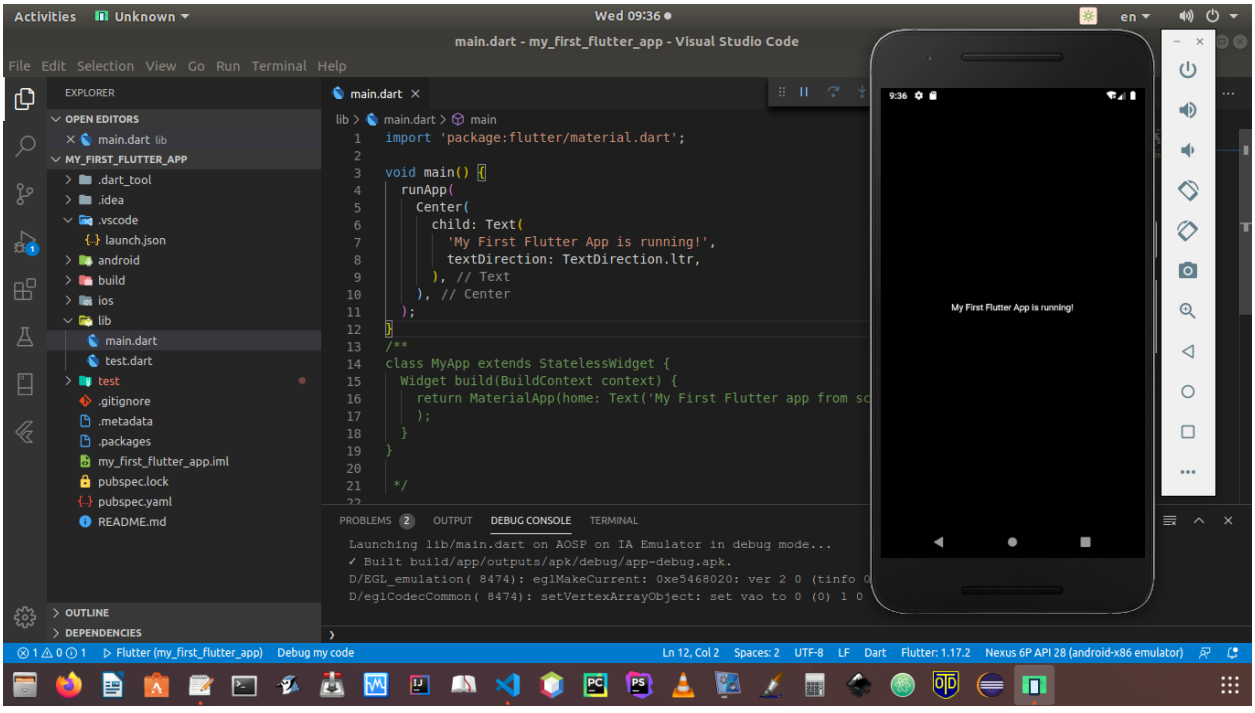


Figure 2.1 – A minimal version of Flutter App

The above image gives us another idea of writing our code, which is preferable especially for macOS and Linux users.

We can run the virtual device from Android Studio, and after that we can open the same Flutter

project using Visual Studio IDE. The virtual device will automatically synchronize with Visual Studio (VS); the advantage of using VS IDE is it gives you chance to 'hot reload' property. Each time we change the code, we will just click the 'hot reload' button that hangs loosely over the VS IDE. We can immediately see the change on the connected Virtual Device.

Now, we will get back to our old code snippet that gave us an ugly looking text output (Figure 1.9).

We will start building our code from the following code that we had seen in code snippets 1.22: import 'package:flutter/material.dart';

```
void main() {
runApp(MyFirstApp());
}

class MyFirstApp extends StatelessWidget {
Widget build(BuildContext context) {
    return MaterialApp(home: Text('My First Flutter app from scratch...'),
    );
}
}
```

Let us try to understand how our first Flutter Application 'MyFirstApp' gives the text output. First of all, it is a generic class that extends another Widget class StatelessWidget(). The 'material.dart' file has defined all these in-built classes.

Inside our 'MyFirstApp' class we have called a Widget class method build() that passes an object 'context' that is an instance of class 'BuildContext'.

As we have said earlier, in Dart, everything is Widget. For that reason, inside the build() method of Widget class we have returned another Widget 'MaterialApp()', which draws everything from the 'material.dart' file.

We need to study this part of code more closely to know a few key concepts of Dart language. A function sometimes passes positional argument or parameter; and, sometimes a function passes named argument. Look at this line of code:

```
Widget build(BuildContext context)
```

Here, the 'context' object is a positional argument. But the 'MaterialApp()' Widget passes named argument, like this:

```
return MaterialApp(home: Text('My First Flutter app from scratch...'),
```

We need to understand this key concept, first. After that we will again come back to our Flutter project and keeps on building our first platform independent mobile application using Flutter.

Positional and Named argument

Whether in a class method or in a function, sometimes you need to pass values. We call them arguments or parameters, whichever you like.

Dart is so flexible, it gives ample opportunity to the developers to manipulate the parameters. We may use the default parameters, in such cases, you need to pass the parameters. That is compulsory. But there are two other options available in Dart. You can use 'positional parameter' or 'named parameter'.

Let us see that in a code where we have used default and positional parameters:

```
1  //code 2.9
2  //default parameters
3  String defaultParameters(String name, String address, {int age = 10}){
4  return "$name and $address and age $age";
5  }
6  //optional parameters
7  String optionalParameters(String name, String address, [int age] ){
8  return "$name and $address and $age";
9  }
10 void main(){
11 print(defaultParameters("John", "Jericho"));
12 print(optionalParameters("John", "Form Chikago"));
13 // overriding the default age
14 print(defaultParameters("JOhn", "Jericho", age : 20));
15 }
```

Inside the main() function, in our default parameter function, we have passed only two values: name and address. We did not pass the 'age'. We did not have to, because it had already been defined in our function: {int age = 10}. Remember to use the curly brace to define the default parameter.

Can we override the default parameter? Yes, we can. See this part inside the main() function:

```
1  // overriding the default age
2  print(defaultParameters("JOhn", "Jericho", age : 20));
```

We have overridden the default age and made it from 10 to 20.

Next, in the optional parameter function, we have made the age optional by keeping the value inside the second bracket opened and closed.

```
//optional parameters
String optionalParameters(String name, String address, [int age] ){
return "$name and $address and $age";
}
```

Since the parameter 'age' is optional, we can either pass it or can ignore it. However, ignoring the optional parameter will return 'null'. So the output of the above code will be like this:

```
//output of code 3.22
John and Jericho and age 10
John and Form Chikago and null
JOhn and Jericho and age 20
```

In the case of the 'named parameter', we can swap the value and it has got a very high flexibility. Here sequence does not matter. Let us consider this code:

```
//code 2.10
//named parameter
int findTheVolume(int length, {int height, int breadth}){
return length * height * breadth;
}
void main(){
//sequence does not matter
var  result1 = findTheVolume(10, height: 20, breadth: 30);
var  result2 = findTheVolume(10, breadth: 30, height: 10);
print(result1);
print(result2);
}
```

In the above code, we have placed 'height and breadth' inside curly braces. So they are named parameters that we can interchange while passing the values. Interchanging the value will not affect our code. In the case of named parameters or arguments, position does not matter anymore. That is the advantage of named parameters.

Now, again we will go back to our Flutter project and watch the code 1.22, the very first code that we have written to run our first Flutter application. It is a positional argument:

```
Widget build(BuildContext context)
```

And this is a named argument:

```
return MaterialApp(home: Text('My First Flutter app from scratch...'),
```

Flutter SDK and Widget libraries have done all the heavy lifting for us. However, we should be aware of the very basic programming paradigms, otherwise, the Flutter code will not appear meaningful to us.

In the next chapter, we will try to learn a few basic rules regarding the Dart programming language. We should have a clear knowledge about how variables work, what are data types, etc. Moreover, we should know about the programming logic and basic idea about data structures and algorithm by learning the control flow.

After learning the language basic of Dart, we will again come back to our Flutter project. That will help us to understand the Flutter app logic.

Want to read more Flutter related Articles and resources? For more Flutter related Articles and Resources[3]

[3] https://zerodotone.net

3. Dart Language Basic and its implementation in Flutter

In this chapter, we will discuss some initial key concepts of Dart that are absolutely necessary for the beginners.

First of all, like C++, or Java, Dart is also an object-oriented programming language. Everything is an object here. It means a lot to the modern day programming paradigm. Every object in Dart has a class behind it. All the objects inherit from the "Object" class.

Consider a whole number like 2. In nature, it is a positive integer. In Dart all integers are objects. Even functions and null are also objects. I know, the term "object" may fill a beginner with bewilderment. We have already discussed object-oriented programming a little bit. Besides that, we should know what are variables, what are constants and what is function in more detail. Otherwise, we could not follow the Flutter App logic in coming chapters.

Like other programming languages, Dart has also several types, such as integers, strings, boolean, etc. Although Dart is strongly typed language, it also allows you to be duck typed.

What is that? In normal circumstances, in Dart, we mention what type we are going to use. If we use integers and strings, we write it like this:

```
1  //code 3.1
2  int myAge = 12;
3  String myName = "John Smith";
```

In the above examples, we have explicitly declared the type that would be inferred. In the next example we do the same thing, but implicitly.

Therefore, you can also write the same code this way:

```
1  //code 3.2
2  var myAge = 12;
3  var myName = "John Smith";
```

Variables Store References

Explicit or implicit, we have actually created two variables and initialized them with values. Variables store references to objects. In other words, you may say, a variable is a spot on the memory or

a container that contains some references to some values. Since the name is “variable”, the reference can change.

Now, the question is: with the change of reference, does the type also change? Please read on.

In the above code snippets, variable ‘myAge’ store a value 12 and reference it to an integer object. The same way, ‘myAge’ variable store a value ‘John Smith’ and reference it to a ‘String’ object. The type of the ‘myName’ variable is inferred to string-specific but you can change it. If you don’t want a specific or restricted type, specify ‘Object’ or ‘dynamic’ type.

```
dynamic myName = “John Smith”;
```

If you don’t initialize a variable, the default value is set to be ‘null’. Let us consider the following code: int myNumber;

Although it is an integer, it is not initialized. Therefore the default value is NULL. Let us run the code and watch the output.

```
//code 3.3
main() {
print("Hello World!");
int myNumber;
print(myNumber);
}
```

The output is as expected:

```
Hello World!
null
```

Before the discussion of ‘const’ and ‘final’ let us know what are the ‘Built-in Types’ in Dart. So far we have seen some of the types, such as number and string. We have not seen the others.

Built-in Types in Dart

The Dart language has special support for the following types and you can always follow the strongly typed or duck typed pattern to initialize them:

```
1. numbers
2. strings
3. boolean
4. lists (also known as arrays)
5. sets
6. maps
7. runes (for expressing Unicode characters in a string)
8. symbols
```

You can initialize an object of any of these special types using a literal. For example, 'Hello John Smith' is a string literal, and "false" is a boolean literal. Consider this code:

```
//code 3.4
main() {
String saySomething = "Hello John Smith";
var isFalse = true;
if(saySomething == null){
    print("It is ${isFalse}");
}else print("It is not ${isFalse}");
}
```

Since the string variable is not 'null', the output should be:

```
It is not true
```

Since the string variable is not 'null', the output came out as 'not true'.

We will encounter the first four built-in types most often. We will find the usages of other built-in types also as situation demands.

Suppose, you don't like Variables

Well, in some cases, you need the value to be constant. There are two ways that you can follow where you never intend to change the value of a variable. You may use 'const' instead of 'var' or 'String, int or bool' type declaration.

You may also use 'final'; but remember, 'final' variable can be set only once. So there is a difference between these two keywords: 'const' and 'final'. We will again come back to this topic when we will discuss object-oriented programming. Because instance variable can be 'final' but not 'const'; unless we start learning object-oriented programming we are in no position to discuss what instance variable is.

Consider this code:

```
//code 3.5
main() {
const firstName = "Sanjib";
final lastName = "Sinha";
String firstName = "John";
String lastName = "Sinha";
}
```

Watch the output full of errors:

```
//output
bin/main.dart:8:10: Error: 'firstName' is already declared in this scope.
String firstName = "John";
        ^^^^^^^^^
bin/main.dart:5:9: Context: Previous declaration of 'firstName'.
const firstName = "Sanjib";
        ^^^^^^^^^
bin/main.dart:9:10: Error: 'lastName' is already declared in this scope.
String lastName = "Sinha";
        ^^^^^^^^
bin/main.dart:6:9: Context: Previous declaration of 'lastName'.
final lastName = "Sinha";
```

When you want a variable to be compile-time constants, use ‘const’; and use ‘final’ for the instance variable that you will never change.

More about built-in types

In a quick review, we will first check the numbers. Then one after another we will learn about string, boolean, and other types.

Dart numbers are of two types: integers and decimals. We write them as ‘int’ and ‘double. Integers are numbers without decimal points. Examples: 1, 2, 22, etc. Doubles do have a decimal point like this: 1.5, 3.723, etc. Both ‘int’ and ‘double’ types are sub-types of ‘num’. The ‘num’ type includes basic Arithmatic operators such as “+, -, /, and *”; and they represent ‘plus, minus. division and multiplication’ signs. We can call them arithmetic operators and it also includes modulo, that is, remainder and the sign is: ‘%’.

Let us see some interesting examples:

```
//code 3.6
main() {
var one = int.parse('1');
print(one);
if(one.isOdd){
    print("It is an odd number.");
} else print("It is an even number.");
}
```

We have converted a string into an integer, or number.

```
//output
1
It is an odd number.
```

We can also turn a string to a double number. Let us change the above code a little bit:

```
//code 3.7
main() {
var one = int.parse('1');
var doubleToString = double.parse('23.564');
print(one);
print(doubleToString);
if(one.isOdd && doubleToString.isFinite){
    print("The first number is an odd number and the second one is a double ${double\
ToString} and a finite number.");
} else print("It is an even number and the second one is not a double ${doubleToStri\
ng} and a non-finite number.");
}
```

The output is quite expected. Both statements are true so the relational operation takes to this output:

```
//output
1
23.564
```

A first number is an odd number and the second one is a double 23.564 and a finite number. We can do the vice versa too. We are going to turn an integer to string.

```
//code 3.8
main() {
int myNUmber = 542;
double myDouble = 3.42;
String numberToString = myNUmber.toString();
String doubleToString = myDouble.toString();
if ((numberToString == '542' && myNUmber.isEven) && (doubleToString == '3.42' && myD\
ouble.isFinite)){
    print("Both have been converted from an even number ${myNUmber} and a finite dou\
ble ${myDouble} to string. ");
} else print("Number and double have not been converted to string.");
}

//the output
Both have been converted from an even number 542 and a finite double 3.42 to string.
```

As we progress, we will find, Dart is extremely flexible language and the syntax are simple to remember with lots of help from the core libraries.

Understanding Strings

A Dart string is a sequence of UTF-16 code units. For absolute beginners, I am going to give a short note on UTF-8, UTF-16, and UTF-32. They all store Unicode but use different bytes. Let us first try to understand the advantages of using UTF-16 code over the other two. Let us know about UTF-8.

Where ASCII characters represent the majority of texts, UTF-8 has an advantage. Like ASCII, UTF-8 also encodes all characters into 8 bits. It is the opposite for UTF-16; where ASCII is not predominant, UTF-16 has an advantage. UTF-16 remains at just 2 bytes for most characters. However, UTF-32 tries to cover all possible characters in 4 bytes, it means, processors have extra load making it pretty bloated.

The Unicode support makes Dart more powerful and you can make your mobile and web applications in any language. Let us see one example where I have tried some Bengali script.

```
//code 3.9
main(List<String> arguments) {
//print("Hello World ${IdeaProjects.calculate()}");
String bengaliString = "□□□□□□ □□□□";
String englisgString = "This is some English text.";
print("Here is some Bengali script - ${bengaliString} and some English script ${engl\
isgString}");
}


//output
Here is some Bengali script - □□□□□□ □□□□ and some English script This is some Englis\
h text.
```

While handling strings, we should remember a few things. We can use both single quote(‘’) and double quote(“”).

```
//code 3.10
main(List<String> arguments) {
String stringWithSingleQuote = 'I\'m a single quote';
String stringWithDoubleQuote = "I'm a double quote.";
print("Using delimiter in single quote - ${stringWithSingleQuote} and using delimite\
r in double quote - ${stringWithDoubleQuote}");
}
```

We can use the delimiter in both cases, but the double quote is more helpful in such cases. Watch the output:

```
//output
Using delimiter in the single quote - I'm a single quote and using delimiter in the \
double quote - I'm a double quote
```

We have put the value of expression inside a string by using our variable in this way: ${stringWith-SingleQuote}.

If you are about to express the variable in normal circumstances, you do not have to use the curly braces {}. You can use the variable this way: print(”$stringWithSingleQuote”); or print($stringWithSingleQuote); String interpolation, concatenation and even making it multi-line is quite easy in Dart. Consider this code:

```
//code 3.11
main(List<String> arguments) {
String stringInterpolation = 'string ' + 'interpolation';
print(stringInterpolation);
String multiLIneString = """
    This is
    a multi line
    string.
""";
print(multiLIneString);
}
```

Watch the output here, we have used a triple quote with either single or double quotation marks:

```
//output
string interpolation
    This is
    a multi line
    string.
```

If you want to store some constant value inside a constant string, the value cannot be variables. Consider this code:

```
//code 3.12
main(List<String> arguments) {
const aConstantInteger = 12;
const aConstantBoolean = true;
const aConstantString = "I am a constant string.";
const aValidConstantString = "this is a constant integer: ${aConstantInteger}, a con\
stant boolean: ${aConstantBoolean}, a constant string: ${aConstantString}";
print("This is a valid constant string and the output is: $aValidConstantString");
}
```

We have created a valid constant string by storing constant value inside them. The output is perfectly OK.

```
//output
This is a valid constant string and the output is: this is a constant integer: 12, a\
 constant boolean: true, a constant string: I am a constant string.
```

It will not work if you want to hold variable data inside a constant string. We have changed the code 2.10 to this:

```
//code 3.13
main(List<String> arguments) {
var aConstantInteger = 12;
var aConstantBoolean = true;
var aConstantString = "I am a constant string.";
const aValidConstantString = "this is a constant integer: ${aConstantInteger}, a con\
stant boolean: ${aConstantBoolean}, a constant string: ${aConstantString}";
print("This is a valid constant string and the output is: $aValidConstantString");
}
```

It does not work, it will give us errors. As we progress, we will learn more about string, because understanding string is very important in the context of making Flutter applications. In the next section, we will try to understand boolean; that also plays a vital role in building algorithms.

To be True or to be False

We have already seen that Dart has a type called 'bool'. The boolean literals 'true' and 'false' have type 'bool'. They are compiled time constants. Consider this code:

```
//code 3.14
main(List<String> arguments) {
bool isTrue = true;
bool isFalse = false;
if(isFalse || isTrue){
    print("It is true.");
} else print("It is false.");
}
```

We have set two boolean literals: true and false; and after that, we try to find out between two boolean literals using 'if control logic'. Between 'true' and 'false', use 'OR' conditional operator, it always chooses the 'true'.

Hence the output is:

```
//output
It is true.
```

What happens if we use 'AND' conditional operator? Let us check the code:

```
//code 3.15
main(List<String> arguments) {
bool isTrue = true;
bool isFalse = false;
if(isFalse && isTrue){
    print("It is true.");
} else print("It is false.");
}
```

Use ‘AND’, it chooses ‘false’.

```
//output
It is false.
```

This is an extremely important concept in computer science because, in our control structure, we always depend on whether a statement is ‘true’ or ‘false’. At the same time,we got a hint of relational operation; which we will discuss in a minute.

Introduction to Collections: Arrays are Lists in Dart

This is the most common collection in every programming language: array or an “ordered group of objects”. In Dart, arrays are List objects. We will address them as ‘lists’ in our future discussion. At the time of building Flutter application, we will use ‘list’ quite extensively.

Therefore, this is a very key concept. It is also the first step to learn data structures.

JavaScript array literals look like Dart lists. Here is a sample code we may consider to understand why this concept is important:

```
//code 3.15
main(List<String> arguments) {
List fruitCollection = ['Mango', 'Apple', 'Jack fruit'];
print(fruitCollection[0]);
}
```

Consider another piece of code:

```
//code 3.16
main(List<String> arguments) {
List fruitCollection = ['Mango', 'Apple', 'Jack fruit'];
var myIntegers = [1, 2, 3];
print(myIntegers[2]);
print(fruitCollection[0]);
}
```

What is the difference between these two code snippets? In the above code 2.14, we have explicitly mentioned that we are going to declare a collection of fruits. And we can pick any item from that collection from the key. As we know, when in an array key is not mentioned with the value pair, it automatically infers that the key starts from 0.

Therefore, the output of code 2.14 is 'Mango'. In the second instance we do not have any explicit declaration about the 'myInteger' lists. We have written: var myIntegers = [1, 2, 3]; however, Dart infers that list has type List<int>. Let us see the output of the code 2.15:

```
//output
3
Mango
```

If we try to inject non-integer objects to the 'myInteger' list, what happens?

```
//code 3.17
main(List<String> arguments) {
List fruitCollection = ['Mango', 'Apple', 'Jack fruit'];
var myIntegers = [1, 2, 3, 'non-integer object'];
print(myIntegers[3]);
print(fruitCollection[0]);
}
```

It did not raise any error. See the output:

```
//output
non-integer object
Mango
```

Only remember, Dart Lists use zero-based indexing like all other collections we have seen in other programming languages. Just think it as a key⇒value pair, where 0 is the index of the first value or element. As we progress, we will discuss Lists as there are other useful methods around that we will use when we will build our first mobile application. Dart Lists have many handy methods.

Get, Set and Go

In Dart, a Set is an unordered collection of unique items. There are small difference in syntax between List and Set. Let us see an example first to know more about the difference.

```
//code 3.18
main(List<String> arguments) {
var fruitCollection = {'Mango', 'Apple', 'Jack fruit'};
print(fruitCollection.lookup('Apple'));
}

//output
Apple
```

We can search the Set using the lookup() method. If we search something else, it returns 'null'.

```
//code 3.19
main(List<String> arguments) {
var fruitCollection = {'Mango', 'Apple', 'Jack fruit'};
print(fruitCollection.lookup('Something Else'));
}
//output of code 2.19
null
```

Remember one key point regarding Set and Map. When we write:

```
var myInteger = {};
```

It does not create a Set, but a Map. The syntax for map literals is similar to that of for set literals. Why does it happen? Because map literals came first, the literal {} is a default to the Map type. We can prove this by a simple test:

```
//code 3.20
main(List<String> arguments) {
var myInteger = {};
if(myInteger.isEmpty){
    print("It is a map that has no key, value pair.");
} else print("It is a set that has no key, value pair.");
}

//output
It is a map that has no key, value pair.
```

It means the map is empty. If it was a set, we would have got the output in that direction. We will see lots of examples of Sets in the future, while we build our mobile application. At present just remember, in general, a map is an object that associates keys and values. The set has also keys, but that are implicit. In cases of Sets, we call it indexes.

Let us see one example of Map type by map literals. While writing keys and values, it is important to note that each key occurs only once, but you can use the same value many times.

```
1  //code 3.21
2  main(List<String> arguments) {
3  var myProducts = {
4      'first' : 'TV',
5      'second' : 'Refrigerator',
6      'third' : 'Mobile',
7      'fourth' : 'Tablet',
8      'fifth' : 'Computer'
9  };
10 print(myProducts['third']);
11 }
```

The output is obvious : ‘Mobile’. Dart understands that the ‘myProducts’ has the type Map<String, String>(Map<Key, Value>); we could have made the key integers or number type, instead of a string type. In any Flutter application, the implementation of ‘map’ takes place very often.

```
1  //code 3.22
2  main(List<String> arguments) {
3  var myProducts = {
4      1 : 'TV',
5      2 : 'Refrigerator',
6      3 : 'Mobile',
7      4 : 'Tablet',
8      5 : 'Computer'
9  };
10 print(myProducts[3]);
11 }
```

The output is the same as before – mobile. Can we add a Set type collection of value inside a Map? Yes, we can. Consider this code:

```
//code 3.23
main(List<String> arguments) {
Set mySet = {1, 2, 3};
var myProducts = {
    1 : 'TV',
    2 : 'Refrigerator',
    3 : mySet.lookup(2),
    4 : 'Tablet',
    5 : 'Computer'
};
print(myProducts[3]);
}
```

In the above code (3.23) we have injected a collection of Set type and we also have looked up for the defining value through the Map key. Here, inside the Map key, value pair we have added the set element number 2, this way: 3 : mySet.lookup(2), and later we have told our Android Studio editor to display the value of the Map type 'myProducts' . The output is quite expected: 2.

You can create the same products lists by Map constructor. For the beginners, the term "constructor" might seem difficult. We will discuss this term in detail in our object-oriented programming category. Consider this code:

```
//code 3.24
main(List<String> arguments) {
var myProducts = Map();
myProducts['first'] ='TV';
myProducts['second'] ='Mobile';
myProducts['third'] ='Refrigerator';
if(myProducts.containsValue('Mobile')){
    print("Our products' list has ${myProducts['second']}");
}
}

//output
Our products' list has Mobile
```

Since we have had an instance (in code 3.24) of Map class, the seasoned programmer might have expected 'new Map()' instead of only 'Map()'.

As of Dart 2, the new keyword is optional. We will learn about these, in detail, in the coming object-oriented programming chapter.

We will also have a separate "Collections" chapter later, where we will learn more about List, Set and Map.

Operators are Useful

In Dart, when you use operators, you actually create expressions. If you are a seasoned programmer, you may skip this section entirely. If you are completely new, then please go on reading. Here expressions mean such examples: a++, a + b, a * b, a/b, a∼/b, a%b etc. There are many types of operators in Dart. Even absolute beginners probably have heard of arithmetic operators. Relational operators are extremely useful for the control structures.

We will have a look at them one after another.

Usual arithmetic operators are - add (+), subtract (-), multiply (*), divide (/), and modulo or remainder (%); a special operator divide, returning an integer is like this: ∼/.

Let us see one example:

```
1  //code 3.25
2  main(List<String> arguments) {
3  int aNum = 12;
4  double aDouble = 2.25;
5  var theResult = aNum ~/ aDouble;
6  print(theResult);
7  }
8  //output of code 3.25
9  5
```

Note this special operator has displayed an integer; not a double. However, if we had divided it in a plain fashion, it would have this output:

```
1  //code 3.26
2  main(List<String> arguments) {
3  int aNum = 12;
4  double aDouble = 2.25;
5  var theResult = aNum / aDouble;
6  print(theResult);
7  }
8
9  //output of code 2.26
10 5.333333333333333
```

One key feature of Dart is it supports both prefix and postfix increment and decrement operators. Let us see an example:

```
//code 3.27
main(List<String> arguments) {
int aNum = 12;
aNum++;
++aNum;
int anotherNum = aNum + 1;
print(anotherNum);
}
```

The output is as expected: 15. Prefix and postfix, both work in case of '–' also.

Equality and relational operators

The seasoned programmers know what relational operators actually mean. It is also called equality operators because '==' means equal and other relational operators usually check the equality in various forms. Let us consider some code snippets which would show us many types of relational operators at one glance.

```
//code 3.28
main(List<String> arguments) {
int firstNum = 40;
int secondNum = 41;
if (firstNum != secondNum){
    print("$firstNum is not equal to the $secondNum");
} else print("$firstNum is equal to the $secondNum");
}
//output of code 3.28
40 is not equal to the 41
```

The output is quite expected. Let us change this code a little bit:

```
//code 3.29
main(List<String> arguments) {
int firstNum = 40;
int secondNum = 40;
if (firstNum == secondNum){
    print("$firstNum is equal to the $secondNum");
} else print("$firstNum is not equal to the $secondNum");
}
```

Quite expected, it will give us the first output. Since the condition is true. Two values are equal. Let us add some more logic to our code:

```
//code 3.30
main(List<String> arguments) {
int firstNum = 40;
int secondNum = 40;
int thirdNum = 74;
int fourthNum = 56;
if (firstNum == secondNum || thirdNum == fourthNum){
    print("If choice between 'true' or 'false', the 'true' gets the precedence.");
} else print("If choice between 'true' or 'false', the 'false' gets the precedence."\
);
}

//output of code 3.30
If choice between 'true' or 'false', the 'true' gets the precedence.
```

We have learned a key concept when one value is true and another value is false, if we use 'OR', '||' operator, the 'true' value gets preceded. It is not true for the 'AND', '&&' relational operator. Watch this code:

```
//code 3.31
main(List<String> arguments) {
int firstNum = 40;
int secondNum = 40;
int thirdNum = 74;
int fourthNum = 56;
if (firstNum == secondNum && thirdNum == fourthNum){
    print("If choice between 'true' or 'false', in this case the 'true' gets the pre\
cedence.");
} else print("If choice between 'true' or 'false', in this case the 'false' gets the\
 precedence.");
}

//output of code 3.31
If choice between 'true' or 'false', in this case the 'false' gets the precedence.
```

We have used the '&&' conditional operator and here the 'false' gets preceded. The '!' sign has many roles. Consider this code snippet:

```
//code 3.32
main(List<String> arguments) {
int aNUmber = 35;
if(!(aNUmber != 150) && aNUmber <= 150){
    print("It's true");
} else print("It's false.");
}
```

Can you guess what would be the output? The first statement is false because we have negated a true statement by using '!' sign and the second statement is true, value is less than or equal to 150. Since the logical operator is '&&' or 'AND' here, it will be false. Had we used the '||', 'OR' logical operator, the output would have come out as true.

Just to remember, the '>=' operator means greater than or equal to. It is '>' greater than, it is '<' less than. Just play around your logical or relational operators as because this is one of the main pillars of computer science.

Type test operators

The "as, is, and is!" operators are handy for checking types at runtime. Consider this code:

```
//code 3.33
main(List<String> arguments) {
int myNumber = 13;
bool isTrue = true;
print(myNumber is int);
print(myNumber is! int);
print(myNumber is! bool);
print(myNumber is bool);
}

//output of code 2.33
true
false
true
false
```

Assignment operators

While assigning a value we use '=' operator. What happens when the assigned-to value is null? We use a special type of operator - '??='. Consider this code:

```
//code 3.34
main(List<String> arguments) {
int firstNum = 10;
int secondNum;
if(firstNum == 10) print("The value of ${firstNum} is set.");
if (secondNum == null) print("It is true.");
secondNum ??= firstNum;
print(secondNum);
}

//output of code 2.34
The value of 10 is set.
It is true.
10
```

In the above code 2.34, we have assigned the value of 'firstNum' to 10 and the type is an integer. So we can say,the value of 'firstNum' is set. At the same time, we have not assigned any value to the 'secondNum', so by default, it is null. After that, we have assigned a null value to an integer value by this special operator: '??='.

Almost the same thing happens in the case of compound assignment operators. Now we are going to write the above code in this way:

```
//code 3.35
main(List<String> arguments) {
int firstNum = 10;
int secondNum;
if(firstNum == 10) print("The value of ${firstNum} is set.");
if (secondNum == null) print("It is true.");
secondNum ??= firstNum;
print(secondNum);
print("After using an assignment operator, the value changes.");
secondNum += secondNum;
print(secondNum);
print("After using an assignment operator, the value changes again.");
secondNum -= secondNum;
print(secondNum);
if (secondNum == null) print("It is true.");
else print("it is false, because the 'secondNUm' has the value of ${secondNum} now."\
);
}
```

Watch the output where it is evident that we have changed the value of 'secondNum' consecutively and finally get this output:

```
//output
The value of 10 is set.
It is true.
10
After using an assignment operator, the value changes.
20
After using an assignment operator, the value changes again.
0
it is false, because the 'secondNUm' has the value of 0 now.
```

As we progress, we will find more examples of operators.

Summary of this Part

Numbers, Strings and Boolean, all they are Literals in Dart.

Consider these Literals: 1, 2.3, “Some Strings”, true, false. We need to remember a few things, such as the following:

```
var isValid = true;
1. var□data type
2. isValid□Variable Name (or Spot in the Memory)
3. true□Literal
4. We can mention the data type of the variable as ‘int’, ‘double’, ‘String’ or ‘boo\
l’. If we don’t, we can simply refer to them as ‘var’. In that case, if not mentione\
d, the data type is inferred.
5. String Interpolation is a good practice. Don’t use ‘+’ sign to add two strings.
6. We should not use the expression for a single variable name, like this: ${name}. \
We should write $name, instead.
7. Use expression for operators such as: ${number1 + number2}.
8. What will be your choice? The ‘final’ or ‘const’? It is a difficult choice. You n\
eed to remember a few things: when you choose ‘final’, it is initialized and when it\
 is accessed, the memory is allocated for it. The ‘const’ is implicitly ‘final’; it \
means when while compilation it is initialized, the memory is allocated for it.
```

So far we have learned a few key concepts of Dart language;now, we would like to implement those lessons to our first Flutter application.

Implementing Dart concepts to Flutter

As we have said earlier, Flutter is all about Widgets. Because we have just started to learn Flutter we will concentrate on basic Widgets. We will try to learn them, master them, and after that, we will proceed to the less known facts about Flutter framework.

Widgets are of two types. One is visible, and the other is invisible. In Flutter, Text() class passes a String data, which is visible. We have already passed a String data, although it does not look good, yet it is visible. The same way, we are going to use different types of buttons, those are also visible.

In a few minute we are going to use a Widget, 'RaisedButton()'; that will be also very much visible.

Incidentally, there are Widgets that are not visible, such as Column(), Row(), and many more other Widgets that we will use in the future when we will build our application. With reference to the invisible Widgets, we would like to mind you that these invisible Widgets actually help the visible Widgets to draw every pixel on the mobile screen. Therefore, they depend on each other.

To return to the previous subject of implementing our Dart concepts to our Flutter project, let us start from that point where we had left.

Our first challenge is to change the look of our application. To do that we should think our mobile screen as the 'home'. At that 'home' we should have separate sections, such as the header part, the body part, etc. We have seen in our last code Widget build() method passes one object as an argument; the name of the object is 'context'. At the same time, Flutter returns a class constructor MaterialApp() Widget through that method.

This MaterialApp() Widget is getting its all materials from the flutter package 'material.dart'. There are hundreds of classes extending one another, weaving together and form a synchronized effect on the UI design. Now, it is our duty to code that UI design with the help of many 'named parameters'.

At the very beginning of MaterialApp() Widget, we will call another Widget Scaffold(). As the word 'scaffold' literally means, it is the base platform from where we can execute our other important commands.

First of all, we need a header section where we will display the name of the application we are going to build. There is a named parameter called 'appbar'. It directly points to the another class AppBar(). Through the AppBar() class constructor we can pass another named parameter 'title', which points to the Text() constructor. Our next code snippet will show you how we are going to organize our first Flutter project:

```
// code 3.36
import 'package:flutter/material.dart';

void main() {
runApp(MyFirstApp());
}

class MyFirstApp extends StatelessWidget {

@override
Widget build(BuildContext context) {
    return MaterialApp(
    home: Scaffold(
```

```
14          appBar: AppBar(
15          title: Text(
16              'Test Your Personality...',
17              style: TextStyle(),
18              ),
19          backgroundColor: Color(0123,),
20          )
21      )
22      );
23  }
24  }
```

It will change the look of the application considerably (Figure 3.1). Although, that is just a temporary change.

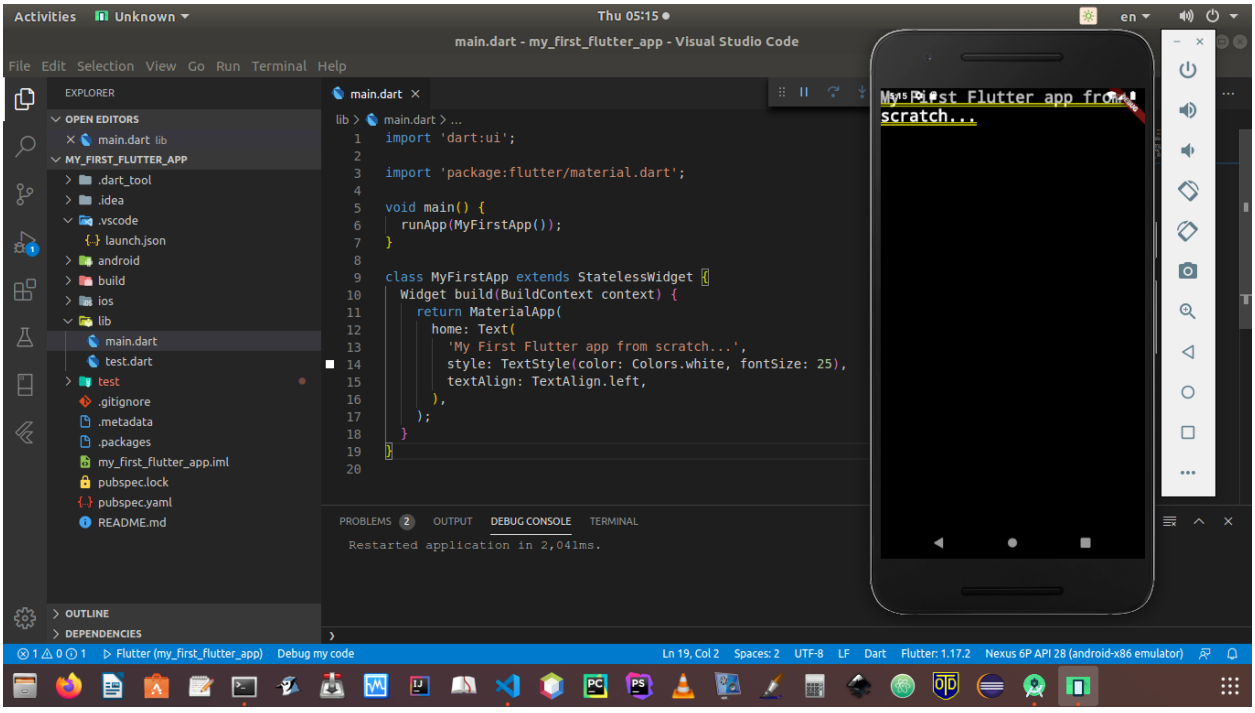


Figure 3.1 – Changing the look of the Flutter application

In any case, we need to change the look considerably better. If we had not used any Text() class constructor, and just left the Scaffold() empty, it would give us a white background (Figure 3.2).

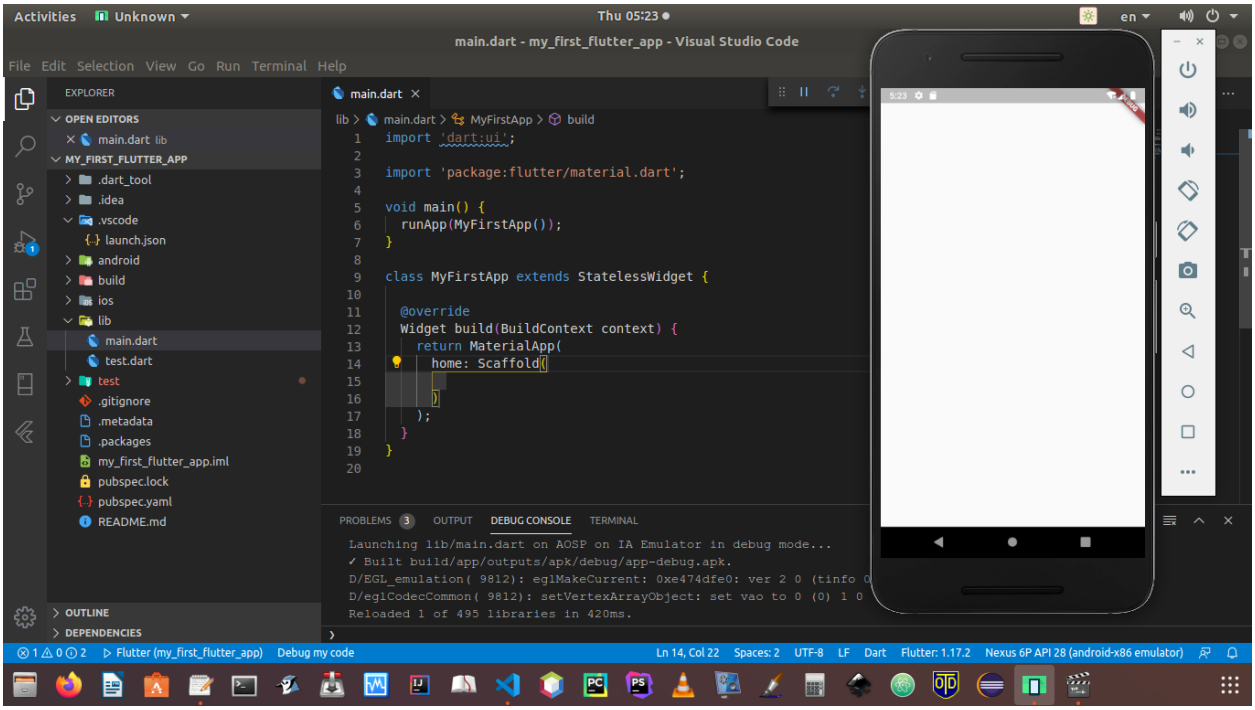


Figure 3.2 – Scaffold() class constructor empty

We could have only passed the Text() class constructor Widget with a message inside the Scaffold() Widget. That would also give us an output, where no styling would have maintained.

The following image (Figure 3.3) gives us an idea.

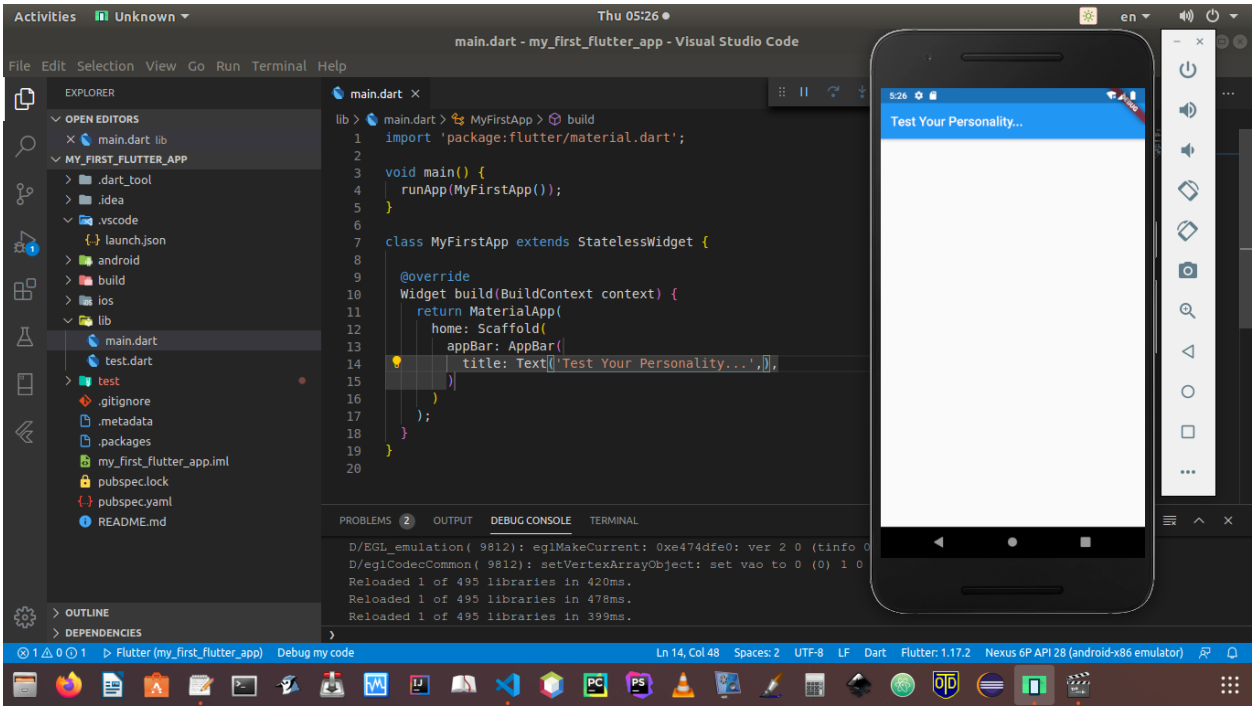


Figure 3.3 – Changing the look of the application

We have started understanding one core feature of Flutter, it mixes many Widgets, and can render a beautiful UI design. Nonetheless, it is needless to say that we need to code that design. To do that, we need to change the previous code snippet, adding many other Widgets.

```
// code 3.38
import 'package:flutter/material.dart';

void main() {
runApp(MyFirstApp());
}

class MyFirstApp extends StatelessWidget {
@override
Widget build(BuildContext context) {
    return MaterialApp(
    home: Scaffold(
        appBar: AppBar(
        title: Text(
            'Test Your Personality...',
            style: TextStyle(
            fontSize: 36,
            ),
        ),
```

```
20          backgroundColor: Color(
21              0125,
22          ),
23          ),
24      ),
25      );
26  }
27  }
```

We have changed the background color of the 'appBar' Text, adding more styling. The font size has also been changed.

All together, many different types of Widgets have acted upon collectively. It now consecutively changes the look of the application (Figure 3.4).

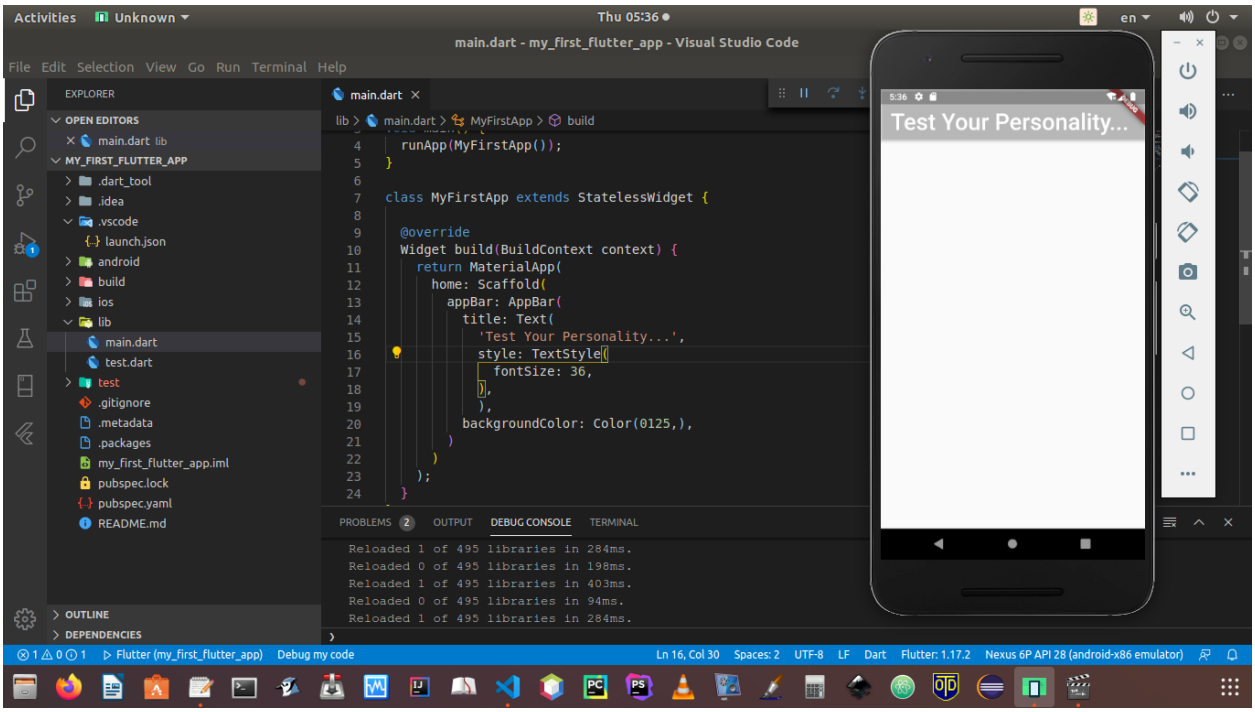


Figure 3.4 – The header text has different background color and font size

Now, we should think about the 'body' part of our application. This is the main part, where we need to do many things. We need to use many Widgets, class constructors, named parameters, etc.

If you are still not feeling quite sure about Dart programming concepts has been implemented, do not worry.

As an absolute beginner, we all have to take time to understand the core features of any programming language. We do not want to code our UI design without understanding what we are doing actually. In the next code snippet, we will introduce a Widget, which is 'list'.

We have already covered 'list' data structure before. If you do not feel very sure about it, please go back to the previous lessons, and try to understand them first.

```
// code 3.39
import 'package:flutter/material.dart';

void main() {
runApp(MyFirstApp());
}

class MyFirstApp extends StatelessWidget {
@override
Widget build(BuildContext context) {
    return MaterialApp(
    home: Scaffold(
        appBar: AppBar(
        title: Text(
            'Test Your Personality...',
            style: TextStyle(
            fontSize: 36,
            ),
        ),
        backgroundColor: Color(
            0125,
        ),
        ),
        body: Column(children: <Widget>[],),
    ),
    );
}
}
```

We have added a new line where the named parameter 'body' points to a Column Widget. As we have learned before, Column Widget is invisible, but it helps other visible Widgets to get displayed on the screen.

Watch this line:

```
body: Column(children: <Widget>[],),
```

Column Widget constructor class passes one named parameter 'children', which directly points out to a Widget 'list'. How do we know it represents a 'list' data structure? By the symbol - []. We see the second bracket open and close.

Inside that we can keep a collection of data.

We will discuss more about 'collection' and data structure when times comes. Until then, we will remember that in a collection of data structure, we can keep more than one data.

The next code snippet will clear the picture:

```
// code 3.40
import 'package:flutter/material.dart';

void main() {
runApp(MyFirstApp());
}

class MyFirstApp extends StatelessWidget {
@override
Widget build(BuildContext context) {
    return MaterialApp(
    home: Scaffold(
        appBar: AppBar(
        title: Text(
            'Test Your Personality...',
            style: TextStyle(
            fontSize: 36,
            ),
        ),
        backgroundColor: Color(
            0125,
        ),
        ),
        body: Column(
        children: [
            Text(
            'You need to answer a few questions',
            style: TextStyle(
                fontSize: 22,
            ),
            ),
        ],
        ),
    ),
    );
}
}
```

It will change the look of the connected virtual device. Inside the Column Widget, we have passed a list or collection. Now, we are not only able to pass more than one Text() class constructor, but also many more other Widgets.

Before doing that, let us check the different look that Flutter has drawn on the screen.

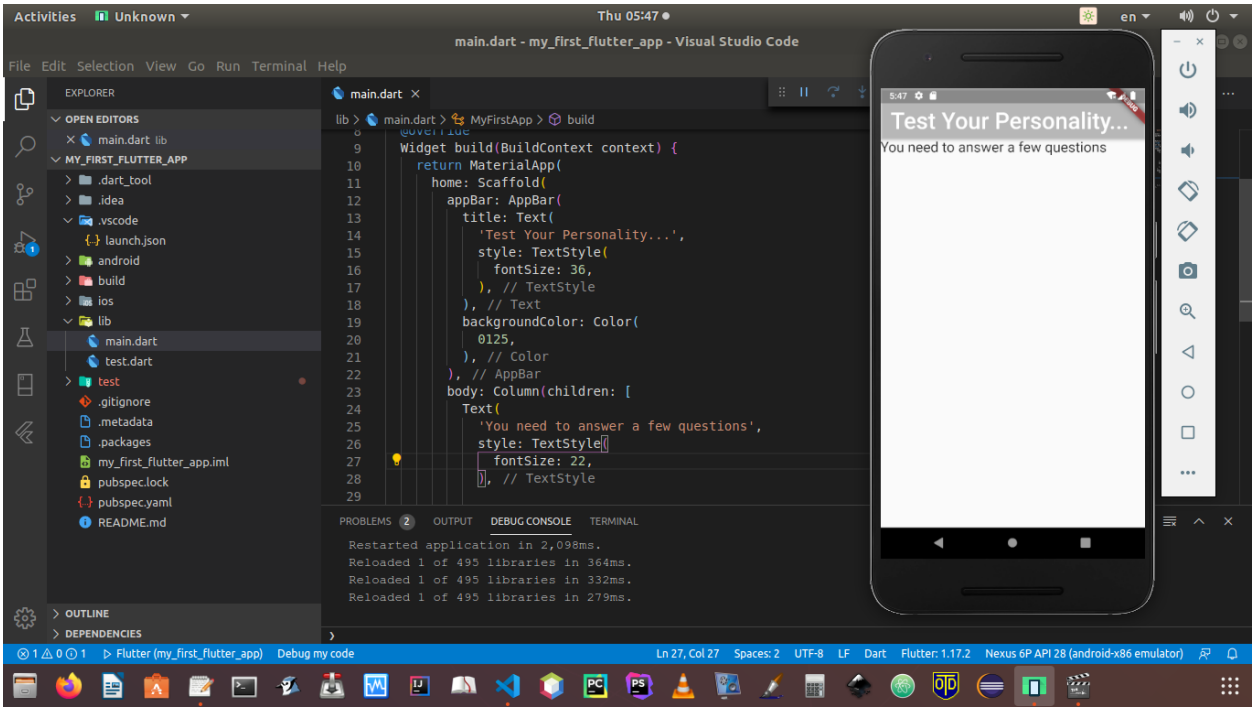


Figure 3.5 – We have successfully added more Text Widget in the body part

Watch this part of the code snippet, where we have added one element inside the list. It is a Text Widget with font size

```
children: [
            Text(
            'You need to answer a few questions',
            style: TextStyle(
                fontSize: 22,
            ),
            ),
        ],
```

Now, inside the ‘children’ list, the following part is one element.

```
Text(
            'You need to answer a few questions',
            style: TextStyle(
                fontSize: 22,
            ),
            ),
```

We can add more elements. It can be button Widget, because when you ask a question to your user, you expect an answer. Moreover, there should be more than one choices or options that the user can click.

In Flutter, there is a Widget called RaisedButton(). It is a visible Widget, and that will deliver a button to click. We can add three buttons, with three different values like the following code snippet:

```
// code 3.41
import 'package:flutter/material.dart';

void main() {
runApp(MyFirstApp());
}

class MyFirstApp extends StatelessWidget {
@override
Widget build(BuildContext context) {
    return MaterialApp(
    home: Scaffold(
        appBar: AppBar(
        title: Text(
            'Test Your Personality...',
            style: TextStyle(
            fontSize: 36,
            ),
        ),
        backgroundColor: Color(
            0125,
        ),
        ),
        body: Column(
        children: [
            Text(
            'You need to answer a few questions',
            style: TextStyle(
                fontSize: 22,
```

```
30              ),
31              ),
32              RaisedButton(
33              child: Text('You have chosen answer 1'),
34              onPressed: null,
35              ),
36              RaisedButton(
37              child: Text('You have chosen answer 1'),
38              onPressed: null,
39              ),
40              RaisedButton(
41              child: Text('You have chosen answer 1'),
42              onPressed: null,
43              ),
44          ],
45          ),
46      ),
47      );
48  }
49  }
```

Automatically it will render three buttons with text ‘You have chosen answer 1’. It should be in order, that is 1, 2,and 3. We will correct that in our next code. Before that, let us take a look at the following image (Figure 3.6).

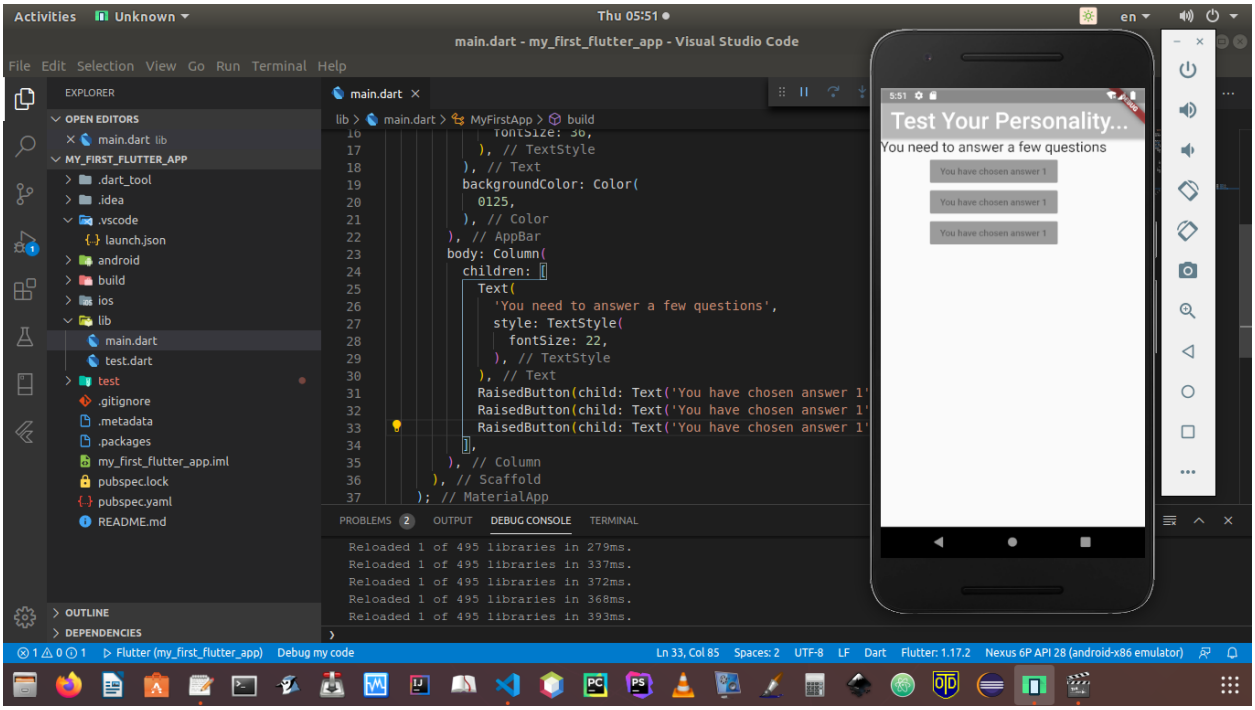


Figure 3.6 – Rendering header question and three buttons

Although we have reacted successfully to the challenge of adding more elements (here RaisedButton() Widget) to our list, yet we are not satisfied with the look. The text inside the button appears to be small. We want to make them bigger.

Now, we have learned how to add more than one Widgets inside one Widget. Therefore, it should not give us anymore trouble.

```
// code 3.42
import 'package:flutter/material.dart';

void main() {
runApp(MyFirstApp());
}

class MyFirstApp extends StatelessWidget {
@override
Widget build(BuildContext context) {
    return MaterialApp(
    home: Scaffold(
        appBar: AppBar(
        title: Text(
            'Test Your Personality...',
            style: TextStyle(
```

```
17              fontSize: 36,
18              ),
19          ),
20          backgroundColor: Color(
21              0125,
22          ),
23          ),
24          body: Column(
25          children: [
26              Text(
27              'You need to answer a few questions',
28              style: TextStyle(
29                  fontSize: 22,
30              ),
31              ),
32              RaisedButton(
33              child: Text(
34                  'You have chosen answer 1',
35                  style: TextStyle(
36                  fontSize: 18,
37                  ),
38              ),
39              onPressed: null,
40              ),
41              RaisedButton(
42              child: Text(
43                  'You have chosen answer 2',
44                  style: TextStyle(
45                  fontSize: 18,
46                  ),
47              ),
48              onPressed: null,
49              ),
50              RaisedButton(
51              child: Text(
52                  'You have chosen answer 3',
53                  style: TextStyle(
54                  fontSize: 18,
55                  ),
56              ),
57              onPressed: null,
58              ),
59          ],
```

```
60            ),
61        ),
62        );
63  }
64  }
```

We have added TextStyle() Widget inside the RaisedButton() Widget and change the font size to 18. As we had previously changed the look of the connected virtual device, this time, the body part gets a new makeover (Figure 3.7).

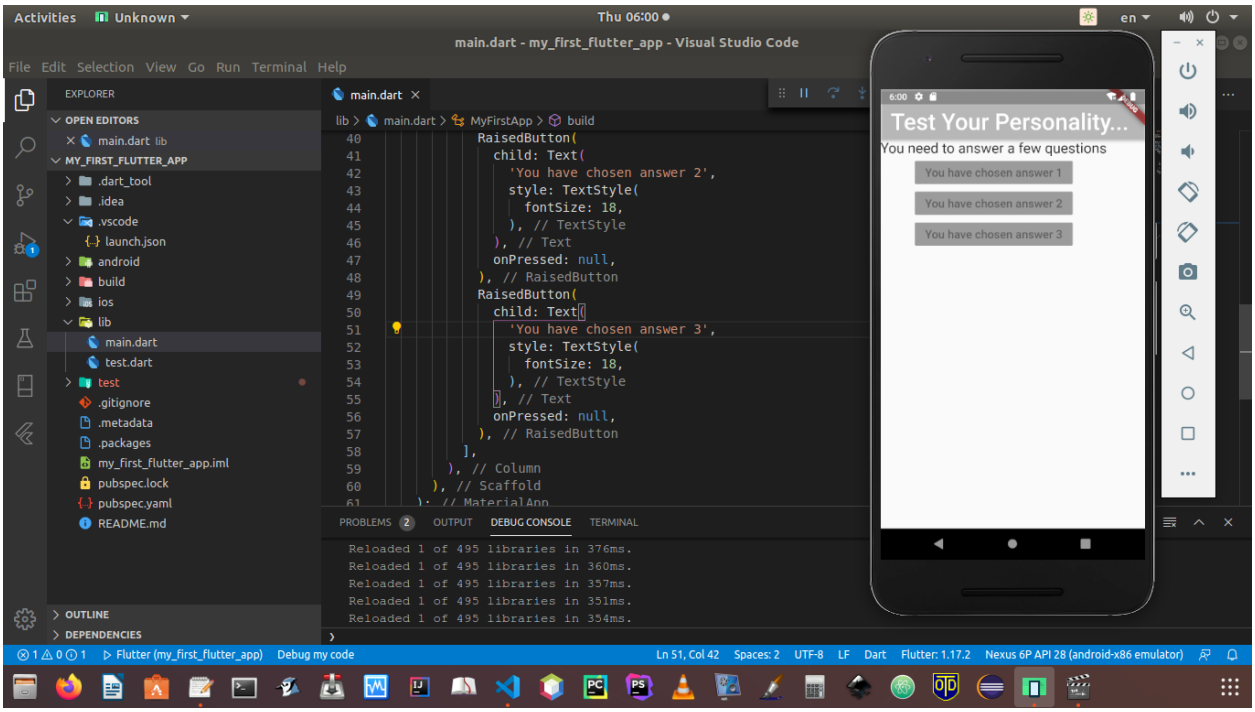


Figure 3.7 – The font size of the button has been increased

Although we have added three buttons to our question, when we click the button nothing happens.

It is happening, because we have kept the 'onPressed' named parameter to 'null'. This property is also related to the State management of any application. We will learn that in the coming chapters.

Before that, in the coming chapter, we will learn a few more features of Dart language, so that we will be able to proceed with our flutter application.

Want to read more Flutter related Articles and resources?

For more Flutter related Articles and Resources[4]

[4] https://zerodotone.net

4. Digging Deep into Dart to learn Flutter Logic

When we say: functions are objects in Dart, it seems confusing to the absolute beginners. The seasoned programmers may get the hint: Dart is an out and out object-oriented language. So even functions are objects and have a type called – Function. It means many things. One of the key things is you can assign a function to a variable, and even you can pass a function as arguments to other functions. In our Flutter app, we will see many implementations of this concept. We have already seen examples. You can also call an instance of a Dart class as if it were a function. However, to understand this key concepts that we are going to implement in our Flutter app, we need to know some more functionalities.

Control the flow of your code

Controlling the flow of your code is very important. Every programmer wants to control the logic for many reasons; one of the main reasons is the user of the software should have many options open to them. You do not know the conditions beforehand. You can only guess and as a developer, you should open as many avenues before the user as possible. There are several techniques adopted for controlling the flow of the code. The 'if and else' logic is very popular.

If and Else

Let us see an example where it works to control the flow of the code:

```
1  //code 4.1
2  main(List<String> arguments) {
3  bool firstButtonTouch = true;
4  bool secondButtonTouch = false;
5  bool thirdButtonTouch = true;
6  bool fourthButtonTouch = false;
7  if(firstButtonTouch) print("The giant starts running.");
8  else print("To stop the giant please touch the second button.");
9  if(secondButtonTouch) print("The giant stops.");
10 else print("You have not touched the second button.");
11 print("Touch any button to start the game.");
12 if(thirdButtonTouch) print("The giant goes to sleep.");
```

```
else print("You have not touched any button.");
if(fourthButtonTouch) print("The giant wakes up.");
else print("You have not touched any button.");
}

//output of code
The giant starts running.
You have not touched the second button.
Touch any button to start the game.
The giant goes to sleep.
You have not touched any button.
```

Now you can make this small code snippet more complicated.

```
//code 4.2
main(List<String> arguments) {
bool firstButtonTouch = true;
var firstButtonUntouch;
bool secondButtonTouch = false;
bool thirdButtonTouch = true;
bool fourthButtonTouch = false;
firstButtonUntouch ??= firstButtonTouch;
firstButtonUntouch = false;
if (firstButtonUntouch == false || firstButtonTouch == true) print("The giant is sle\
eping.");
else print("You need to wake up the giant. Touch the first button.");
if(firstButtonTouch == true && firstButtonUntouch == false) print("The giant starts \
running.");
print("To stop the giant please touch the second button.");
if((secondButtonTouch == true && thirdButtonTouch == true) || fourthButtonTouch == f\
alse) print("The giant stops.");
else print("You have not touched the second button.");
print("Touch any button to start the game.");
if(thirdButtonTouch) print("The giant goes to sleep.");
else print("You have not touched any button.");
if(fourthButtonTouch) print("The giant wakes up.");
else print("You have not touched any button.");
}
```

According to your complexity of code, you should arrange your ‘if and else’ logic. And your output varies.

```
//output
The giant is sleeping.
The giant starts running.
To stop the giant please touch the second button.
The giant stops.
Touch any button to start the game.
The giant goes to sleep.
You have not touched any button.
```

For ‘if and else’ logic always remember these golden rules. 1. 1. When both conditions are true, the result is true. 2. 2. When both conditions are false, the result is false. 3. 3. When one condition is true and the other condition is false, the result is false. 4. 4. When one condition is true or one condition is false, the result is true.

In the above code, I just try to give you an idea about how you can use ‘if and else’ logic where you really need it. However, this example is too simple. It can be complex when relational operators get added and the logic may become complex.

Finally, before leaving this section, I would like to show you another code snippet where the existing set of rules or principles has been changed.

```
//code 4.3
main(List<String> arguments) {
bool firstButtonTouch = true;
var firstButtonUntouch;
bool secondButtonTouch = false;
bool thirdButtonTouch = true;
bool fourthButtonTouch = false;
firstButtonUntouch ??= firstButtonTouch;
firstButtonUntouch = false;
if (firstButtonUntouch == false || firstButtonTouch == true) print("The giant is sle\
eping.");
else if (thirdButtonTouch) print("You need to wake up the giant. Touch the first but\
ton.");
else if(firstButtonTouch == true && firstButtonUntouch == false) print("The giant st\
arts running.");
else if (secondButtonTouch) print("To stop the giant please touch the second button.\
");
else if((secondButtonTouch == true && thirdButtonTouch == true) || fourthButtonTouch\
 == false) print("The giant stops.");
else if (thirdButtonTouch) print("You have not touched the second button.");
else if (secondButtonTouch) print("Touch any button to start the game.");
else if(thirdButtonTouch) print("The giant goes to sleep.");
else if (firstButtonUntouch) print("You have not touched any button.");
```

```
if(fourthButtonTouch) print("The giant wakes up.");
else print("You have not touched any button.");
}

//output of code
The giant is sleeping.
You have not touched any button.
You can change the pattern and see what happens.
```

Conditional Expression

Consider this code where we check only one condition :

```
//condition? exp1 : exp2;
int num1 = 20;
int num2 = 30;
int smallerNumber = num1 < num2? num1 : num2;
// it is expected that num1 will always be smaller
int smallNumber = num1 ?? "Default number $num2";

TIPS: For small operations, we can use this conditional expression; it is extremely \
handy and useful. When we know the output, we can go for the second one. When there \
are two numbers, we can go for the first one.
```

Looking at Looping

For loop is necessary for iterating any collections of data, with the standard ‘for’ loop. Here is a typical example of ‘for’ loop.

```
//code 4.4
main(List<String> arguments) {
var proverb = StringBuffer('As Dark as a Dungeon.');
for(var x = 0; x <= 10; x++){
    proverb.write("!");
    print(proverb);
}
}
```

In the above code we have used two in-built functions, in our following ‘functions’ and ‘object-oriented programming’ chapters, we will discuss it later. The output is as follows:

```
//output
As Dark as a Dungeon.!
As Dark as a Dungeon.!!
As Dark as a Dungeon.!!!
As Dark as a Dungeon.!!!!
As Dark as a Dungeon.!!!!!
As Dark as a Dungeon.!!!!!!
As Dark as a Dungeon.!!!!!!!
As Dark as a Dungeon.!!!!!!!!
As Dark as a Dungeon.!!!!!!!!!
As Dark as a Dungeon.!!!!!!!!!!
As Dark as a Dungeon.!!!!!!!!!!!
```

In our future discussions, we will use 'for loop' quite extensively, so at present, we stop here. I hope you get the concept.

I am going to tell you about a very interesting feature of iterating collections such as 'Set' and 'Map'. When the object you are going to iterate is Iterable, you can use 'forEach()' method. We are about to present two sets of collections; one is Set and the other is Map. In our Flutter app we will use the concept of Map. Without using Map data structure, we cannot add interactivity in our Flutter application.

```
//code 4.5
main(List<String> arguments) {
Set mySet = {1, 2, 3};
var myProducts = {
    1 : 'TV',
    2 : 'Refrigerator',
    3 : mySet.lookup(2),
    4 : 'Tablet',
    5 : 'Computer'
};
var userCollection = {"name": "John Smith", 'Email': 'john@sanjib.site'};
myProducts.forEach((x, y) => print("${x} : ${y}"));
userCollection.forEach((k,v) => print('${k}: ${v}'));
}

//output of code
1 : TV
2 : Refrigerator
3 : 2
4 : Tablet
5 : Computer
```

```
name: John Smith
Email: john@sanjib.site
```

When we do not know the current iteration counter, the 'forEach()' method is a good option. In usual cases, Iterable classes, such as, List and Set also support the 'for()' loop form of iteration. Consider this code:

```
//code 4.6
main(List<String> arguments) {
var myCollection = [1, 2, 3, 4];
for(var x in myCollection){
    print("${x}");
}
}

//output of code
1
2
3
4
```

While and Do-While

Be careful about handling the while loop. Since a while loop evaluates the condition before the loop, you must know to stop the loop at right time before it enters into infinity.

This is the pretty basic concept, but people often get confused about it.

```
//code 4.7
main(List<String> arguments) {
var num = 5;
var factorial = 1;
print("The value of the variable 'num' is decreasing this way:");
while(num >=1) {
    factorial = factorial * num;
    num--;
    print("'=>' ${num}");
}
print("The factorial  is ${factorial}");
}
```

In the above code, before the loop begins, the while() loop evaluates the condition. Since the value of the variable 'num' is 5 and it is greater than or equal to 1, the condition is true. So the loop begins. As the loop begins, we have also kept reducing the value of the variable 'num'; otherwise, it would have been entered an infinite loop.

The value of the variable reduces this way:

```
//output of code
The value of the variable 'num' is decreasing this way:
'=>' 4
'=>' 3
'=>' 2
'=>' 1
'=>' 0
The factorial  is 120
```

In case of do-while loop, it evaluates the condition after the loop.

```
//code 4.8
main(List<String> arguments) {
var num = 5;
var factorial = 1;
do {
    factorial = factorial * num;
    num--;
    print("The value of the variable 'num' is decreasing to : ${num}");
    print("The factorial  is ${factorial}");
}
while(num >=1);
}
```

We have slightly changed the code snippet so that it will show the reducing value of the variable and at the same time it will show you how the value of the factorial increases.

```
//output of code
The value of the variable 'num' is decreasing to : 4
The factorial  is 5
The value of the variable 'num' is decreasing to : 3
The factorial  is 20
The value of the variable 'num' is decreasing to : 2
The factorial  is 60
The value of the variable 'num' is decreasing to : 1
The factorial  is 120
The value of the variable 'num' is decreasing to : 0
The factorial  is 120
```

We can summarize the whole looping system. Actually they have a pattern. Once you understand the pattern, you can easily choose between ‘for’ , ‘while’ or ‘do-while’.

Understanding the Looping Patterns

I have met many students who feel confused about the ‘while’ loop. People often do not know that a ‘for’ loop can also turn into an infinite loop if it is not handled properly.

Actually, the concept of ‘loop’ is the same for every loop;be it ‘for’, ‘while’ or ‘do-while’. There are three things to remember:

```
    1. 1. counter variable
    2. 2. condition checking
    3. 3. according to the condition, increment or decrement.
```

Let us consider a code snippet:

```
void forLoopFunction(){
for(var i = 0; i <= 5; i ++){
    print(i);
}
}
void whileLoopFunction (){
var i = 0;
while(i <= 5){
    print(i);
    i++;
}
}
void doWhileLoop (){
```

```
var i = 0;
do{
    print(i);
    i++;
} while(i <= 5);
}
main(){
//print(smallerNumber);
//print(smallNumber);
forLoopFunction();
print("");
whileLoopFunction();
print("");
doWhileLoop();
}
```

I did not display the output, because you know what the output could have been. Let us consider the 'for' loop first.

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

We have started with the 'counter variable', here 'i = 0'. Then we have checked the condition: 'i <= 5'. After the second step, we have the third and the final step, according to the condition, we have incremented the value: 'i++'.

The steps are quite logical. We could not have decremented the value. It would have taken us to the infinite loop. Because in the 'condition checking' step, we will stop when the value of 'i' either less than or equal to 5.

If we had decremented the value of 'i', by writing 'i–', the condition checking would have never stopped until our computer's memory permits.

Now we have done the same thing in 'while' loop. Only the steps are a little bit different.

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

In the above code, the 'counter variable' comes before the 'while loop' originally starts. The 'while loop' starts with the 'condition checking: i <= 5'; the same thing we have seen in the second step

of ‘for’ loop. After that, according to the condition, we have incremented the value of ‘i’ inside the ‘while loop’. Once the value of ‘i’ equals 5, it immediately stops.

Now check the ‘do-while loop’ code. We start with the counter variable. And then we increment or decrement the value.

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

In the last stage we check the condition inside the ‘while loop’.

You may ask which one is better. Actually, it depends on the context. In some situations, ‘for loop’ is enough. In fact, in most cases, we can manage with the ‘for loop’. However, in some situations, we have to use while loop. In our Flutter application, we will face such situations. At that point, we will understand the actual mechanism of looping.

For Loop Labels

In some situations, we use nested for loops. Inside a ‘for loop’, we can run another ‘for loop’; in many cases, it is essential. In Dart, there is a concept called ‘Label’. We can handle the ‘outer loop’ and the ‘inner loop’ separately.

Let us see this code first:

```
void labelsLoop (){
outerloop: for(var x = 1; x <= 3; x++){
    print("One cycle of outerloop with $x starts and the whole innerloop runs.");
    innerloop: for(var y = 1; y <= 3; y++){
    if(x == 1 && y == 1){
        print("Since outerloop $x and innerloop $y both are 1, it gives no output.");
        break innerloop;
    }
    print(y);
    }
    print("One cycle of outerloop ends with $x");
}
}
main(List<String> arguments){
labelsLoop();
}
```

If you look at the output, you can understand how it works: One cycle of the outer loop with 1 starts and the whole inner loop runs.

```
Since outer loop 1 and inner loop 1 both are 1, it gives no output.
One cycle of the outer loop ends with 1
One cycle of the outer loop with 2 starts and the whole inner loop runs.
1
2
3
One cycle of the outer loop ends with 2
One cycle of the outer loop with 3 starts and the whole inner loop runs.
1
2
3
One cycle of the outer loop ends with 3
```

As you see in the above code the counter variable, condition checking and the increment parts are the same in both cases: the outer loop and the inner loop. So when the outer loop starts with 1, the inner loop inside the outer loop also starts with 1 and it should have completed the whole cycle. But we have injected an 'if clause' and told the program that when the value of the outer loop and the inner loop both are 1, break the 'inner loop'. We have used the 'Label': 'outer loop' and 'inner loop' to demarcate the loops. For the 'if clause' that particular cycle of 'inner loop' could not complete the whole cycle. However, after that, it goes on as usual.

The 'Label' is a very distinctive concept of Dart. Although, we have seen the same concept in Java.

Continue with For Loop

You have just seen how we have explicitly broken the inner loop and stopped one cycle of the inner loop. So 'break' is a very important concept while using 'for loop'. At the same breath, the 'continue' keyword also plays a very key role in 'for loop'. Let us consider this code snippet:

```
void loopContinue(){
for(var num = 1; num <= 5; num++){
    if(num % 2 == 0 ){
    print("These are all even numbers. $num");
    continue;
    } print("These are all odd numbers. $num");
}
}
main(List<String> arguments){
loopContinue();
}
```

Watch the output and you will understand how the keyword ‘continue’ works.

```
These are all odd numbers. 1
These are all even numbers. 2
These are all odd numbers. 3
These are all even numbers. 4
These are all odd numbers. 5
```

Let us change the above code a little bit and see how the output changes accordingly.

```
void loopContinue(){
for(var num = 1; num <= 5; num++){
    if(num % 2 == 0 ){
    //print("These are all even numbers. $num");
    continue;
    } print("These are all odd numbers. $num");
}
}

// output
These are all odd numbers. 1
These are all odd numbers. 3
These are all odd numbers. 5
```

According to the context, the keyword ‘continue’ means when the value is divisible by 2 and there is no remainder,just skip printing. Let us change the code again and see the output.

```
void loopContinue(){
for(var num = 1; num <= 5; num++){
    if(num % 2 == 0 ){
    print("These are all even numbers. $num");
    continue;
    } //print("These are all odd numbers. $num");
}
}

// output

These are all even numbers. 2
These are all even numbers. 4
```

Now our context has changed. When the value is divisible by 2 and there is no remainder, the keyword ‘continue’ tells the program to continue with printing the value as long as the ‘if clause’ stays true.

'Break and Continue' are two very important concepts not only in Dart, but in every programming language.

Decision making with Switch and case

In some cases, decision making seems to be easier, when you use 'Switch' instead of 'if and else' logic. Switch statements in Dart compare integers, string, or compile-time constants using the double equal sign '=='; it maintains a rule though, the compared objects must be instances of the same class and not of any of its sub types.

However, Switch statements in Dart are intended for limited circumstances, such as in interpreters or scanners. Let us see an example first to have an first-hand experience.

```
//code 4.9
main(List<String> arguments) {
//that could be the input value that would take inputs from users
var startingTime = 5;
switch (startingTime) {
    case 5:
    print("Printer Ready");
    break;
    case 6:
    print("Start printing");
    break;
    case 7:
    print("Stop for a second");
    break;
    case 8:
    print("Loading a tray and roll the paper.");
    break;
    case 9:
    print("Printer Ready, start priniting.");
    break;
    default:
    print("Default ${startingTime}");
}
}
```

When someone starts the printer it gives us output like this:

```
//output
Printer Ready
```

We have used a default clause to execute code when no case clause matches.

Controlling the flow of code is essential for many reasons. This is the base of any algorithms that instruct the machines to behave in a certain way. Building a mobile or web application needs a hundred and thousands of such instructions; algorithms could be complex and the set of algorithms require an understanding of a few other key concepts such as data structures, functions and object-oriented programming.

Digging Deep into Object-Oriented Programming

When we use a (.) notation, we usually refer to object properties or methods. A class may have properties and methods. After all, it is a blueprint of how an object will behave. How an object will behave in the future, depends on the class that has already been written. Whether a car object will start or stop, depends on that blueprint.

So we can say that objects have members consisting of functions and data; when you call a method you actually invoke it on an object.

Let us see some more examples to get acquainted with the idea of class and object. To start with let us assume a father bear is eating 6 fishes. To create the object of father bear, we need to have a bear class first where we should have one member variable or property ‘number of fish’ and one member method ‘eating that number of fish’. Ideally, both the property and the method should be annotated with the type ‘int’.

```
//code 4.10
class Bear {
int numberOfFish;
int eatFish(int numberOfFish){
    return numberOfFish;
}
}
main(List<String> arguments){
var fatherBear = new Bear();
print("Father bear eats ${fatherBear.eatFish(6)} number of fish.");
}
```

Very simple program. We have this output:

```
//output
Father bear eats 6 number of fish.
```

Can we take this code to the next level? As father bear eats fish and sleeps for some hours, he gains weight. Consider this code:

```
//code 4.11
class Bear {
int numberOfFish;
int hourOfSleep;
int weightGain;
int eatFish(int numberOfFish){
    return numberOfFish;
}
int sleepAfterEatingFish(int hourOfSleep){
    return hourOfSleep;
}
int weightGaining(int weightGain){
    weightGain = numberOfFish * hourOfSleep;
    return weightGain;
}
}
main(List<String> arguments){
var fatherBear = new Bear();
fatherBear.numberOfFish = 6;
fatherBear.hourOfSleep = 10;
fatherBear.weightGain = fatherBear.numberOfFish * fatherBear.hourOfSleep;
print("Father bear eats ${fatherBear.eatFish(fatherBear.numberOfFish)} number of fis\
h. And he sleeps for ${fatherBear.sleepAfterEatingFish(fatherBear.hourOfSleep)} hour\
s.");
print("Father bear has gained ${fatherBear.weightGaining(fatherBear.weightGain)} pou\
nds of weight.");
}
```

With the previous code we have added a few things, such as ‘hourOfSleep’ and ‘weightGain’; further we have added two related methods: ‘sleepAfterEatingFish(hourOfSleep)’ and ‘weightGaining(weightGain)’. As you see, we have passed two related parameters through those methods.

Father bear sleeps after eating the fish and gains weight. The value of weight he gains comes from the multiplication of ‘hourOfSleep’ and ‘weightGain’.

So we get this output while running this small program:

```
//output
Father bear eats 6 number of fish. And he sleeps for 10 hours.
Father bear has gained 60 pounds of weight.
```

Dart is extremely flexible language. You can write the same code in fewer lines. You do not have to use typical curly braces, and even you can omit the 'return' keyword to return the value automatically. You can also omit the 'new' word to create an instance. We are going to write the same code in this way now:

```
//code 4.12
class Bear {
int numberOfFish;
int hourOfSleep;
int weightGain;
//changing the styles of the methods completely
int eatFish(int numberOfFish) => numberOfFish;
int sleepAfterEatingFish(int hourOfSleep) => hourOfSleep;
int weightGaining(int weightGain) => weightGain = numberOfFish * hourOfSleep;
}
main(List<String> arguments){
var fatherBear = Bear(); //omitted the 'new' word
fatherBear.numberOfFish = 7;
fatherBear.hourOfSleep = 20;
fatherBear.weightGain = fatherBear.numberOfFish * fatherBear.hourOfSleep;
print("Father bear eats ${fatherBear.eatFish(fatherBear.numberOfFish)} fishes. And h\
e sleeps for ${fatherBear.sleepAfterEatingFish(fatherBear.hourOfSleep)} hours.");
print("Father bear has gained ${fatherBear.weightGaining(fatherBear.weightGain)} pou\
nds of weight.");
}
```

We have slightly changed the value of hours and the number of fishes. And the output also changes:

```
//output
Father bear eats 7 fishes. And he sleeps for 20 hours.
Father bear has gained 140 pounds of weight.
```

Before creating an object, we should have a clear picture of what that object is going to do. How we will use that object? According to that plan, we should have a blueprint,and write down the algorithms.

To make our life easier, in object-oriented programming, there is a concept called "constructor". Whenever you create an instance or object by using or by not using the 'new' keyword, inside the class, a method is automatically called, it is called the constructor method. In the next section, we will try to understand the concept.

More about Constructors

The first and foremost task of constructors is the construction of objects. In our Flutter app logic, we have already encountered such situations, where a Widget class constructor passes another Widget class constructor or methods.

Whenever we try to create an object the constructor is called first.

```
var fatherBear = Bear();
```

In the above code snippet, the left hand side of the equation represents the reference type of variable, which indicates to the new Bear object that has just been created in some memory place.

We actually try to arrange a spot in the memory for that object. The real work begins when we connect that spot with class properties and methods.

Using 'constructor' we can do that job more efficiently. Not only that, Dart allows to create more than one 'constructor', which is a great advantage. Let us write our 'Bear' class in a new way of using constructor:

```
//code 4.13
class Bear {
int numberOfFish;
int hourOfSleep;
int weightGain;
Bear(this.numberOfFish, this.hourOfSleep);
int eatFish(int numberOfFish) => numberOfFish;
int sleepAfterEatingFish(int hourOfSleep) => hourOfSleep;
int weightGaining(int weightGain) => weightGain = numberOfFish * hourOfSleep;
}
main(List<String> arguments){
var fatherBear = Bear(6, 10);
fatherBear.weightGain = fatherBear.numberOfFish * fatherBear.hourOfSleep;
print("Father bear eats ${fatherBear.eatFish(fatherBear.numberOfFish)} fishes. And h\
e sleeps for ${fatherBear.sleepAfterEatingFish(fatherBear.hourOfSleep)} hours.");
print("Father bear has gained ${fatherBear.weightGaining(fatherBear.weightGain)} pou\
nds of weight.");
}
```

Creating 'constructor' is extremely easy. Watch this line: Bear(this.numberOfFish, this.hourOfSleep);

The same class name works as a function or method and we have passed two arguments through that method. Once we get those values, we would calculate the third variable. Writing constructor

this way is known as “Syntactic Sugar”. In the later section of the book we will know more about the constructor.

Now it gets easier to pass the two values while creating the object. We could have done the same by creating constructor this way, which is more traditional:

```
//code 4.14
class Bear {
int numberOfFish;
int hourOfSleep;
int weightGain;
Bear(int numOfFish, int hourOfSleep){
this.numberOfFish = numOfFish;
this.hourOfSleep = hourOfSleep;
}
//Bear(this.numberOfFish, this.hourOfSleep);
int eatFish(int numberOfFish) => numberOfFish;
int sleepAfterEatingFish(int hourOfSleep) => hourOfSleep;
int weightGaining(int weightGain) => weightGain = numberOfFish * hourOfSleep;
}
main(List<String> arguments){
var fatherBear = Bear(6, 10);
fatherBear.weightGain = fatherBear.numberOfFish * fatherBear.hourOfSleep;
print("Father bear eats ${fatherBear.eatFish(fatherBear.numberOfFish)} fishes. And h\
e sleeps for ${fatherBear.sleepAfterEatingFish(fatherBear.hourOfSleep)} hours.");
print("Father bear has gained ${fatherBear.weightGaining(fatherBear.weightGain)} pou\
nds of weight.");
}
```

In both cases, the output is same as before:

```
//output
Father bear eats 6 fishes. And he sleeps for 10 hours.
Father bear has gained 60 pounds of weight.
```

In the above code, you can even get the object’s type very easily. We can change the type of value quite easily. Watch the main() function again:

```
//code 4.15
main(List<String> arguments){
var fatherBear = Bear(6, 10);
fatherBear.weightGain = fatherBear.numberOfFish * fatherBear.hourOfSleep;
print("Father bear eats ${fatherBear.eatFish(fatherBear.numberOfFish)} fishes. And h\
e sleeps for ${fatherBear.sleepAfterEatingFish(fatherBear.hourOfSleep)} hours.");
print("Father bear has gained ${fatherBear.weightGaining(fatherBear.weightGain)} pou\
nds of weight.");
print("The type of the object : ${fatherBear.weightGain.runtimeType}");
String weightGained = fatherBear.weightGain.toString();
print("The type of the same object has changed to : ${weightGained.runtimeType}");
}

//output of code
Father bear eats 6 fishes. And he sleeps for 10 hours.
Father bear has gained 60 pounds of weight.
The type of the object : int
The type of the same object has changed to : String
```

How to implement Classes

Now we have an idea of how classes and objects work together. A class is a blueprint that has some instance variables and methods. A class might have many tasks; but, it is a good practice and one of the major paradigms of object-oriented programming – a single class should have a single task. When many classes work together they should not be tightly coupled. They should be loosely coupled. It is a principle that is known as SOLID design principle. In Dart, we might implement the same principle while creating classes. We create a single class with a single task. We are going to create a class that will check whether the URL is secured or not.

```
//code 4.16
class CheckHTTPS {
String urlCheck;
CheckHTTPS(this.urlCheck);
bool checkingURL(String urlCheck){
if(this.urlCheck.contains("https")){
    return true;
} else return false;
}
}
main(List<String> arguments){
var newURL = CheckHTTPS('http://example.com');
```

```
print("The URL ${newURL.urlCheck} is not secured");
}
```

We get this output after checking the URL:

```
//output of code
The URL http://example.com is not secured
```

So we have some basic steps to follow. Whenever we want to create a class we should have a clear vision about what this class will do. What will be its task?

First, we need some variables. Next, we need one or more methods where we can play with these variables.

```
//code 4.17
class MyClass {
String myVariable; //property or instance variable, initially null
MyClass(this.myVariable); //constructor
String myMethod(){ //method declaration
    return "This is my method and this is ${myVariable}"; //returning value
}
}
main(List<String> arguments){
var myObject = MyClass("My String"); //creating new instance of class MyClass
print("${myObject.myMethod()}"); //printing the value
}
```

Watch the code: we have declared an instance variable first. It is of 'String' type. Since we have not initialized the variable, it is initially null. In the next step, we have constructed an object by declaring a constructor where we have passed the instance variable.

Our method's type is also 'String'. In the method, we have returned the instance variable.

In the 'main()' function, we have created an object declaring the class 'MyClass'; and at the same time, we passed a string value through the class name. We have done this for one reason: when we constructed the object by declaring the constructor, one instance variable had been passed through it. Finally, we have called the class method and display the output.

From the above example, one thing is certain. We need to know more about the functions. So in the next section, we will write some functions and will try to understand how the functions work. Remember, inside a class, we usually call a function by a different name – method. Methods are essential parts of any class because these are the action part. So we need to understand it properly.

More on Functions or Methods

We need to understand a few important features of functions before we dig deep into object-oriented programming again. The proper understanding of functions will help us to understand methods inside a class. First of all there are functions that just do nothing. It called: 'void'.

Let us consider this code:

```
//code 4.18
main(List<String> arguments){
print(showConnection());
}
//optional positional parameter
String myConnection(String dbName, String hostname, String username, [String optiona\
lPassword]){
if(optionalPassword == null){
    return "${dbName}, ${hostname}, $username";
} else return "${dbName}, ${hostname}, $username, $optionalPassword";
}
void showConnection(){
myConnection("MySQL", "localhost", "root", "*******");
}
```

We have declared the function 'showConnection()' as 'void' and want to return it through the 'main()' function. We have this output:

```
//output
bin/main.dart:4:9: Error: This expression has type 'void' and can't be used.
print(showConnection());
```

We cannot use the type 'void'. If we use, we cannot return something through that function. So from a function we always expect something. We want a function to return a value. So we are going to change the above code and write it this way:

```
//code 4.19
main(List<String> arguments){
var myConnect = myConnection("MySQL", "localhost", "root", "*******");
print(myConnect);
}
//optional positional parameter
String myConnection(String dbName, String hostname, String username, [String optiona\
lPassword]){
if(optionalPassword == null){
    return "${dbName}, ${hostname}, $username";
} else return "${dbName}, ${hostname}, $username, $optionalPassword";
}
```

The above code displays a simple program to express the "database connections" using parameters. In the above code, we have used a new concept called the optional parameter. You have already known that if we declare parameters or arguments in our functions, we have to pass them as it is. Otherwise, it gives us errors. However, we can use the concept of 'optional parameters'. In the function 'myConnections()' we have passed four arguments. Watch this line:

```
String myConnection(String dbName, String hostname, String username, [String optiona\
lPassword]){}
```

We have written the last argument as [String optionalPassword]. It means this argument is optional. If you study the Flutter packages, and default code libraries, you will find many instances of such optional parameters. In fact, we have already seen examples of named parameters and positional parameters.

You do not have to pass it when you call the function. Here the logic is simple. If the optional parameter 'optionalPassword' is not defined when we pass it inside the 'main()' function, it is treated as 'null'. Since it has been defined and passed it afterward, it is not null. Therefore, we have got this output:

```
//output of code
MySQL, localhost, root, *******
```

Now, we change the above code slightly and will not pass that argument anymore.

```
//code 4.20
main(List<String> arguments){
var myConnect = myConnection("MySQL", "localhost", "root");
print(myConnect);
}
//optional positional parameter
String myConnection(String dbName, String hostname, String username, [String optiona\
lPassword]){
if(optionalPassword == null){
    return "${dbName}, ${hostname}, $username";
} else return "${dbName}, ${hostname}, $username, $optionalPassword";
}

// The output also changes:

//output of code
MySQL, localhost, root
```

Compare these two lines before and after and we will only then understand why optional parameter is important:

```
//code 4.21
var myConnect = myConnection("MySQL", "localhost", "root", "*******");

//code 4.22
var myConnect = myConnection("MySQL", "localhost", "root");
```

If we did not declare it as optional, it would have given us an error. Let us change the optional parameter and see what type of error we get.

```
//code 4.23
main(List<String> arguments){
var myConnect = myConnection("MySQL", "localhost", "root");
print(myConnect);
}
//optional positional parameter is no more
String myConnection(String dbName, String hostname, String username, String optional\
Password){
if(optionalPassword == null){
    return "${dbName}, ${hostname}, $username";
} else return "${dbName}, ${hostname}, $username, $optionalPassword";
}
```

Now optional parameter is no more, and for that reason, we encounter this error in the output:

```
//output of code
bin/main.dart:3:31: Error: Too few positional arguments: 4 required, 3 given.
var myConnect = myConnection("MySQL", "localhost", "root");
                             ^
bin/main.dart:8:8: Context: Found this candidate, but the arguments don't match.
String myConnection(String dbName, String hostname, String username, String optional\
Password){
```

Since there was no optional parameter we had to pass the fourth argument. In the previous code snippets, we did not have to do that.

In Dart, a function is an object, for that reason, the same concept is true in case of methods and you will find it when we will discuss object-oriented programming again.

Lexical Scope in Function

This concept is extremely important as long as Dart functions are concerned. In building Flutter application this concept is also crucial.

In the later chapters, when we will dig more deeply into the object-oriented programming, we will see how the concepts of 'access' play vital roles in Dart, as well as Flutter.

Let us be back into functions again. Fist watch the code below and read the comments added with the lines:

```
//code 4.24
var outsideVariable = "I am an outsider.";
main(List<String> arguments){
//we can access the outside variable
print(outsideVariable);
// we cannot access the insider variable, it gives us error
//print(insiderVariable);
// it is an insider function
String insiderFunction(){
    // I can access the outside variable, no problem
    print("This is from the insider function.");
    print(outsideVariable);
    String insiderVariable = "I am an insider";
    print(insiderVariable); // it's okay to access this insider
}
insiderFunction();
}
```

First, we have declared a variable outside our ‘main()’ function. It is called ‘outsideVariable’. We can access that variable inside the main() function as an object. Remember, everything in Dart is an object or an instance of a class.

Second, we have declared an insider function called: ‘insiderFunction()’ of type ‘String’. Now inside that insider function, we can safely call the outsider variable. Besides, if we create another insider variable, we can also call that.

So we get this output:

```
//output of code
I am an outsider.
This is from the insider function.
I am an outsider.
I am an insider
```

As such, there is no problem regarding the output. However,it will not be the same experience if we try to call the insider variable from outside the scope of our insider function.

```
//code 4.25
var outsideVariable = "I am an outsider.";
main(List<String> arguments){
//we can access the outside variable
print(outsideVariable);
// we cannot access the insider variable, it gives us error
print(insiderVariable);
// it is an insider function
String insiderFunction(){
    // I can access the outisde variable, no problem
    print("This is from the insider function.");
    print(outsideVariable);
    String insiderVariable = "I am an insider";
    print(insiderVariable); // it's okay to access this insider
}
insiderFunction();
}

// Now, watch the output:
//output of code
bin/main.dart:11:9: Error: Getter not found: 'insiderVariable'.
print(insiderVariable);
        ^^^^^^^^^^^^^^^
```

This output takes us to another interesting concept, ‘getter’. We will see it in a minute. Before that, we should understand this ‘inside and outside’ case. This is called Lexical scope. You can call an

outside variable inside ‘main()’ function. However, if you define an object inside a function, you cannot call it outside.

A few words about Getter and Setter

We again come back to object-oriented programming with a key concept ‘getter and setter’. Whenever we write a class,it implicitly sets the value and we can get that value by using the ‘.’ notation. We can explicitly set the value and get it in this way:

```
1  //code 4.26
2  class myClass {
3      String name;
4      String get getName => name;
5      set setName(String aValue) => name = aValue;
6  }
7  main(List<String> arguments){
8  var myObject = myClass();
9  myObject.setName = "Sanjib";
10 print(myObject.getName);
11 }
```

It gives us the output ‘Sanjib’ as usual. But how does this happen? In ‘myClass’, we have ‘set’ the ‘setName()’ method by passing a parameter ‘aValue’. Later we have accessed that value through an instance (myObject.setName) of the class ‘myClass’. The interesting thing is, the method ‘setName(String aValue)’ defined inside the ‘myClass’, now works as an attribute.

You may ask why should we use ‘getter and setter’ when every class has been associated with default ‘getter and setter’?

Actually, it is a kind of overriding the default value by explicitly defining the ‘getter and setter’. The advantage is a getter has no parameter and returns a value, and the setter has one parameter and does not return a value.

More than one Constructor

In any class, there are many types of constructors that can be used in any application. As usual, we have a default constructor. We can pass parameters through it. We also have named parameters. When we call any Flutter Widget, there are hundreds of in-built classes that have many constructors defined for our use.

Let us see them in a code snippet and try to understand how they work.

```
//code 4.27
class Bear {
//reference variable
int collarID;
//default and parameterized constructor
Bear(this.collarID);
//first named constructor
Bear.firstNamedConstructor(this.collarID);
//second named constructor
Bear.secondNamedConstructor(this.collarID);
void trackingBear() {
    String color; // local variable
    print("Tracking the bear with collar ID ${collarID}");
}
}
main(List<String> arguments){
// bear1 is reference variable
// Bear() is object
var bear1 = Bear(1);
bear1.trackingBear();
var bear2 = Bear.firstNamedConstructor(2);
bear2.trackingBear();
var bear3 = Bear.secondNamedConstructor(3);
bear3.trackingBear();
}
```

In the above code, by Dart convention, when we write a class, we might have many things in place. First of all, we have a reference variable here: int collarID;

Inside the main() function, when we create an instance, we will again have a reference variable:

```
// bear1 is reference variable
// Bear() is object
var bear1 = Bear(1);
```

According to the Dart convention, the default Bear() constructor is the object here. We have passed the class level reference variable 'collar ID' through this constructor.

So while defining a class and afterward creating an instance, we have two types of reference variable: the first is class level reference variable, which can be pointed out as class properties or attributes; and the second one is object level or instance level reference variable.

In the constructor part we have one default and parameterized constructor:

```
//default and parameterized constructor
Bear(this.collarID);
Besides, we have two named constructors.
//first named constructor
Bear.firstNamedConstructor(this.collarID);
//second named constructor
Bear.secondNamedConstructor(this.collarID);
```

Through the named constructors, we have created three bear instances; moreover, each instance works as if we are using the default constructors. Finally, when you run the code, you cannot distinguish between the default and the named constructors.

```
Tracking the bear with collar ID 1
Tracking the bear with collar ID 2
Tracking the bear with collar ID 3
```

In the following code snippets, we will see how we follow this central idea of Flutter. We will code our UI and change the view of the application with the help of different widgets.

With the help of widgets we will describe what the view of the application should look look like. Suppose we want to add an icon of search, or an icon of menu button, there are default widgets available for that. Suppose we want to place the title in the middle of the body section. There are widgets for that.

Primarily every widget has two key things – configuration and state. We will discuss about State later in chapter seven. In the next chapter, we will discuss about widgets in great detail, we will also try to understand how we can configure the widgets. Whenever, we change the configuration or state of the widgets, the change takes place in the description part.

Changing the UI of the Flutter projects

In this chapter, we have learned many Dart language basic concepts. These concepts are implemented through the widget trees.

We can think of our UI as a collection of hundreds and hundreds of widgets. One widget consists of many widgets, which again consist of several other widgets, and by doing this we change the view of the Flutter application. Inside the main function of the Flutter project, we have seen the runApp() function. We pass our application object as parameter or argument inside that function like this:

```
void main() {
runApp(MyFirstApp());
}
```

But, what is this 'MyFirstApp()', it actually extends another Widget 'StatelessWidget'. The widget tree is building in that way. According to our code, 'MyFirstApp()' becomes the root widget when we pass it through the runApp() function. The root widget sits on the top of the widget tree structure. Below that tree, we start connecting other widgets and our tree grows bigger and bigger.

In the coming chapter, we will discuss this widget tree structure in great detail, because this is the core concept of any Flutter application.

In this section, we are going to change the UI of the existing Flutter application by adding and altering some widgets.

Quite naturally, the code is getting bigger and bigger, however, if we can focus on widget tree structure, we will understand how one widget consists of another widget. Just use 'CTRL+SHIFT and I', it will auto-format your code and give the perfect shape of the widget tree structure. We will also inspect the changes, especially those changes that modifies our UI design.

```
// code 4.28
import 'package:flutter/material.dart';

void main() {
runApp(MyFirstApp());
}

class MyFirstApp extends StatelessWidget {
@override
Widget build(BuildContext context) {

    return MaterialApp(
    home: Scaffold(
        appBar: AppBar(
        leading: IconButton(
            icon: Icon(Icons.menu),
            tooltip: 'Navigation menu',
            onPressed: null,
        ),
        title: Text('Test Your Knowledge...',
        style: TextStyle(fontSize: 25.00,
        fontStyle: FontStyle.normal,
        ),
        ),
```

```
25          actions: <Widget>[
26              IconButton(
27              icon: Icon(Icons.search),
28              tooltip: 'Search',
29              onPressed: null,
30              ),
31          ],
32          ),
33
34          body: Center(
35          child: Text('First Flutter Application...',
36          style: TextStyle(fontSize: 20.00,
37          fontStyle: FontStyle.italic,
38          ),
39          ),
40          ),
41          floatingActionButton: FloatingActionButton(
42          tooltip: 'Add',
43          child: Icon(Icons.add),
44          onPressed: null,
45          ),
46      ),
47      );
48  }
49  }
```

Running this code will build the new UI for us with the help of new widgets (Figure 4.1). We will change the the title of the application from 'Test your personality...' to 'Test Your Knowledge...'. The background color has also been changed.

There is a change in the look of the 'body' section. Here Scaffold() widget is the main layout of the major material components.

For the reason, under the Scaffold() widget, comes the 'appBar' widget tree.

Let us first see the new look of our Flutter application first. After that we will discuss how changing of the widgets affects the old UI design.

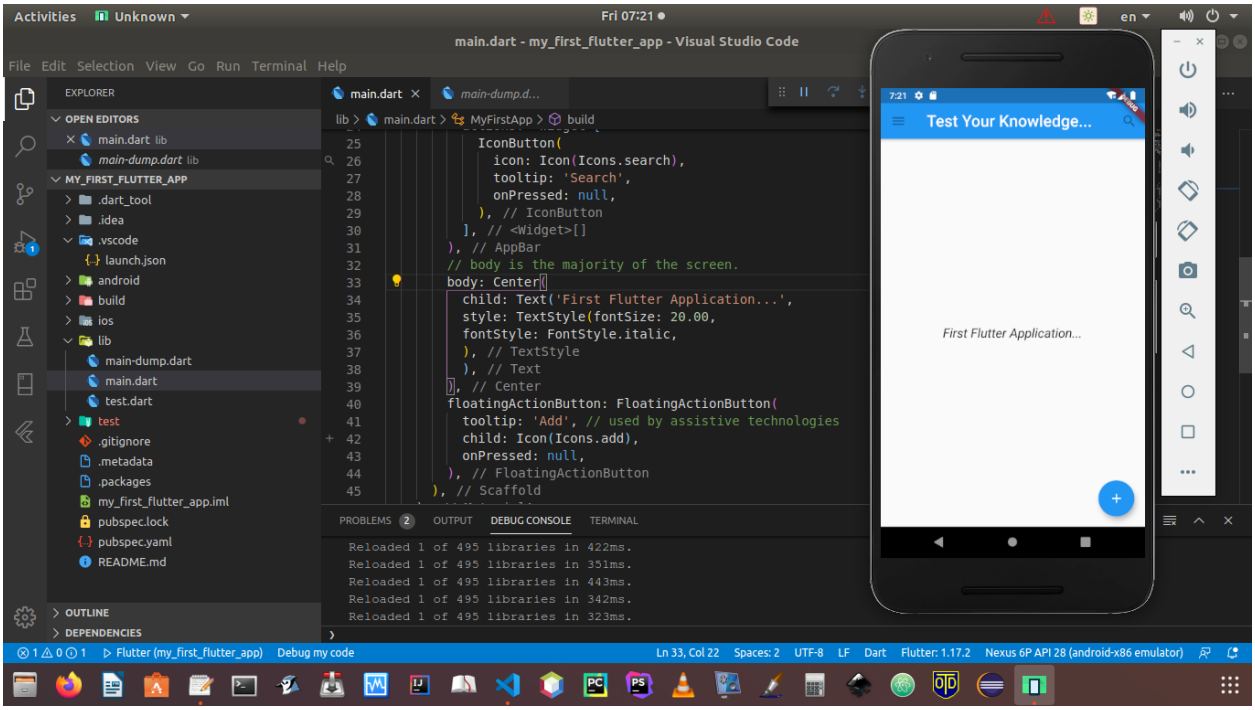


Figure 4.1 – Figure 4.1 – New widgets build a new UI for the Flutter application

Watch the 'appBar' section especially.

```
appBar: AppBar(
        leading: IconButton(
            icon: Icon(Icons.menu),
            tooltip: 'Navigation menu',
            onPressed: null,
        ),
        title: Text('Test Your Knowledge...',
        style: TextStyle(fontSize: 25.00,
        fontStyle: FontStyle.normal,
        ),
        ),
        actions: <Widget>[
            IconButton(
            icon: Icon(Icons.search),
            tooltip: 'Search',
            onPressed: null,
            ),
        ],
        ),
```

Now the appearance has been completely changed. In the upper part, we have 'leading' widget that

has three sub-trees under it - 'icon', 'tooltip and 'onPressed'. Obviously, these widgets act as named parameters. They describe what will be the view of the header section. For that reason, on the left side of the text 'Test Your Knowledge...' we have a menu button, and on the right side,we have a 'search' icon.

Until now, the body part is quite straight forward. Inside the 'body' widget, we have three sub-trees - 'child', 'style', and 'fontStyle'.

In the lower part of the body section, we have this code:

```
floatingActionButton: FloatingActionButton(

        tooltip: 'Add',
        child: Icon(Icons.add),
        onPressed: null,
        ),
```

The 'floatingActionButton' widget has again three widget sub-trees - 'tooltip', 'child', and 'on-Pressed'.

In the next figure (Figure 4.2), we have not brought a considerable change in the UI design. We have brought back the three RaisedButton() widget, just like before. Below the figure we have our changed source code snippet.

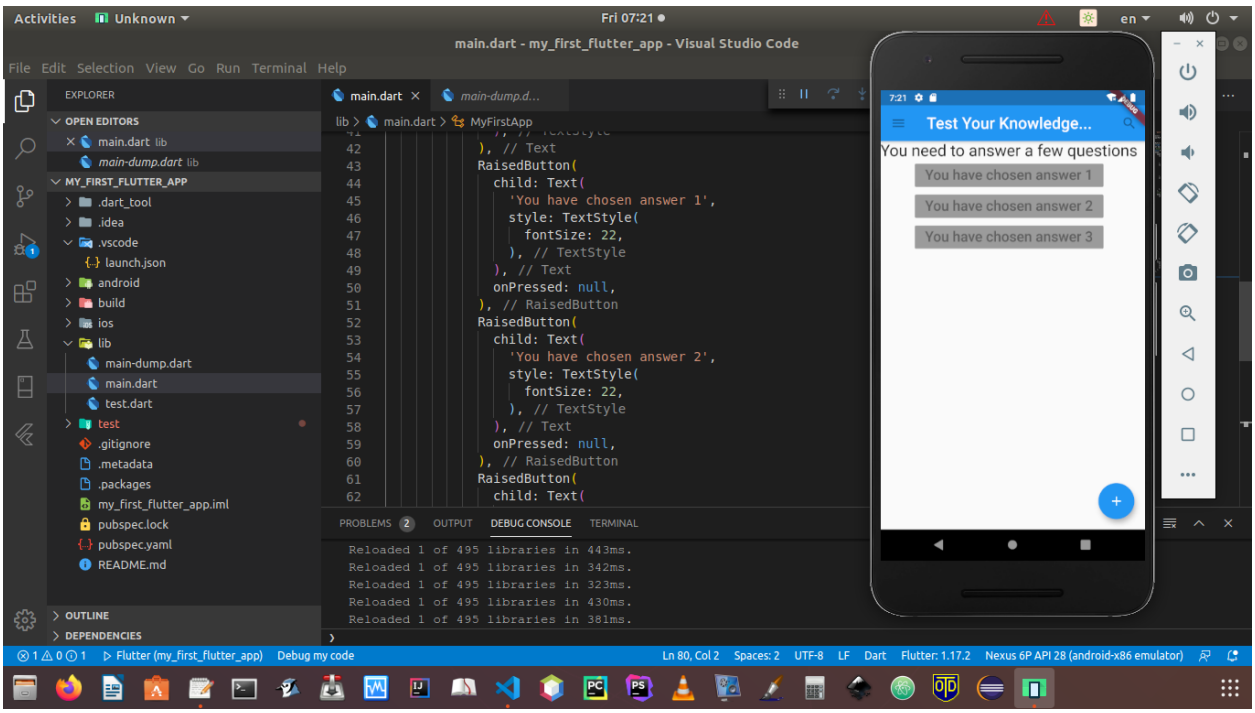


Figure 4.2 – Bringing back the RaisedButton() widgets

We have changed mainly the body part. Altering the body part gives us the above image.

```
//code 4.29
import 'package:flutter/material.dart';

void main() {
runApp(MyFirstApp());
}

class MyFirstApp extends StatelessWidget {
@override
Widget build(BuildContext context) {
    return MaterialApp(
    home: Scaffold(
        appBar: AppBar(
        leading: IconButton(
            icon: Icon(Icons.menu),
            tooltip: 'Navigation menu',
            onPressed: null,
        ),
        title: Text(
            'Test Your Knowledge...',
            style: TextStyle(
            fontSize: 25.00,
            fontStyle: FontStyle.normal,
            ),
        ),
        actions: <Widget>[
            IconButton(
            icon: Icon(Icons.search),
            tooltip: 'Search',
            onPressed: null,
            ),
        ],
        ),

        body: Column(
        children: [
            Text(
            'You need to answer a few questions',
            style: TextStyle(
                fontSize: 25,
            ),
            ),
            RaisedButton(
```

```
            child: Text(
                'You have chosen answer 1',
                style: TextStyle(
                fontSize: 22,
                ),
            ),
            onPressed: null,
            ),
            RaisedButton(
            child: Text(
                'You have chosen answer 2',
                style: TextStyle(
                fontSize: 22,
                ),
            ),
            onPressed: null,
            ),
            RaisedButton(
            child: Text(
                'You have chosen answer 3',
                style: TextStyle(
                fontSize: 22,
                ),
            ),
            onPressed: null,
            ),
        ],
        ),
        floatingActionButton: FloatingActionButton(
        tooltip: 'Add', // we can add more questions later
        child: Icon(Icons.add),
        onPressed: null,
        ),
    ),
    );
}
}
```

Now, the tree structure is getting clearer than before. We have one widget, and under that widget, we add many other widgets. It goes on like this.

However, the next figure (Figure 4.3) will show you how we have changed the entire look by adding more widgets inside our existing tree structure.

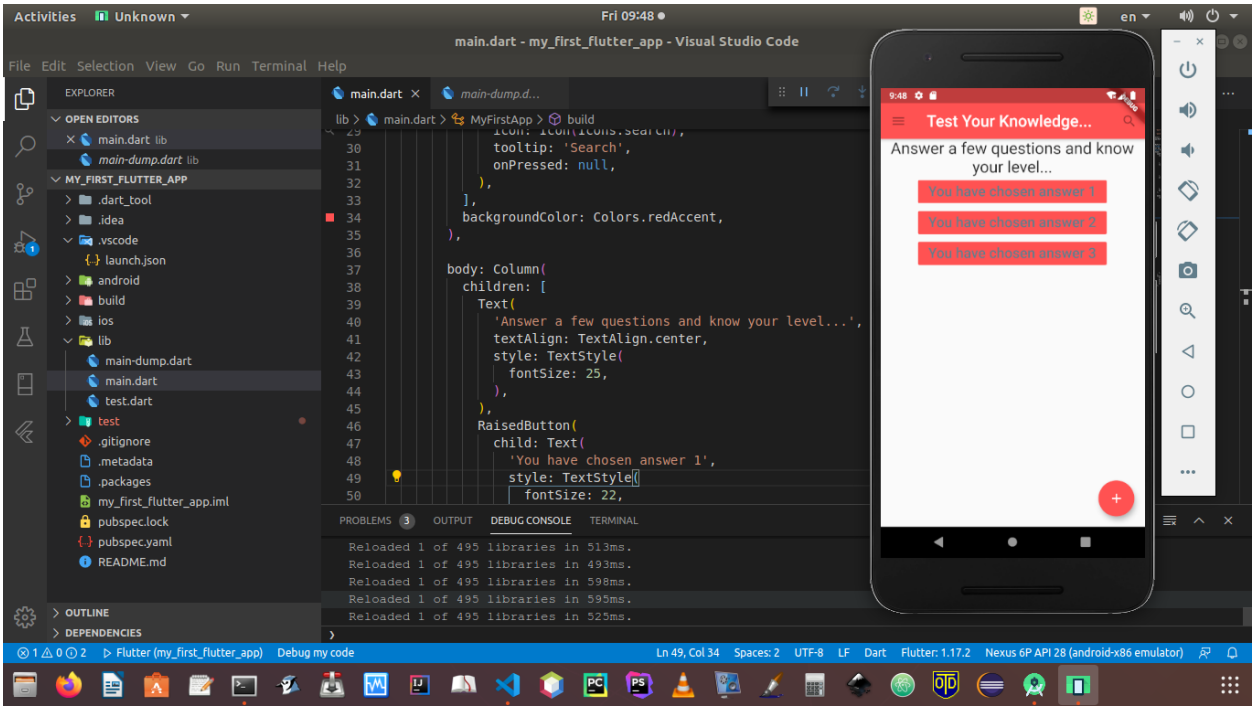


Figure 4.3 – A completely new look of our existing Flutter application

First, we will see the entire code. The previous code have slightly been changed. It makes our Flutter application more presentable.

```
//code 4.30

import 'package:flutter/material.dart';

void main() {
runApp(MyFirstApp());
}

class MyFirstApp extends StatelessWidget {
@override
Widget build(BuildContext context) {
    return MaterialApp(
    home: Scaffold(
        appBar: AppBar(
        leading: IconButton(
            icon: Icon(Icons.menu),
            tooltip: 'Navigation menu',
            onPressed: null,
        ),
        title: Text(
```

```
21              'Test Your Knowledge...',
22              style: TextStyle(
23              fontSize: 25.00,
24              fontStyle: FontStyle.normal,
25              ),
26          ),
27          actions: <Widget>[
28              IconButton(
29              icon: Icon(Icons.search),
30              tooltip: 'Search',
31              onPressed: null,
32              ),
33          ],
34          backgroundColor: Colors.redAccent,
35          ),
36
37          body: Column(
38          children: [
39              Text(
40              'Answer a few questions and know your level...',
41              textAlign: TextAlign.center,
42              style: TextStyle(
43                  fontSize: 25,
44              ),
45              ),
46              RaisedButton(
47              child: Text(
48                  'You have chosen answer 1',
49                  style: TextStyle(
50                  fontSize: 22,
51                  color: Colors.blueGrey,
52                  ),
53              ),
54              disabledColor: Colors.redAccent,
55              onPressed: null,
56              ),
57              RaisedButton(
58              child: Text(
59                  'You have chosen answer 2',
60                  style: TextStyle(
61                  fontSize: 22,
62                  color: Colors.blueGrey,
63                  ),
```

```
64              ),
65              disabledColor: Colors.redAccent,
66              onPressed: null,
67              ),
68              RaisedButton(
69              child: Text(
70                  'You have chosen answer 3',
71                  style: TextStyle(
72                  fontSize: 22,
73                  color: Colors.blueGrey,
74                  ),
75              ),
76              disabledColor: Colors.redAccent,
77              onPressed: null,
78              ),
79          ],
80          ),
81          floatingActionButton: FloatingActionButton(
82          tooltip: 'Add', // we can add more questions later
83          backgroundColor: Colors.redAccent,
84          child: Icon(Icons.add),
85          onPressed: null,
86          ),
87      ),
88      );
89  }
90  }
```

So far, we have understood that Widgets are everything in a Flutter project.

In the next chapter we will mainly discuss the Widget part. There we will dissect the above code and see what fires this change in the look

Want to read more Flutter related Articles and resources?

For more Flutter related Articles and Resources[5]

[5] https://zerodotone.net

5. How to build Flutter UI using Widgets

We have already learned that Flutter UI is built out of Widgets. In that sense, widgets are everything in Flutter. First of all, we need to memorize the basic widgets, without which we cannot start building our Flutter UI design. For another thing, we need to know how to build our widgets tree logically. Depending on how widgets sub-trees are arranged, the User Interface gets the final shape. Finally, while writing our application, we will use widgets that are sub-classes of either 'StatelessWidget or StatefulWidget'.

We will discuss State management, and 'StatefulWidget' in chapter seven; although in this chapter, we are going to get a glimpse of how 'StatefulWidget' works.

In general, the main job of a widget is the implementation of a build() function. This build() function returns the root widget, which in turn builds the UI with the help of lower-level widgets.

The job of Flutter framework is to build those widgets, synchronizing one widget with another, keeping the process moving on until the process reaches the low point to represent the 'RenderObject', which computes and gives the representation of the final UI design.

We need to remember that widgets are passed as arguments to other widgets. We have already seen how one widget takes a number of different widgets as named arguments. We have seen that our Flutter application 'MyFirstApp()' extends 'StatelessWidget' class and the calls the Widget build() function that passes 'BuildContext context' as its only argument.

The 'MaterialApp' widget acts as the root widget. By and large, it passes two named arguments 'title' and 'home'. The 'title' could be a simple text, the title of the application we are going to build. Both are passed as arguments to other widgets, one of them is Scaffold(), another important widget.

In turn, the Scaffold widget takes a number of different widgets as named arguments, of which 'AppBar' widget plays a major role. The 'AppBar' widget again passes different types of widgets as named arguments.

We can define our title widget here also. This pattern of calling and adding lower-level widgets one after another, goes on until your design is complete.

Common Widgets in Flutter

As we were saying, we need to use hundreds of widgets to build our UI. However, keeping all the widgets in one place, especially in 'main.dart' file, does not look clean. It also makes our code unnecessarily lengthy, hard to debug.

The advantage of object-oriented programming is that we can use our objects as different module. It keeps the modularity. The objects are not tightly coupled, they should remain loosely coupled.

For that reason, we will use break our code snippets in different dart files and finally import them in the ‘main.dart’ file.

We will see the code snippet first, after that we will see how it affects our UI design. After that we will discuss the widgets used in our code.

```
1  //code 5.1
2  // my_appbar.dart
3  import 'package:flutter/material.dart';
4
5  class MyAppBar extends StatelessWidget {
6  MyAppBar({this.title});
7
8  final Widget title;
9
10 @override
11 Widget build(BuildContext context) {
12     return Container(
13     height: 116.0,
14     decoration: BoxDecoration(color: Colors.redAccent),
15
16     child: Row(
17
18         children: <Widget>[
19         IconButton(
20             icon: Icon(Icons.menu),
21             tooltip: 'Navigation menu',
22             onPressed: null,
23         ),
24
25         Expanded(
26             child: title,
27         ),
28         IconButton(
29             icon: Icon(Icons.search),
30             tooltip: 'Search',
31             onPressed: null,
32         ),
33         ],
34     ),
35     );
```

```
}
}

-----------------

//my_scaffold.dart
import 'package:flutter/material.dart';
import 'package:my_first_flutter_app/chap5_widgets/my_appbar.dart';

class MyScaffold extends StatelessWidget {
@override
Widget build(BuildContext context) {

    return Material(

    child: Column(
        children: <Widget>[
        MyAppBar(
            title: Text(
            'Test Your Knowledge...',
            style: Theme.of(context).primaryTextTheme.headline6,
            ),
        ),
        Expanded(
            child: Center(
            child: Text('Here we will place our body widget...',
            style: TextStyle(fontSize: 25),
            ),
            ),
        ),
        ],
    ),
    );
}
}

--------------------
//main.dart
import 'package:flutter/material.dart';
import 'package:my_first_flutter_app/chap5_widgets/my_scaffold.dart';

void main() {
runApp(MyFirstApp());
```

```
79  }
80
81  class MyFirstApp extends StatelessWidget {
82  @override
83  Widget build(BuildContext context) {
84      return MaterialApp(
85      title: 'My app',
86      home: MyScaffold(),
87      );
88  }
89  }
```

We have used three different Dart files. While talking about this, we need to remember that either we can keep these files in 'lib' folder along with the 'main.dart' file, or we can create separate folders inside the 'lib' folder, and keep those files there. Wherever, we keep this file, while importing we have to mention the full path. We will come to that point in a minute, before we need to see how our widgets build the UI.

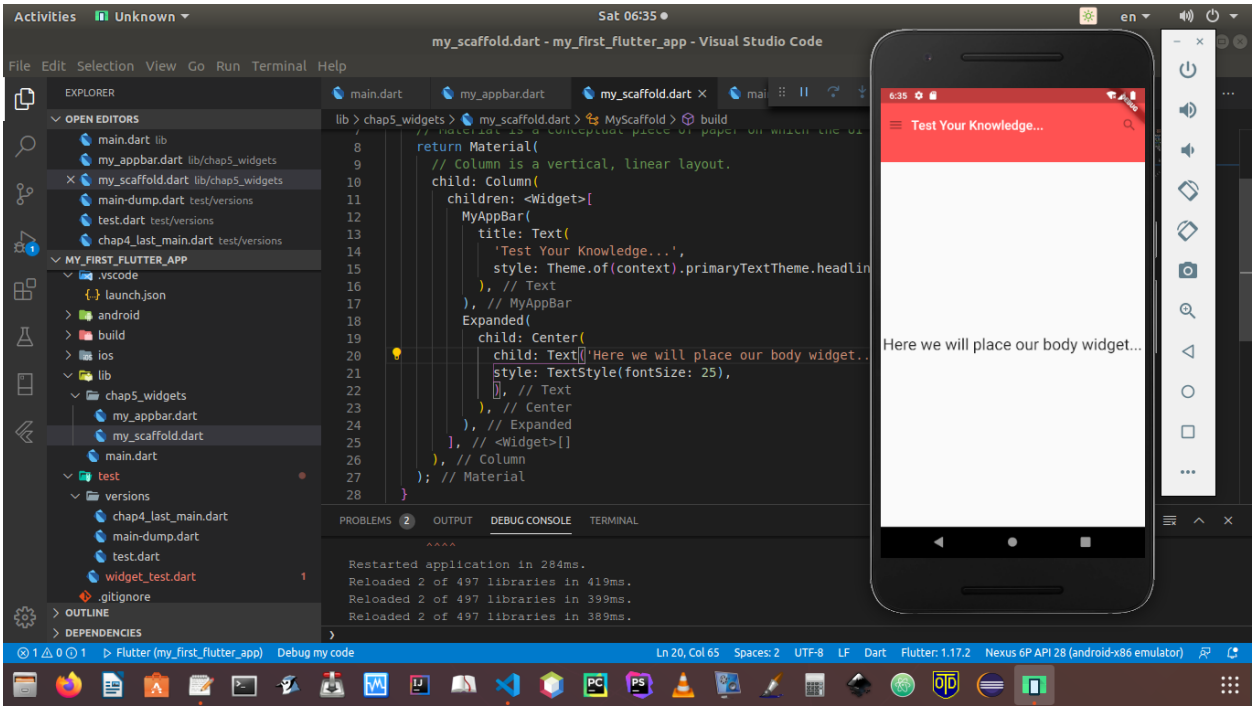


Figure 5.1 – How widgets build the UI of the application

It represents a simple UI design. To make it happen we have used three different Dart files to make our code clean.

Now, let us watch the widgets used in those files. Let us start with 'my_appbar.dart' file. It describes the 'AppBar' widget. We could have called it directly, while building the UI. However, we have

decided to keep it in a generic class, where we have passed one named argument ‘tile’ through its constructor.

```
MyAppBar({this.title});

final Widget title;
```

Fields (here ‘title’) in a Widget subclass (here ‘MyAppBar’) are always marked “final”. It is a convention, and we will maintain always.

Here as a lower-level widget we first use the ‘Container’. That widget, in turn, passes many named arguments as lower-level widgets. As the ‘child’ it passes the ‘Row’ widget and inside passes a list of ‘children’ widgets.

```
children: <Widget>[]
```

The above line describes what type of Widget we should pass as a list. We have learned the syntax of list data structure in Dart.

Whenever we use the lower-level widgets we always maintain the structure of hierarchies. A mobile screen will finally render this UI design. We have to build our widget trees keeping that in our mind.

Powerful Basic Widgets

Flutter tools come with hundreds of powerful widgets. We will start with some most commonly used basic widgets that we have already seen. Text widget helps us to build a load of styled text within our application. We will see their implementation as we progress.

Then come Row and Column. Both have extreme flexibility to create our layouts. When we want to build layout in horizontal, linear directions, we use Row. If we need vertically aligned layout, we use Column widget. Container widget is another very important widget that we will use in our application. It creates a rectangular visual element. We can skew the whole Container widget using matrix, we can also decorate with a ‘BoxDecoration’ that helps to build background, border, or a shadow. The size of the Container can be controlled by margins, padding, and other constraints.

AppBar class produces one of the most commonly used widgets. In the previous code, we have seen its implementation. It mainly displays the toolbar widgets, leading, title, icons, and other actions. For rendering different types of icons, it uses IconButton widget, that also has many sub-trees of widgets that we will see in a minute.

AppBar is a material design app bar that we may treat as the header section. It comes under the Scaffold widget, which is also one of the most commonly used widgets. Let us see our next code snippet and the figure of our changed application. After that, we will discuss the Scaffold widget.

```
//code 5.2
// testing_my_first_app.dart
import 'package:flutter/material.dart';

class MyFirstApp extends StatelessWidget {
@override
Widget build(BuildContext context) {

    return Scaffold(
    appBar: AppBar(
        leading: IconButton(
            icon: Icon(Icons.menu),
            tooltip: 'Navigation menu',
            onPressed: null,
        ),
        title: Text(
            'Test Your Knowledge...',
            style: TextStyle(
            fontSize: 25.00,
            fontStyle: FontStyle.normal,
            ),
        ),
        actions: <Widget>[
            IconButton(
            icon: Icon(Icons.search),
            tooltip: 'Search',
            onPressed: null,
            ),
        ],
        backgroundColor: Colors.redAccent,
        ),

    body: Column(
        children: [
            Text(
            'Answer a few questions and know your level...',
            textAlign: TextAlign.center,
            style: TextStyle(
                fontSize: 25,
            ),
            ),
            RaisedButton(
            child: Text(
```

```
                'You have chosen answer 1',
                style: TextStyle(
                fontSize: 22,
                color: Colors.blueGrey,
                ),
            ),
            disabledColor: Colors.redAccent,
            onPressed: null,
            ),
            RaisedButton(
            child: Text(
                'You have chosen answer 2',
                style: TextStyle(
                fontSize: 22,
                color: Colors.blueGrey,
                ),
            ),
            disabledColor: Colors.redAccent,
            onPressed: null,
            ),
            RaisedButton(
            child: Text(
                'You have chosen answer 3',
                style: TextStyle(
                fontSize: 22,
                color: Colors.blueGrey,
                ),
            ),
            disabledColor: Colors.redAccent,
            onPressed: null,
            ),
        ],
        ),
    floatingActionButton: FloatingActionButton(
        tooltip: 'Add',
        child: Icon(Icons.add),
        onPressed: null,
    ),
    );
}
}

// Scaffold is a layout for the major Material Components.
```

```
// body is the majority of the screen.

//main.dart
import 'package:flutter/material.dart';
import 'package:my_first_flutter_app/chap5_widgets/my_first_app.dart';

void main(List<String> args) => runApp(MyFirstApp());
```

We have used two Dart files, 'testing_my_first_app.dart' and the 'main.dart'. We have made our 'main.dart' file just a one-line code:

```
void main(List<String> args) => runApp(MyFirstApp());
```

We have seen the fat arrow notation '⇒' before in our Dart practices. It can reduce any function to one line of code, when it calls one function.

The 'testing_my_first_app.dart' file starts with the Scaffold widget that has two main sub-trees widgets – appbar, and body. Let us see some code snippets from the appbar widget first.

```
appBar: AppBar(
        leading: IconButton(
            icon: Icon(Icons.menu),
            tooltip: 'Navigation menu',
            onPressed: null,
        ),
        title: Text(
            'Test Your Knowledge...',
            style: TextStyle(
            fontSize: 25.00,
            fontStyle: FontStyle.normal,
            ),
        ),
        actions: <Widget>[
            IconButton(
            icon: Icon(Icons.search),
            tooltip: 'Search',
            onPressed: null,
            ),
        ],
        backgroundColor: Colors.redAccent,
        ),
```

The 'appbar' named argument points to AppBar Class constructor that passes three main widgets, leading, title and actions. Between these three widgets, 'actions' is a list widget. Although, at present,

we have only one element, IconButton class constructor that passes several other widgets as named parameters.

We are not repeating the 'body' widget, that as a named argument points to the one of the most commonly used widgets, Column. The Column has many other widgets that are being pointed by the named parameter 'children' which represents a 'list' data structure (Figure 5.2).

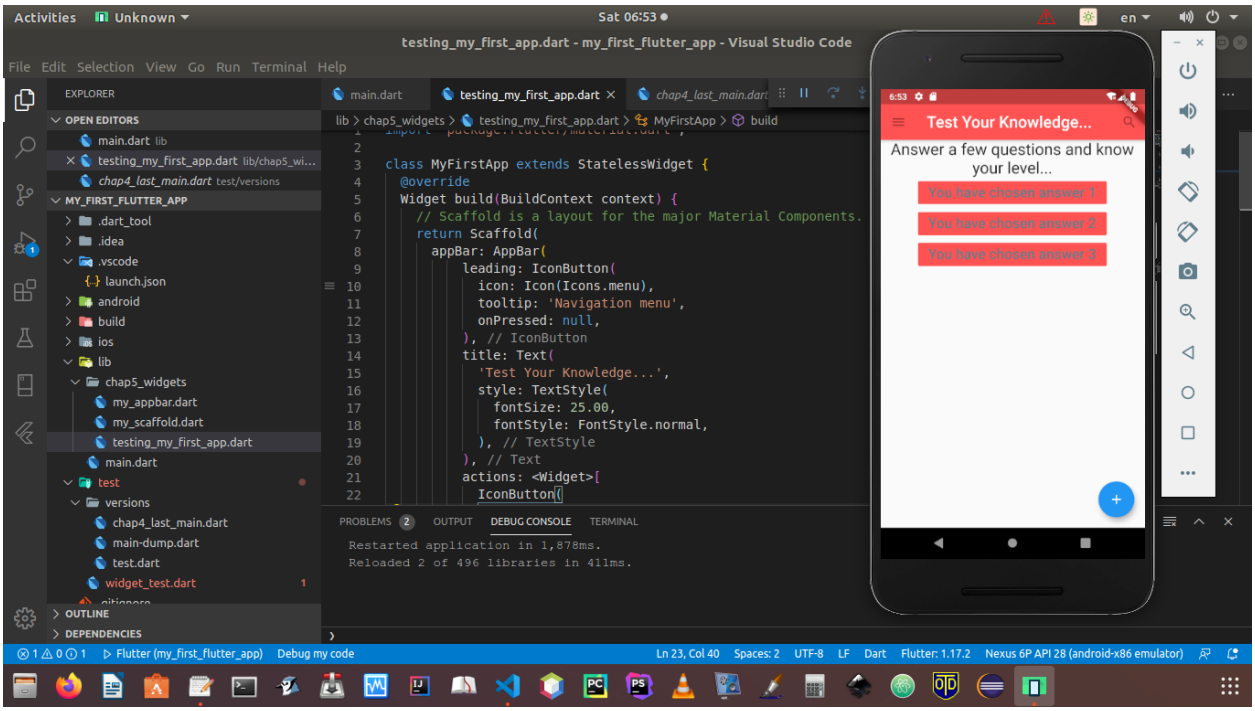


Figure 5.2 – The Scaffold widget and its widget sub-trees

So far, we have tried to maintain our code design as it was. Now we are going to add more functionalities to our application. So far we have used the 'StatelessWidget'. For the first time, we are going to change the state of our application. As we have found before, 'state' is used to maintain the data or information used by the application. When a user logs into our application, we should try to maintain the state as long as the user gets logged in.

We will discuss this elaborately in the coming chapter seven. We will also implement that in our application in due course. For the time being, we must remember that the 'Widget state' holds the current user input. Suppose we click a button, or icon, it automatically gives us a Text output and maintains that state for a few seconds. For the 'App state', this duration could be longer, because a user may want to gets logged in to our application for hours.

To get an idea, we have created three separate Dart files, 'my_stateless_scaffold.dart', 'my_first_-app.dart', and the 'main.dart' file, as usual.

We have also created a folder 'chap5_widgets' inside the 'lib' folder, where we put our code snippets. We need to import the full path so that the application works in a synchronized way.

```dart
//code 5.3
//my_stateless_scaffold.dart
import 'package:flutter/material.dart';

final GlobalKey<ScaffoldState> scaffoldKey = GlobalKey<ScaffoldState>();
final SnackBar snackBarOne = const SnackBar(
    content: Text(
'Alert has been pressed!',
style: TextStyle(fontSize: 30),
));
final SnackBar snackBarTwo = const SnackBar(
    content: Text(
'Search has been pressed!',
style: TextStyle(fontSize: 30),
));
final SnackBar snackBarThree = const SnackBar(
    content: Text(
'Navigation has been pressed!',
style: TextStyle(fontSize: 30),
));

void clickNextPage(BuildContext context) {
Navigator.push(context, MaterialPageRoute(
    builder: (BuildContext context) {
    return Scaffold(
        appBar: AppBar(
        title: const Text('Know Yourself...'),
        ),
        body: const Center(
        child: Text(
            'Dig deep into every layer of your mind to find yourself...',
            style: TextStyle(fontSize: 24),
            textAlign: TextAlign.center,
        ),
        ),
    );
    },
));
}

class MyStatelessScaffoldWidget extends StatelessWidget {

MyStatelessScaffoldWidget({Key key}) : super(key: key);
```

```
44
45  @override
46  Widget build(BuildContext context) {
47      return Scaffold(
48      key: scaffoldKey,
49      appBar: AppBar(
50          actions: <Widget>[
51          IconButton(
52              icon: const Icon(Icons.add_alert),
53              tooltip: 'Show Snackbar',
54              onPressed: () {
55              scaffoldKey.currentState.showSnackBar(snackBarOne);
56              },
57          ),
58          IconButton(
59              icon: Icon(Icons.search),
60              tooltip: 'Search',
61              onPressed: () {
62              scaffoldKey.currentState.showSnackBar(snackBarTwo);
63              },
64          ),
65          IconButton(
66              icon: const Icon(Icons.navigate_next),
67              tooltip: 'Next page',
68              onPressed: () {
69              clickNextPage(context);
70              },
71          ),
72          ],
73          leading: IconButton(
74          icon: Icon(Icons.menu),
75          tooltip: 'Navigation menu',
76          onPressed: () {
77              scaffoldKey.currentState.showSnackBar(snackBarThree);
78          },
79          ),
80          title: Text(
81          'Knowledge Test',
82          style: TextStyle(
83              fontSize: 25.00,
84              fontStyle: FontStyle.normal,
85          ),
86          ),
```

```
87          backgroundColor: Colors.redAccent,
88      ),
89      body: Column(
90          children: [
91          Text(
92              'Answer a few questions and know your level...',
93              textAlign: TextAlign.center,
94              style: TextStyle(
95              fontSize: 25,
96              ),
97          ),
98          RaisedButton(
99              child: Text(
100             'You have chosen answer 1',
101             style: TextStyle(
102                 fontSize: 22,
103                 color: Colors.blueGrey,
104             ),
105             ),
106             disabledColor: Colors.redAccent,
107             onPressed: null,
108         ),
109         RaisedButton(
110             child: Text(
111             'You have chosen answer 2',
112             style: TextStyle(
113                 fontSize: 22,
114                 color: Colors.blueGrey,
115             ),
116             ),
117             disabledColor: Colors.redAccent,
118             onPressed: null,
119         ),
120         RaisedButton(
121             child: Text(
122             'You have chosen answer 3',
123             style: TextStyle(
124                 fontSize: 22,
125                 color: Colors.blueGrey,
126             ),
127             ),
128             disabledColor: Colors.redAccent,
129             onPressed: null,
```

```
        ),
        ],
    ),
    floatingActionButton: FloatingActionButton(
        tooltip: 'Add', // we can add more questions later
        backgroundColor: Colors.redAccent,
        child: Icon(Icons.add),
        onPressed: null,
    ),
    );
}
}

//my_first_app.dart
import 'package:flutter/material.dart';
import 'package:my_first_flutter_app/chap5_widgets/my_stateless_scaffold.dart';

class MyFirstApp extends StatelessWidget {

@override
Widget build(BuildContext context) {
    return MaterialApp(
    home: MyStatelessScaffoldWidget()
    );
}
}

//main.dart
import 'package:flutter/material.dart';
import 'package:my_first_flutter_app/chap5_widgets/my_first_app.dart';

void main(List<String> args) => runApp(MyFirstApp());
```

Although the whole code snippets look pretty lengthy, most parts are repeating the old code, especially the 'body' part.

In the 'appbar' section, we have some few new features, like 'alert' icon and the 'arrow' that points to a completely new page.

Let us first see the display. Now the 'appbar' section has a complete new look with more icons and the arrow symbol.

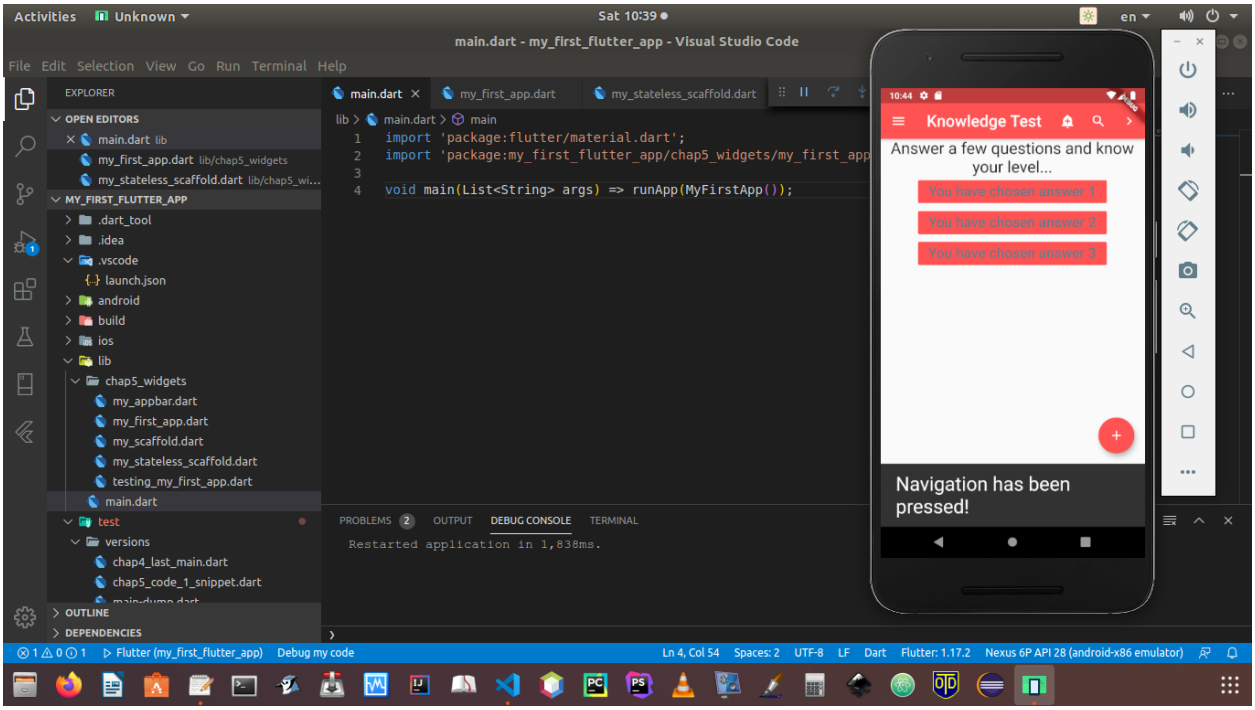


Figure 5.3 – A complete new look of 'appbar' widget, where widget state has been managed

Now the named parameter 'appbar' points to the AppBar class constructor that passes many named parameters, of which 'actions' points to a widget 'list'.

```
actions: <Widget>[
        IconButton(
            icon: const Icon(Icons.add_alert),
            tooltip: 'Show Snackbar',
            onPressed: () {
            scaffoldKey.currentState.showSnackBar(snackBarOne);
            },
        ),
        IconButton(
            icon: Icon(Icons.search),
            tooltip: 'Search',
            onPressed: () {
            scaffoldKey.currentState.showSnackBar(snackBarTwo);
            },
        ),
        IconButton(
            icon: const Icon(Icons.navigate_next),
            tooltip: 'Next page',
            onPressed: () {
            clickNextPage(context);
```

```
            },
        ),
        ],
        leading: IconButton(
        icon: Icon(Icons.menu),
        tooltip: 'Navigation menu',
        onPressed: () {
            scaffoldKey.currentState.showSnackBar(snackBarThree);
        },
        ),
```

So far we have maintained the 'onPressed' to 'null'. For the first time we have used an anonymous function that returns a chained method that connects different widget methods. In the later sections of this chapter, we have discussed anonymous or lambda function of Dart. If you are a beginner, please go through it.

Let us take a look at the last one:

```
leading: IconButton(
        icon: Icon(Icons.menu),
        tooltip: 'Navigation menu',
        onPressed: () {
            scaffoldKey.currentState.showSnackBar(snackBarThree);
        },
        ),
```

It leads us to the Navigation menu. Here according to the 'tooltip' name, we have chosen the 'Icons.menu' field in the Icon class constructor.

Let us see what happens when we press the 'alert' icon (Figure 5.4).

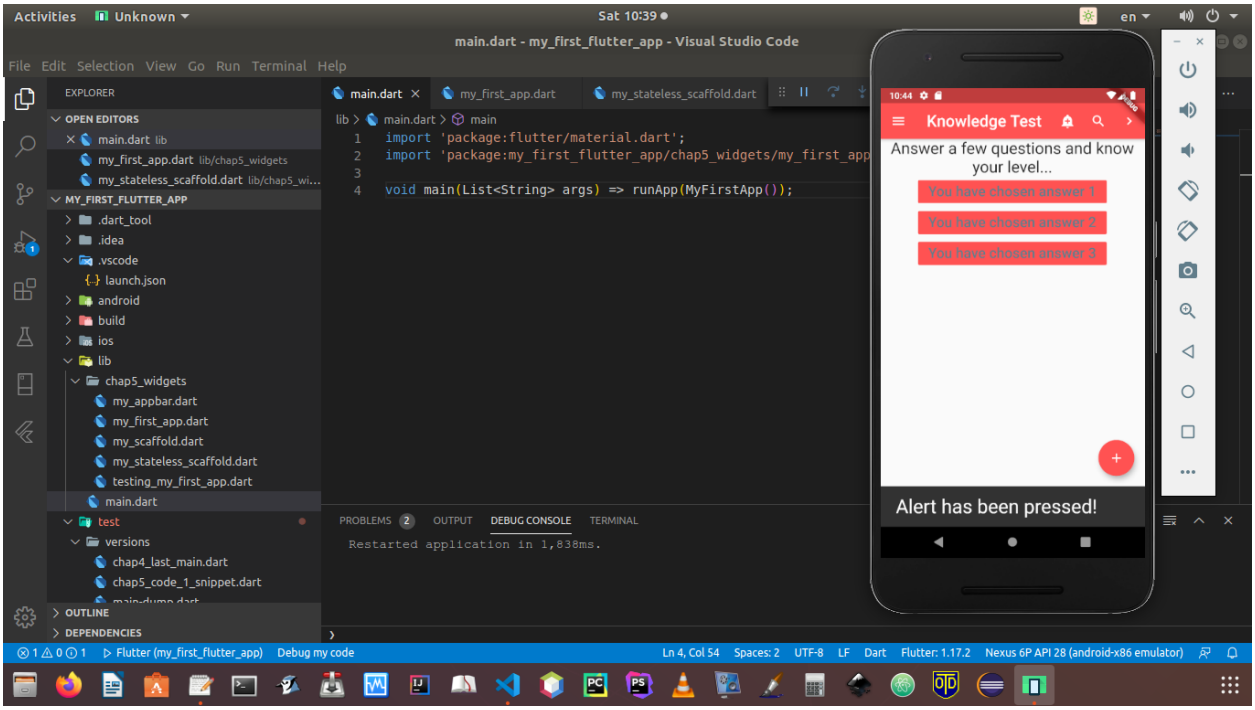


Figure 5.4 – The message 'Alert has been pressed' pops up at the bottom

Pressing the 'alert' icon gives us a message and that takes place due to the change of state. It stays for a few seconds.

Same thing happens, when we press the 'search' icon (Figure 5.5).

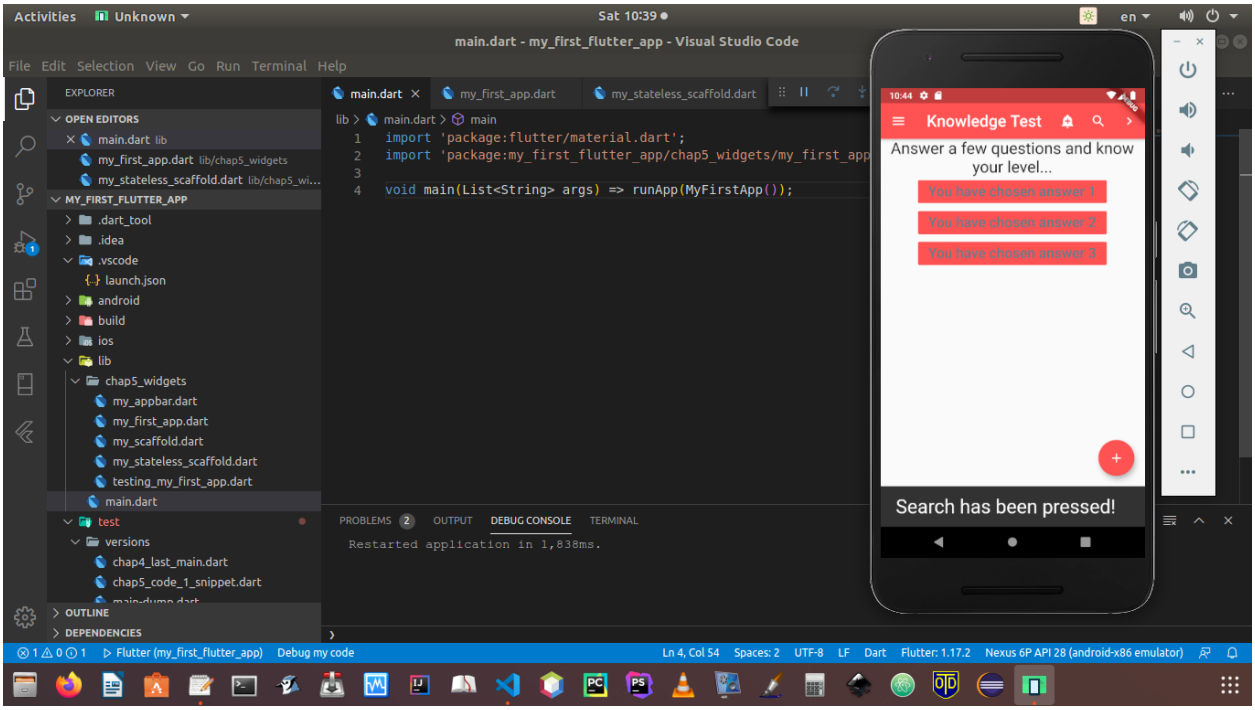


Figure 5.5 – The 'search' icon has been pressed

Now the time has come to go to the next page. We have created a special method for that purpose only.

```
void clickNextPage(BuildContext context) {
Navigator.push(context, MaterialPageRoute(
    builder: (BuildContext context) {
    return Scaffold(
        appBar: AppBar(
        title: const Text('Know Yourself...'),
        ),
        body: const Center(
        child: Text(
            'Dig deep into every layer of your mind to find yourself...',
            style: TextStyle(fontSize: 24),
            textAlign: TextAlign.center,
        ),
        ),
    );
    },
));
}
```

After that, we have called that function or method, in this part of our code:

```
IconButton(
            icon: const Icon(Icons.navigate_next),
            tooltip: 'Next page',
            onPressed: () {
            clickNextPage(context);
            },
        ),
```

Here as a named parameter, ‘onPressed’ passes an anonymous function that calls the ‘clickNextPage(context)’ method. The ‘actions’ widget list has that ‘IconButton’ widget. Clicking the navigation arrow takes us to the next page(Figure 5.6).

We have also defined a simple page inside that function just to get an idea how Flutter widgets manage these UI designs out of the box.

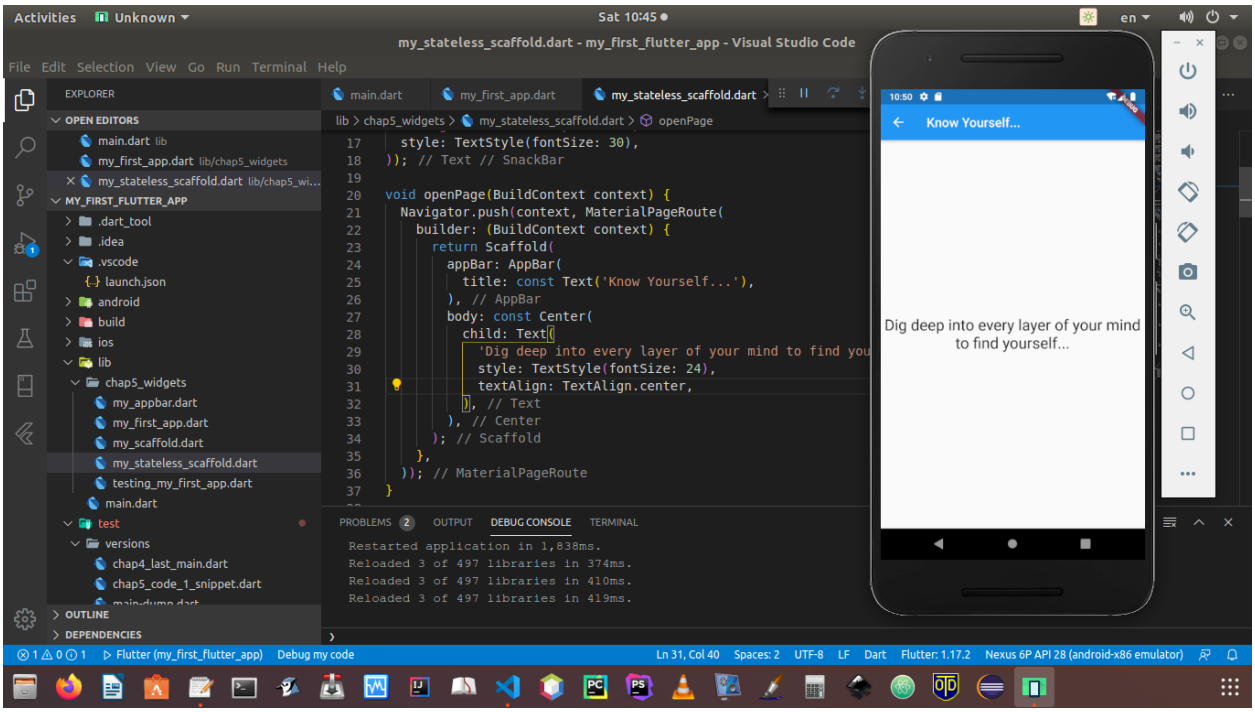


Figure 5.6 – We have navigated to the next page

Now our application becomes more interactive, maintaining the state also.

However, we want to see the Container widget separately, because in the later part of our application, we will use that widget for other purpose.

The next code snippet will take us to a different type of display where we will only show the Container widget.

```dart
//code 5.4
//my_container.dart
import 'package:flutter/material.dart';

class MyContainerWidget extends StatelessWidget {
@override
Widget build(BuildContext context) {
    return Scaffold(
    appBar: AppBar(
        title: Text(
        'Knowledge Test',
        style: TextStyle(
            fontSize: 25.00,
            fontStyle: FontStyle.normal,
        ),
        ),
        backgroundColor: Colors.redAccent,
    ),
    body: Container(
        constraints: BoxConstraints.expand(
        height: Theme.of(context).textTheme.headline4.fontSize * 1.1 + 200.0,
        ),
        padding: const EdgeInsets.all(8.0),
        color: Colors.blue[600],
        alignment: Alignment.center,
        child: Text('This is Container Widget',
            style: Theme.of(context)
                .textTheme
                .headline4
                .copyWith(color: Colors.white)),
        transform: Matrix4.rotationZ(-0.2),
    ),

    floatingActionButton: FloatingActionButton(
        tooltip: 'Add', // we can add more questions later
        backgroundColor: Colors.redAccent,
        child: Icon(Icons.add),
        onPressed: null,
    ),
    );
}
}

```

```
//my_first_app.dart
import 'package:flutter/material.dart';
import 'package:my_first_flutter_app/chap5_widgets/my_container.dart';
//import 'package:my_first_flutter_app/chap5_widgets/my_stateless_scaffold.dart';

class MyFirstApp extends StatelessWidget {

@override
Widget build(BuildContext context) {
    return MaterialApp(
    home: MyContainerWidget()
    );
}
}

//main.dart
import 'package:flutter/material.dart';
import 'package:my_first_flutter_app/chap5_widgets/my_first_app.dart';

void main(List<String> args) => runApp(MyFirstApp());
```

The only new file is 'my_container.dart' where in the 'body' part, we have used the Container widget to get an idea, how we can use the 'matrix'. Watch this part:

```
body: Container(
        constraints: BoxConstraints.expand(
        height: Theme.of(context).textTheme.headline4.fontSize * 1.1 + 200.0,
        ),
        padding: const EdgeInsets.all(8.0),
        color: Colors.blue[600],
        alignment: Alignment.center,
        child: Text('This is Container Widget',
            style: Theme.of(context)
                .textTheme
                .headline4
                .copyWith(color: Colors.white)),
        transform: Matrix4.rotationZ(-0.2),
    ),
```

We can control many things of this Text element. With the Text widget, we cannot do those staff (Figure 5.7). Container widget is extremely handy tool that combines many other widgets, which in turn, controls padding, positioning and sizing.

The most amazing part of the Container class is it transforms the Text in different angles that can be controlled by changing the parameter values. transform: Matrix4.rotationZ(-0.2),

We can also control the background and foreground colors. If we do not use 'children', the Container can be as big as we want to make it look like, as well as we can also make it very small.

Moreover, with children, the Container controls its size according to the size of the children.

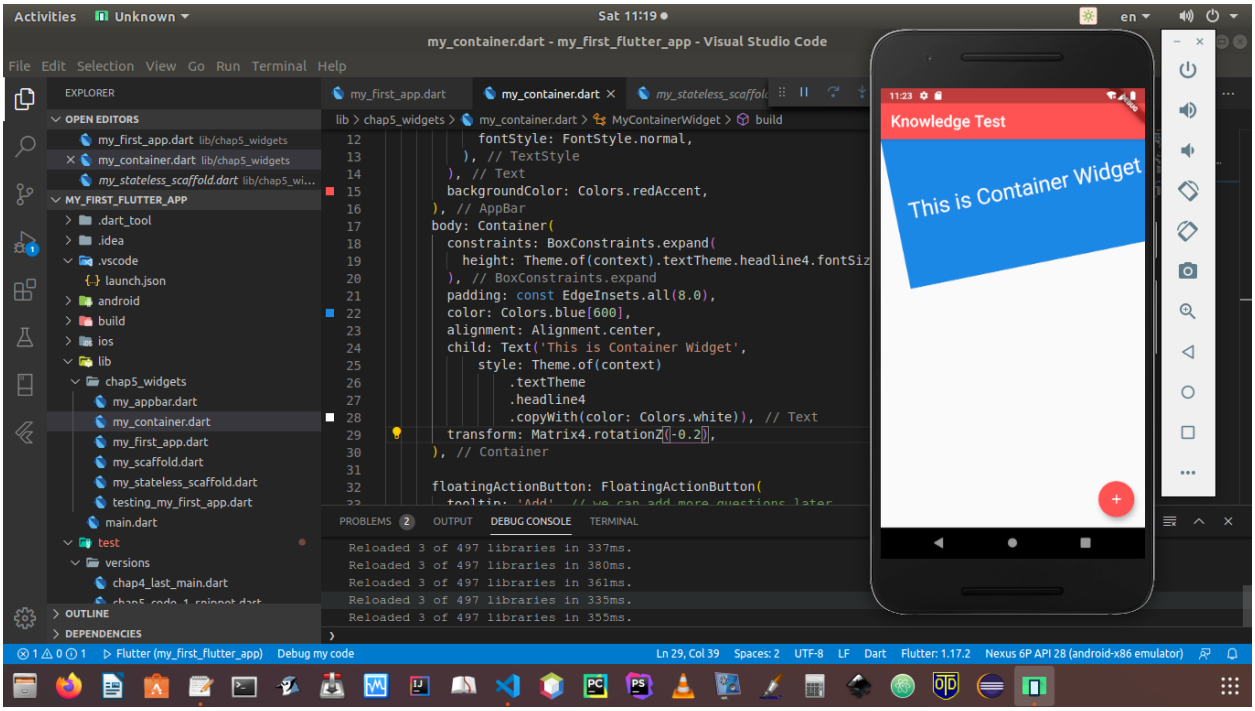


Figure 5.7 – A sample of Container Widget

Just like Container class, the Icon widget or class also plays a major role to build the design of our application UI. By default Icon class is not interactive. If we want to make it interactive we need to use material's 'IconButton' widget.

We will see to that later in our application. Here we again come back to our original Flutter application, the only difference is we have used Icon widget inside the 'RaisedButton' widget.

We will see this code snippet where many things have been repeated. For brevity, we are not going to repeat other snippets, such as 'main.dart', etc. We want to see only the 'MyStatelessScaffoldWidget' class, where we have used the Icon class widget.

```dart
//code 5.5
//my_stateless_scaffold.dart
import 'package:flutter/material.dart';


class MyStatelessScaffoldWidget extends StatelessWidget {
MyStatelessScaffoldWidget({Key key}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Scaffold(
    key: scaffoldKey,
    appBar: AppBar(
        actions: <Widget>[
        IconButton(
            icon: const Icon(Icons.add_alert),
            tooltip: 'Show Snackbar',
            onPressed: () {
            scaffoldKey.currentState.showSnackBar(snackBarOne);
            },
        ),
        IconButton(
            icon: Icon(Icons.search),
            tooltip: 'Search',
            onPressed: () {
            scaffoldKey.currentState.showSnackBar(snackBarTwo);
            },
        ),
        IconButton(
            icon: const Icon(Icons.navigate_next),
            tooltip: 'Next page',
            onPressed: () {
            clickNextPage(context);
            },
        ),
        ],
        leading: IconButton(
        icon: Icon(Icons.menu),
        tooltip: 'Navigation menu',
        onPressed: () {
            scaffoldKey.currentState.showSnackBar(snackBarThree);
        },
        ),
```

```
44          title: Text(
45          'Knowledge Test',
46          style: TextStyle(
47              fontSize: 25.00,
48              fontStyle: FontStyle.normal,
49          ),
50          ),
51          backgroundColor: Colors.redAccent,
52      ),
53      body: Column(
54          children: [
55          Text(
56              'Answer a few questions and know your level...',
57              textAlign: TextAlign.center,
58              style: TextStyle(
59              fontSize: 25,
60              ),
61          ),
62          RaisedButton(
63              child: Text(
64              'You have chosen answer 1',
65              style: TextStyle(
66                  fontSize: 22,
67                  color: Colors.blueGrey,
68              ),
69              ),
70              disabledColor: Colors.redAccent,
71              onPressed: null,
72          ),
73          Icon(
74              Icons.favorite,
75              color: Colors.pink,
76              size: 24.0,
77              semanticLabel: 'Text to announce in accessibility modes',
78          ),
79          RaisedButton(
80              child: Text(
81              'You have chosen answer 2',
82              style: TextStyle(
83                  fontSize: 22,
84                  color: Colors.blueGrey,
85              ),
86              ),
```

```
87              disabledColor: Colors.redAccent,
88              onPressed: null,
89          ),
90          Icon(
91              Icons.audiotrack,
92              color: Colors.green,
93              size: 30.0,
94          ),
95          RaisedButton(
96              child: Text(
97              'You have chosen answer 3',
98              style: TextStyle(
99                  fontSize: 22,
100                 color: Colors.blueGrey,
101             ),
102             ),
103             disabledColor: Colors.redAccent,
104             onPressed: null,
105         ),
106         Icon(
107             Icons.beach_access,
108             color: Colors.blue,
109             size: 36.0,
110         ),
111         ],
112     ),
113     floatingActionButton: FloatingActionButton(
114         tooltip: 'Add', // we can add more questions later
115         backgroundColor: Colors.redAccent,
116         child: Icon(Icons.add),
117         onPressed: null,
118     ),
119     );
120 }
121 }
```

The change in the above code snippet automatically changes the look of our application that we have been building (Figure 5.8).

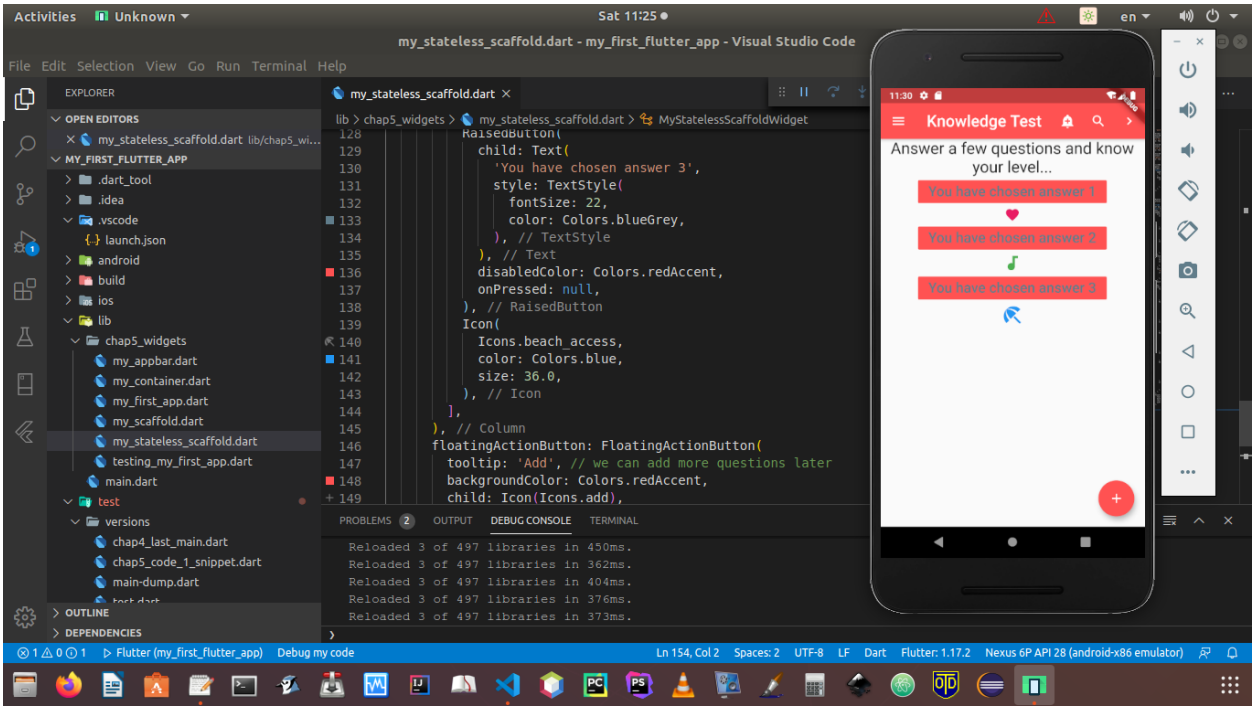


Figure 5.8 – Icon widgets have changed the look of the application

We will see to more widgets later, as we will keep building our Flutter application.

Before closing this chapter, we will take a look at the anonymous or lambda functions of Dart. Besides that, we will also try to understand some core features of object-oriented programming in Dart. It includes, parent-child relationship, extending a class, and many more.

Anonymous Functions: Lambda, Higher Order Functions, and Lexical Closures

Lambda, Higher Order functions, and Lexical Closures have some similarities. In their namelessness and anonymity, these features of Dart are very interesting. Let us start with Lambda. Then we will discuss Higher Order functions and Closures. In reality, you will find that Lambda actually implements Higher-Order Functions. This features have been widely used in our Flutter project.

As the name suggests, Lambda is a nameless function and we can use it in two ways. We can use it in a traditional method and also we can use the 'Fat Arrow'. Consider the first code snippet:

```
//code 5.6
class LambdaCode{
// here addingTwonumbers is a nameless function
Function addingTwonumbers = (int x, int y){
    var sum = x + y;
    return sum;
};
}
main(List<String> arguments){
var lambdaShow = LambdaCode();
print(lambdaShow.addingTwonumbers(12, 47));
}
```

We will give the output after adding the 'Fat Arrow' method. The whole code snippet looks like this:

```
//code 5.7
class LambdaCode{
// here addingTwonumbers is a nameless function
Function addingTwonumbers = (int x, int y){
    var sum = x + y;
    return sum;
};
Function divideByFour = (int num) => num ~/ 4;
}
main(List<String> arguments){
var lambdaShow = LambdaCode();
print(lambdaShow.addingTwonumbers(12, 47));
print(lambdaShow.divideByFour(56));
}
```

The output is quite expected, in the first anonymous function 'Function addingTwonumbers' we have passed two parameters and added them. And using the 'Fat Arrow' method, we have passed a number through another nameless function 'Function divideByFour' and divided it by 4.

```
//output
59
14
```

While building a native iOS or Android app, we have seen how these nameless functions come to your help. Remember the 'onPress' named parameter that points to the anonymous function.

Exploring Higher-Order Functions

The specialty of Higher Order functions is it can accept a function as a parameter. That is why it is named the Higher Order Function. It not only can accept a function as a parameter, it can also return it; actually, it can do both. This concept also has widely used in Flutter tools. We will see a very simple code snippet to get accustomed to the idea.

```
1  //code 5.8
2  //returning a function
3  Function DividingByFour(){
4  Function LetUsDivide = (int x) => x ~/ 4;
5  return LetUsDivide;
6  }
7  main(List<String> arguments){
8  var result = DividingByFour();
9  print(result(56));
10 }
11
12 The output is 14.
```

So we have passed a nameless function ‘Function LetUsDivide’ as a parameter and returned the value of the division through a higher order function ‘Function DividingByFour()’.

Inheritance and Mixins in Dart

One of the key features of object-oriented programming is being able to extend your classes. We extend to create a class and the extended class is known as a subclass. The subclass inherits reference variables and class methods from the parent class, which is known as a superclass. In our Flutter project we have seen a lot extensions. One Widget class extends other class that also extends other, and the process goes on.

Consider this simple example where we have extended an Animal class to a Cat class.

```dart
//code 5.9
class Animal {
String name = "Animal";
Animal(){
    print("I am Animal class constructor.");
}
Animal.namedConstructor(){
    print("This is parent animal named constructor.");
}
void showName(){
    print(this.name);
}
void eat(){
    print("Animals eat everything depending on whay type it is.");
}
}
class Cat extends Animal {
//overriding parent constructor
//although constructors are not inherited
Cat() : super(){
    print("I am child cat class overriding super Animal class.");
}
Cat.namedCatConstructor() : super.namedConstructor(){
    print("The child cat named constructor overrides the parent animal named constru\
ctor.");
}
@override
void showName(){
    print("Hi from cat.");
}
@override
void eat(){
    super.eat();
    print("Cat doesn't eat vegetables..");
}
}
main(List<String> arguments){
var cat = Cat();
cat.name = "Meaow";
cat.showName();
cat.eat();
var anotherCat = Cat.namedCatConstructor();
}
```

Watching these code snippets, automatically helps us to remember one Widget class extends either 'StatelessWidget',or 'StatefulWidget'.

Let us first see the output and after that we will discuss the features of subclass and superclass.

```
//output
I am Animal class constructor.
I am child cat class overriding super Animal class.
Hi from cat.
Animals eat everything depending on what type it is.
Cat doesn't eat vegetables..
This is parent animal named constructor.
The child cat named constructor overrides the parent animal named constructor.
```

The code is quite simple to follow; the superclass 'Animal' has two constructors: default and named constructor. The subclass 'Cat' overrides both the constructors. Superclass constructors are not inherited. Therefore, a subclass always has its own constructor; either default parameterized or named one. However, a subclass can always override the superclass constructors. That is what we have done here.

Mixins: Adding more Features to a Class

Dart has lot to offer when re-usability of classes are needed; there is a very important concept called 'mixins'. It is a way of reusing any class' code in multiple class hierarchies. We have seen the same features while building our Flutter project.

We can rewrite the above code using 'mixins'. All we need to do is use the keyword 'with'. Suppose we have another class 'Dog' that has a method 'canRun()'. A cat object can also run, isn't it? Let us try the same code in a slight different way.

```
//code 5.10
class Animal {
String name = "Animal";
Animal(){
    print("I am Animal class constructor.");
}
Animal.namedConstructor(){
    print("This is parent animal named constructor.");
}
void showName(){
    print(this.name);
}
void eat(){
```

```
    print("Animals eat everything depending on what type it is.");
}
}
class Dog {
void canRun(){
    print("I can run.");
}
}
class Cat extends Animal with Dog {
//overriding parent constructor
//although constructors are not inherited
Cat() : super(){
    print("I am child cat class overriding super Animal class.");
}
Cat.namedCatConstructor() : super.namedConstructor(){
    print("The child cat named constructor overrides the parent animal named constru\
ctor.");
}
@override
void showName(){
    print("Hi from cat.");
}
@override
void eat(){
    super.eat();
    print("Cat doesn't eat vegetables..");
}
}
main(List<String> arguments){
var cat = Cat();
cat.name = "Meaow";
cat.showName();
cat.eat();
var anotherCat = Cat.namedCatConstructor();
anotherCat.canRun();
}
```

The subclass ‘Cat’ has been extended and at the same it has used ‘mixins’ by reusing the ‘Dog’ class’ code. Watch this line:

```
class Cat extends Animal with Dog {…}
```

And in the main() function the ‘Cat’ object uses the ‘Dog’ class’ method this way:

```
anotherCat.canRun();
```

The output has not been changed except the last line:

```
//output
I am Animal class constructor.
I am child cat class overriding super Animal class.
Hi from cat.
Animals eat everything depending on what type it is.
Cat doesn't eat vegetables..
This is parent animal named constructor.
The child cat named constructor overrides the parent animal named constructor.
I can run.
```

Remember, for ‘mixin’ we need to use the ‘with’ keyword followed by one or more ‘mixin’ names. Support for the ‘mixin’ keyword was introduced in Dart 2.1. Before that, in such cases, the abstract class was used.

We will learn how to use abstract class in the coming chapters.

Want to read more Flutter related Articles and resources?

For more Flutter related Articles and Resources[6]

[6] https://zerodotone.net

6. Layouts in Flutter, Tips and Tricks

Any type of UI design is always concerned with layouts. Flutter's layout mechanism, in its core, has nothing but widgets. We have found it earlier, in Flutter, almost everything is widgets.

While designing a layout, we need mainly images, icons, text, etc. These are all visible widgets. However, there are a few invisible widgets, too. These inconspicuous widgets play crucial roles in building a proper layout by placing all the visible widgets in right places. Widgets like Column, Row, Grid, and many others have the quality of not being perceivable by the eye.

We create a layout by using different types of widgets to create more complex widgets.

For example, the first screenshot of this chapter below shows two icons with a label under each one (Figure 6.1).

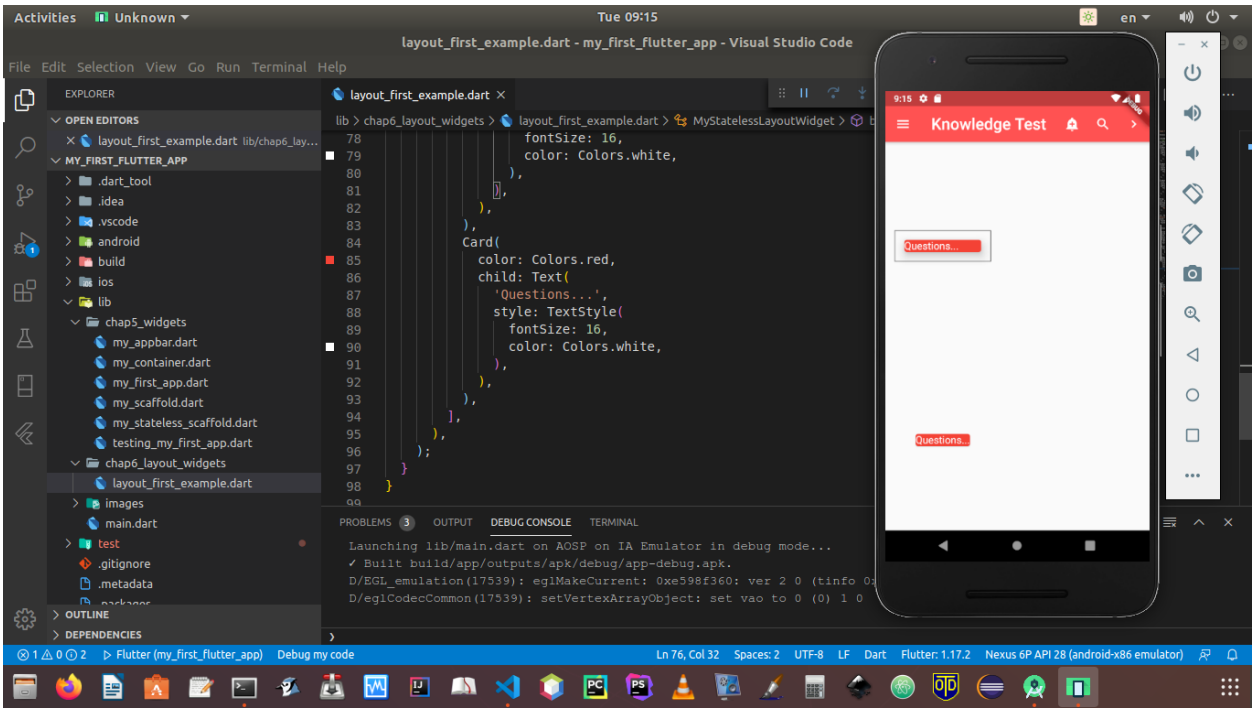


Figure 6.1 – creating a layout by using widgets

Next, the below code snippets show us how we have used column, container and card widgets to create this simple basic layout.

//code 6.1 // layout_first_example.dart

import 'package:flutter/material.dart';

class MyStatelessLayoutWidget extends StatelessWidget { @override Widget build(Object context) {

```
  return Scaffold(
    appBar: AppBar(
      actions: <Widget>[
        IconButton(
          icon: const Icon(Icons.add_alert),
          tooltip: 'Show Snackbar',
          onPressed: () {
            Text(
              'Nothing',
            );
          },
        ),
        IconButton(
          icon: Icon(Icons.search),
          tooltip: 'Search',
          onPressed: () {
            Text(
              'Nothing',
            );
          },
        ),
        IconButton(
          icon: const Icon(Icons.navigate_next),
          tooltip: 'Next page',
          onPressed: () {
            Text(
              'Nothing',
            );
          },
        ),
      ],
      leading: IconButton(
        icon: Icon(Icons.menu),
        tooltip: 'Navigation menu',
        onPressed: () {
          Text(
            'Nothing',
          );
        },
      ),
```

```
41        title: Text(
42          'Knowledge Test',
43          style: TextStyle(
44            fontSize: 25.00,
45            fontStyle: FontStyle.normal,
46          ),
47        ),
48        backgroundColor: Colors.redAccent,
49      ),
50      body: Column(
51        mainAxisAlignment: MainAxisAlignment.spaceAround,
52        children: <Widget>[
53          Container(
54            margin: EdgeInsets.symmetric(
55              vertical: 10,
56              horizontal: 15,
57            ),
58            decoration: BoxDecoration(
59              border: Border.all(
60                color: Colors.black,
61                width: 1,
62              ),
63            ),
64            padding: EdgeInsets.all(10),
65            width: 150,
66            child: Card(
67              elevation: 12,
68              color: Colors.red,
69              child: Text(
70                'Questions...',
71                style: TextStyle(
72                  fontSize: 16,
73                  color: Colors.white,
74                ),
75              ),
76            ),
77          ),
78          Card(
79            color: Colors.red,
80            child: Text(
81              'Questions...',
82              style: TextStyle(
83                fontSize: 16,
```

```
              color: Colors.white,
            ),
          ),
        ),
      ],
    ),
  );
}
```

}

// my_first_app.dart

import ‘package:flutter/material.dart’; import ‘package:my_first_flutter_app/chap6_layout_widgets/layout_first_example.dart’;

class MyFirstApp extends StatelessWidget {

```
@override
Widget build(BuildContext context) {
  return MaterialApp(
    home: MyStatelessLayoutWidget()
  );
}
```

}

As we progress, these code snippets will grow bigger and bigger. We will try to maintain the modularity so that we can understand what is happening under the hood.

In the above code, we will take a look at the ‘body’ section, where the actual layout part is getting built. The ‘body’ named parameter primarily points to column widget.

body: Column(mainAxisAlignment: MainAxisAlignment.spaceAround, children: <Widget>[Container(…

First of all, we can adjust the alignment of the column through another widgets. Then comes the ‘children’, a named parameter, and a widget that holds a list of other widgets. At the very beginning we see container widget.

Inside the container widget, we have a ‘child’ widget ‘Card’.

child: Card(elevation: 12,
color: Colors.red, child: Text(‘Questions…’, style: TextStyle(fontSize: 16, color: Colors.white,),),),

After that, we come out of the container widget, and again use another card widget. We can move forward to see whether, we can create more complex layout, step by step, in this chapter.

The next screenshot shows you how we try to organize the quiz app (Figure 6.2) using other layout widgets.

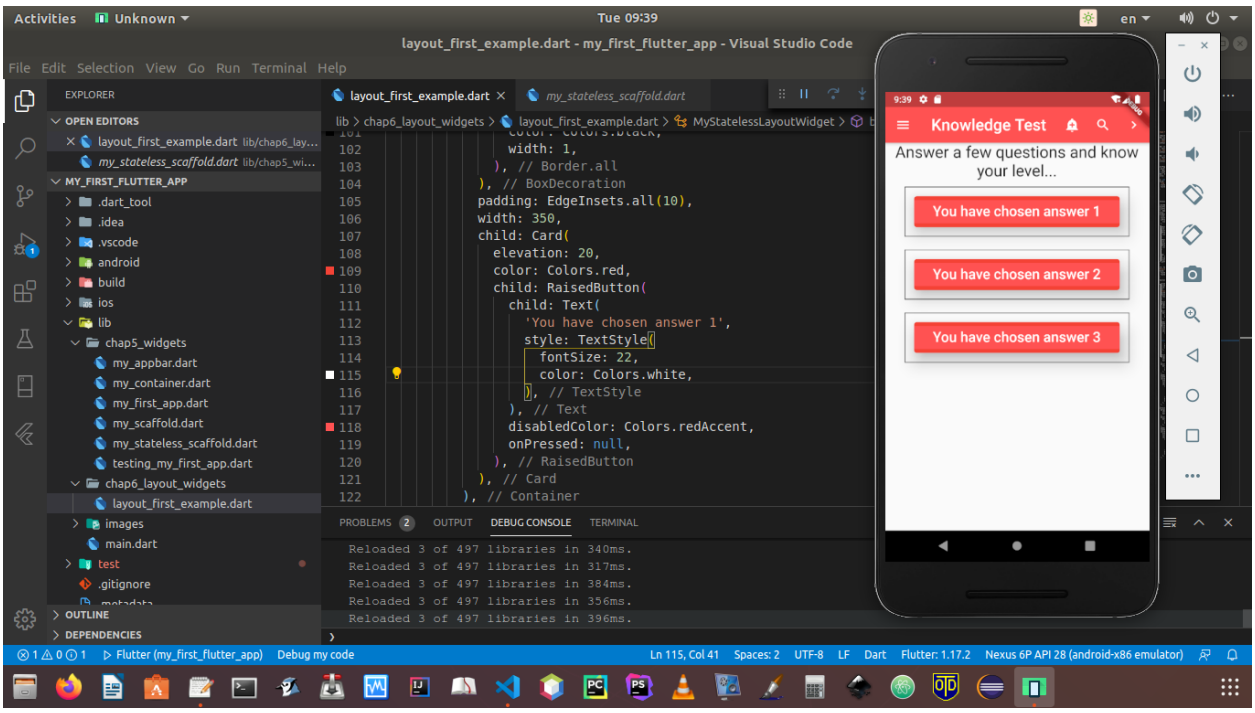


Figure 6.2 – creating complex layout step-by-step

We have also changed our code adding more layout related widgets.

Before we go to the next code snippets, let us take a close look at the Container widget. It is one of the most common and important layout widgets that we will use again and again.

Customize child Widgets

The Container widget class allows you to customize its child widgets. Therefore, it is very handy to create a complex layout using the Container as the root widget of the body sections. There are other advantages of using the Container widget. We can control padding, margins, borders, background colors and many more other capabilities.

```dart
//code 6.2
// layout_first_example.dart

import 'package:flutter/material.dart';

final GlobalKey<ScaffoldState> scaffoldKey = GlobalKey<ScaffoldState>();
final SnackBar snackBarOne = const SnackBar(
    content: Text(
    'Alert has been pressed!',
    style: TextStyle(fontSize: 30),
));
final SnackBar snackBarTwo = const SnackBar(
    content: Text(
    'Search has been pressed!',
    style: TextStyle(fontSize: 30),
));
final SnackBar snackBarThree = const SnackBar(
    content: Text(
    'Navigation has been pressed!',
    style: TextStyle(fontSize: 30),
));

void clickNextPage(BuildContext context) {
    Navigator.push(context, MaterialPageRoute(
    builder: (BuildContext context) {
        return Scaffold(
        appBar: AppBar(
            title: const Text('Know Yourself...'),
        ),
        body: const Center(
            child: Text(
            'Dig deep into every layer of your mind to find yourself...',
            style: TextStyle(fontSize: 24),
            textAlign: TextAlign.center,
            ),
        ),
        );
    },
    ));
}

class MyStatelessLayoutWidget extends StatelessWidget {
    @override
```

```
44      Widget build(Object context) {
45      return Scaffold(
46          key: scaffoldKey,
47          appBar: AppBar(
48          actions: <Widget>[
49              IconButton(
50              icon: const Icon(Icons.add_alert),
51              tooltip: 'Show Snackbar',
52              onPressed: () {
53                  scaffoldKey.currentState.showSnackBar(snackBarOne);
54              },
55              ),
56              IconButton(
57              icon: Icon(Icons.search),
58              tooltip: 'Search',
59              onPressed: () {
60                  scaffoldKey.currentState.showSnackBar(snackBarTwo);
61              },
62              ),
63              IconButton(
64              icon: const Icon(Icons.navigate_next),
65              tooltip: 'Next page',
66              onPressed: () {
67                  clickNextPage(context);
68              },
69              ),
70          ],
71          leading: IconButton(
72              icon: Icon(Icons.menu),
73              tooltip: 'Navigation menu',
74              onPressed: () {
75              scaffoldKey.currentState.showSnackBar(snackBarThree);
76              },
77          ),
78          title: Text(
79              'Knowledge Test',
80              style: TextStyle(
81              fontSize: 25.00,
82              fontStyle: FontStyle.normal,
83              ),
84          ),
85          backgroundColor: Colors.redAccent,
86          ),
```

```
87          body: Column(
88          mainAxisAlignment: MainAxisAlignment.start,
89          children: <Widget>[
90              Text(
91              'Answer a few questions and know your level...',
92              textAlign: TextAlign.center,
93              style: TextStyle(
94                  fontSize: 25,
95              ),
96              ),
97              Container(
98              margin: EdgeInsets.symmetric(
99                  vertical: 10,
100                 horizontal: 15,
101             ),
102             decoration: BoxDecoration(
103                 border: Border.all(
104                 color: Colors.black,
105                 width: 1,
106                 ),
107             ),
108             padding: EdgeInsets.all(10),
109             width: 350,
110             child: Card(
111                 elevation: 20,
112                 color: Colors.red,
113                 child: RaisedButton(
114                 child: Text(
115                     'You have chosen answer 1',
116                     style: TextStyle(
117                     fontSize: 22,
118                     color: Colors.white,
119                     ),
120                 ),
121                 disabledColor: Colors.redAccent,
122                 onPressed: null,
123                 ),
124             ),
125             ),
126             Container(
127             margin: EdgeInsets.symmetric(
128                 vertical: 10,
129                 horizontal: 15,
```

```
130             ),
131             decoration: BoxDecoration(
132                 border: Border.all(
133                 color: Colors.black,
134                 width: 1,
135                 ),
136             ),
137             padding: EdgeInsets.all(10),
138             width: 350,
139             child: Card(
140                 elevation: 20,
141                 color: Colors.red,
142                 child: RaisedButton(
143                 child: Text(
144                     'You have chosen answer 2',
145                     style: TextStyle(
146                     fontSize: 22,
147                     color: Colors.white,
148                     ),
149                 ),
150                 disabledColor: Colors.redAccent,
151                 onPressed: null,
152                 ),
153             ),
154             ),
155             Container(
156             margin: EdgeInsets.symmetric(
157                 vertical: 10,
158                 horizontal: 15,
159             ),
160             decoration: BoxDecoration(
161                 border: Border.all(
162                 color: Colors.black,
163                 width: 1,
164                 ),
165             ),
166             padding: EdgeInsets.all(10),
167             width: 350,
168             child: Card(
169                 elevation: 20,
170                 color: Colors.red,
171                 child: RaisedButton(
172                 child: Text(
```

```
                    'You have chosen answer 3',
                    style: TextStyle(
                    fontSize: 22,
                    color: Colors.white,
                    ),
                ),
                disabledColor: Colors.redAccent,
                onPressed: null,
                ),
            ),
            ),
        ],
        ),
    );
    }
}
```

Let us examine the the Container part especially, where we have added such capabilities that we have just discussed.

We take one instance of Container class widget. We have used three to get the display.

Container(margin: EdgeInsets.symmetric(vertical: 10, horizontal: 15,), decoration: BoxDecoration(border: Border.all(color: Colors.black, width: 1,),), padding: EdgeInsets.all(10), width: 350, child: Card(elevation: 20, color: Colors.red, child: RaisedButton(child: Text('You have chosen answer 3', style: TextStyle(fontSize: 22, color: Colors.white,),), disabledColor: Colors.redAccent, onPressed: null,),),),

We have added margins, padding, width, etc. Inside the Container class widget we have used different types of 'child' widgets, such as Card. Inside Card class widget we have used another 'child' widget 'RaisedButton'.

Layout mechanism of Flutter

Flutter's layout mechanism is controlled by the widgets. In the above example, each Text widget is placed in a RaisedButton widget, which is inside the Card widget, and the Card widget is again placed inside the Container widget.

While adding these widget trees, we have controlled many layout capabilities, such as the padding, margins, width, etc. And by this way, we have designed our UI.

Let us modify our code so that we could make our UI more interactive with the help of 'SnackBar' and 'ScaffoldState' widget classes. We have also added an extra Dart file 'questions.dart', where we have defined a 'Questions' class and through the constructor passed a named parameter.

```dart
//code 6.3
// layout_first_example.dart

import 'package:flutter/material.dart';
import 'package:my_first_flutter_app/chap6_layout_widgets/questions.dart';

final GlobalKey<ScaffoldState> scaffoldKey = GlobalKey<ScaffoldState>();
final SnackBar snackBarOne = const SnackBar(
    content: Text(
    'Alert has been pressed!',
    style: TextStyle(fontSize: 30),
));
final SnackBar snackBarTwo = const SnackBar(
    content: Text(
    'Search has been pressed!',
    style: TextStyle(fontSize: 30),
));
final SnackBar snackBarThree = const SnackBar(
    content: Text(
    'Navigation has been pressed!',
    style: TextStyle(fontSize: 30),
));

void clickNextPage(BuildContext context) {
    Navigator.push(context, MaterialPageRoute(
    builder: (BuildContext context) {
        return Scaffold(
        appBar: AppBar(
            title: const Text('Know Yourself...'),
        ),
        body: const Center(
            child: Text(
            'Dig deep into every layer of your mind to find yourself...',
            style: TextStyle(fontSize: 24),
            textAlign: TextAlign.center,
            ),
        ),
        );
    },
    ));
}

class MyStatelessLayoutWidget extends StatelessWidget {
```

```
44      final questions = [
45      Questions(questions: 'Are you impulsive?'),
46      Questions(questions: 'Do you get angry easily?'),
47      Questions(questions: 'Are you sloth?'),
48      Questions(questions: 'Do you cheat others?'),
49      ];
50      @override
51      Widget build(Object context) {
52      return Scaffold(
53          key: scaffoldKey,
54          appBar: AppBar(
55          actions: <Widget>[
56              IconButton(
57              icon: const Icon(Icons.add_alert),
58              tooltip: 'Show Snackbar',
59              onPressed: () {
60                  scaffoldKey.currentState.showSnackBar(snackBarOne);
61              },
62              ),
63              IconButton(
64              icon: Icon(Icons.search),
65              tooltip: 'Search',
66              onPressed: () {
67                  scaffoldKey.currentState.showSnackBar(snackBarTwo);
68              },
69              ),
70              IconButton(
71              icon: const Icon(Icons.navigate_next),
72              tooltip: 'Next page',
73              onPressed: () {
74                  clickNextPage(context);
75              },
76              ),
77          ],
78          leading: IconButton(
79              icon: Icon(Icons.menu),
80              tooltip: 'Navigation menu',
81              onPressed: () {
82              scaffoldKey.currentState.showSnackBar(snackBarThree);
83              },
84          ),
85          title: Text(
86              'Knowledge Test',
```

```
87              style: TextStyle(
88              fontSize: 25.00,
89              fontStyle: FontStyle.normal,
90              ),
91          ),
92          backgroundColor: Colors.redAccent,
93          ),
94          body: Column(
95          mainAxisAlignment: MainAxisAlignment.start,
96          children: <Widget>[
97              Text(
98              '${questions[0].questions}',
99              textAlign: TextAlign.center,
100             style: TextStyle(
101                 fontSize: 25,
102             ),
103             ),
104             Container(
105             margin: EdgeInsets.symmetric(
106                 vertical: 10,
107                 horizontal: 15,
108             ),
109             decoration: BoxDecoration(
110                 border: Border.all(
111                 color: Colors.black,
112                 width: 1,
113                 ),
114             ),
115             padding: EdgeInsets.all(10),
116             width: 350,
117             child: Card(
118                 elevation: 20,
119                 color: Colors.red,
120                 child: RaisedButton(
121                 child: Text(
122                     'No. Not at all...',
123                     style: TextStyle(
124                     fontSize: 22,
125                     color: Colors.white,
126                     ),
127                 ),
128                 disabledColor: Colors.redAccent,
129                 onPressed: null,
```

```
130                 ),
131             ),
132             ),
133             Container(
134             margin: EdgeInsets.symmetric(
135                 vertical: 10,
136                 horizontal: 15,
137             ),
138             decoration: BoxDecoration(
139                 border: Border.all(
140                 color: Colors.black,
141                 width: 1,
142                 ),
143             ),
144             padding: EdgeInsets.all(10),
145             width: 350,
146             child: Card(
147                 elevation: 20,
148                 color: Colors.red,
149                 child: RaisedButton(
150                 child: Text(
151                     'I try to control it...',
152                     style: TextStyle(
153                     fontSize: 22,
154                     color: Colors.white,
155                     ),
156                 ),
157                 disabledColor: Colors.redAccent,
158                 onPressed: null,
159                 ),
160             ),
161             ),
162             Container(
163             margin: EdgeInsets.symmetric(
164                 vertical: 10,
165                 horizontal: 15,
166             ),
167             decoration: BoxDecoration(
168                 border: Border.all(
169                 color: Colors.black,
170                 width: 1,
171                 ),
172             ),
```

```
            padding: EdgeInsets.all(10),
            width: 350,
            child: Card(
                elevation: 20,
                color: Colors.red,
                child: RaisedButton(
                child: Text(
                    'I am very impulsive.',
                    style: TextStyle(
                    fontSize: 22,
                    color: Colors.white,
                    ),
                ),
                disabledColor: Colors.redAccent,
                onPressed: null,
                ),
            ),
            ),
        ],
        ),
    );
    }
}

//questions.dart

class Questions{
    final String questions;
    Questions({this.questions});
}
```

The next screenshot shows you how our quiz application looks like.

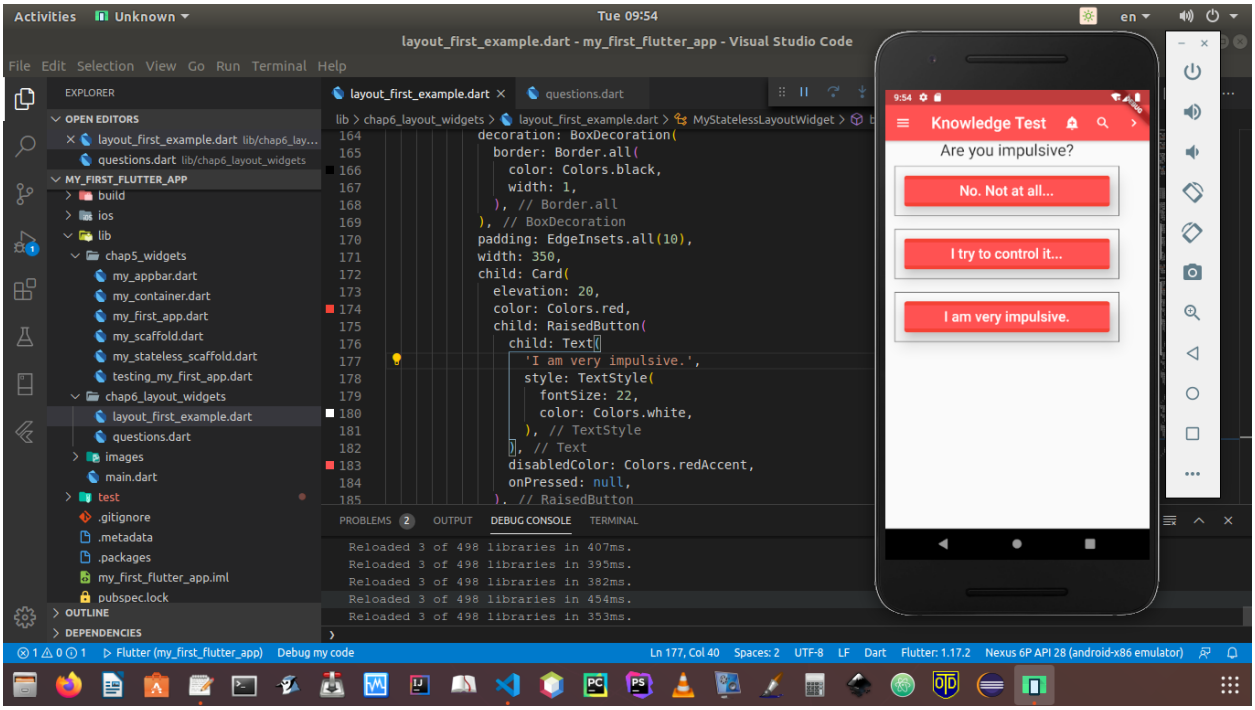


Figure 6.3 – Taking the application to the next level

What happens if we click the buttons? It is now interactive, so at the bottom, it will keep displaying the answers. We are testing our knowledge about ourselves. Therefore, it will give the correct output (Figure 6.4).

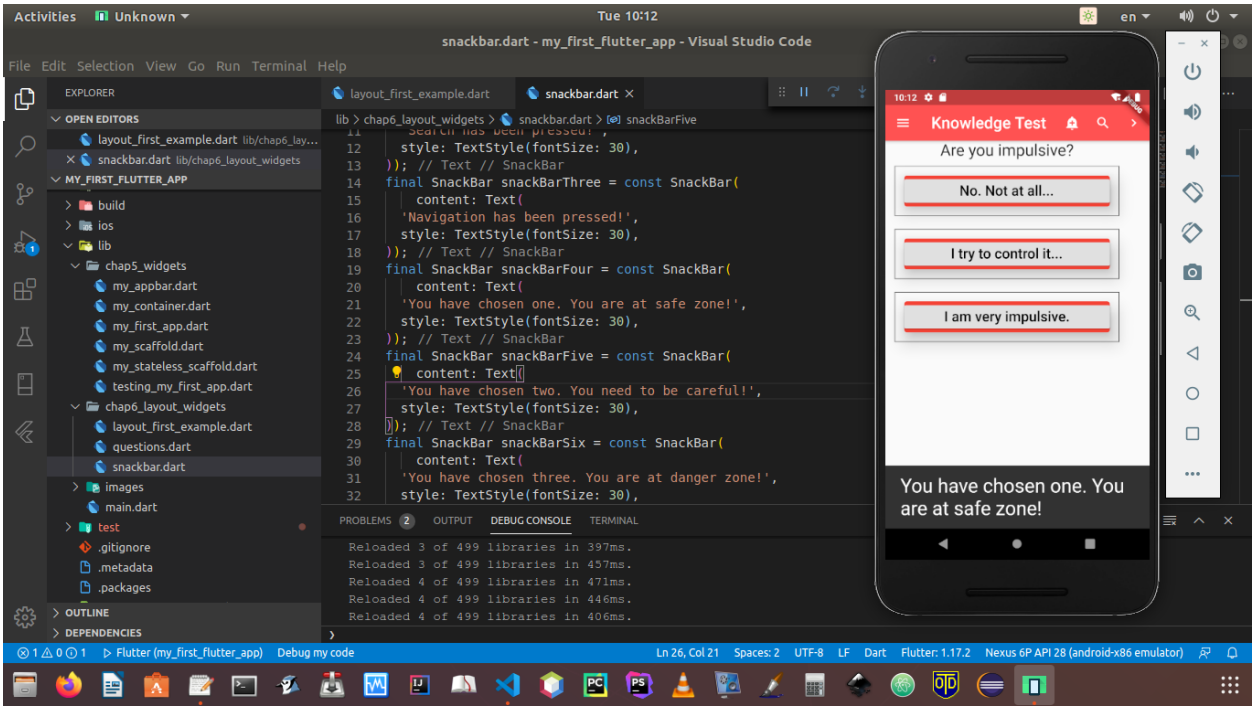


Figure 6.4 – clicking the buttons gives us the related answers

Now we can add more modularity to our code snippets by taking the 'SnackBar' class widget to a different file and import it to our layout first example code, like this.

```
//code 6.4
//snackbar.dart

import 'package:flutter/material.dart';

final SnackBar snackBarOne = const SnackBar(
    content: Text(
    'Alert has been pressed!',
    style: TextStyle(fontSize: 30),
));
final SnackBar snackBarTwo = const SnackBar(
    content: Text(
    'Search has been pressed!',
    style: TextStyle(fontSize: 30),
));
final SnackBar snackBarThree = const SnackBar(
    content: Text(
    'Navigation has been pressed!',
    style: TextStyle(fontSize: 30),
));
```

```
final SnackBar snackBarFour = const SnackBar(
    content: Text(
    'You have chosen one. You are at safe zone!',
    style: TextStyle(fontSize: 30),
));
final SnackBar snackBarFive = const SnackBar(
    content: Text(
    'You have chosen two. You need to be careful!',
    style: TextStyle(fontSize: 30),
));
final SnackBar snackBarSix = const SnackBar(
    content: Text(
    'You have chosen three. You are at danger zone!',
    style: TextStyle(fontSize: 30),
));
//// layout_first_example.dart

import 'package:flutter/material.dart';
import 'package:my_first_flutter_app/chap6_layout_widgets/questions.dart';
import 'package:my_first_flutter_app/chap6_layout_widgets/snackbar.dart';

final GlobalKey<ScaffoldState> scaffoldKey = GlobalKey<ScaffoldState>();
…
```

The code snippets have been shortened for brevity. However, in the next code snippets, we will use the full code so that we can go the next question.

```
//code 6.5
// layout_first_example.dart

import 'package:flutter/material.dart';
import 'package:my_first_flutter_app/chap6_layout_widgets/next_page.dart'
    as next_page;
import 'package:my_first_flutter_app/chap6_layout_widgets/questions.dart';
import 'package:my_first_flutter_app/chap6_layout_widgets/snackbar.dart';

final GlobalKey<ScaffoldState> scaffoldKey = GlobalKey<ScaffoldState>();

class MyStatelessLayoutWidget extends StatelessWidget {
    final questions = [
    Questions(questions: 'Are you impulsive?'),
    Questions(questions: 'Do you get angry easily?'),
    Questions(questions: 'Are you sloth?'),
```

```
17      Questions(questions: 'Do you cheat others?'),
18      ];
19      @override
20      Widget build(Object context) {
21      return Scaffold(
22          key: scaffoldKey,
23          appBar: AppBar(
24          actions: <Widget>[
25              IconButton(
26              icon: const Icon(Icons.add_alert),
27              tooltip: 'Show Snackbar',
28              onPressed: () {
29                  scaffoldKey.currentState.showSnackBar(snackBarOne);
30              },
31              ),
32              IconButton(
33              icon: Icon(Icons.search),
34              tooltip: 'Search',
35              onPressed: () {
36                  scaffoldKey.currentState.showSnackBar(snackBarTwo);
37              },
38              ),
39              IconButton(
40              icon: const Icon(Icons.navigate_next),
41              tooltip: 'Next page',
42              onPressed: () {
43                  next_page.clickNextPage(context);
44              },
45              ),
46          ],
47          leading: IconButton(
48              icon: Icon(Icons.menu),
49              tooltip: 'Navigation menu',
50              onPressed: () {
51              scaffoldKey.currentState.showSnackBar(snackBarThree);
52              },
53          ),
54          title: Text(
55              'Knowledge Test',
56              style: TextStyle(
57              fontSize: 25.00,
58              fontStyle: FontStyle.normal,
59              ),
```

```
60          ),
61          backgroundColor: Colors.redAccent,
62          ),
63          body: Column(
64          mainAxisAlignment: MainAxisAlignment.start,
65          children: <Widget>[
66              Text(
67              '${questions[0].questions}',
68              textAlign: TextAlign.center,
69              style: TextStyle(
70                  fontSize: 25,
71              ),
72              ),
73              Container(
74              margin: EdgeInsets.symmetric(
75                  vertical: 10,
76                  horizontal: 15,
77              ),
78              decoration: BoxDecoration(
79                  border: Border.all(
80                  color: Colors.black,
81                  width: 1,
82                  ),
83              ),
84              padding: EdgeInsets.all(10),
85              width: 350,
86              child: Card(
87                  elevation: 20,
88                  color: Colors.red,
89                  child: RaisedButton(
90                  child: Text(
91                      'No. Not at all...',
92                      style: TextStyle(
93                      fontSize: 22,
94                      color: Colors.black,
95                      ),
96                  ),
97                  disabledColor: Colors.redAccent,
98                  onPressed: () {
99                      scaffoldKey.currentState.showSnackBar(snackBarFour);
100                 },
101                 ),
102             ),
```

```
103             ),
104             Container(
105             margin: EdgeInsets.symmetric(
106                 vertical: 10,
107                 horizontal: 15,
108             ),
109             decoration: BoxDecoration(
110                 border: Border.all(
111                 color: Colors.black,
112                 width: 1,
113                 ),
114             ),
115             padding: EdgeInsets.all(10),
116             width: 350,
117             child: Card(
118                 elevation: 20,
119                 color: Colors.red,
120                 child: RaisedButton(
121                 child: Text(
122                     'I try to control it...',
123                     style: TextStyle(
124                     fontSize: 22,
125                     color: Colors.black,
126                     ),
127                 ),
128                 disabledColor: Colors.redAccent,
129                 onPressed: () {
130                     scaffoldKey.currentState.showSnackBar(snackBarFive);
131                 },
132                 ),
133             ),
134             ),
135             Container(
136             margin: EdgeInsets.symmetric(
137                 vertical: 10,
138                 horizontal: 15,
139             ),
140             decoration: BoxDecoration(
141                 border: Border.all(
142                 color: Colors.black,
143                 width: 1,
144                 ),
145             ),
```

```
146             padding: EdgeInsets.all(10),
147             width: 350,
148             child: Card(
149                 elevation: 20,
150                 color: Colors.red,
151                 child: RaisedButton(
152                 child: Text(
153                     'I am very impulsive.',
154                     style: TextStyle(
155                     fontSize: 22,
156                     color: Colors.black,
157                     ),
158                 ),
159                 disabledColor: Colors.redAccent,
160                 onPressed: () {
161                     scaffoldKey.currentState.showSnackBar(snackBarSix);
162                 },
163                 ),
164             ),
165             ),
166             RaisedButton(
167             child: Text(
168                 'Next Question',
169                 style: TextStyle(
170                 fontSize: 22,
171                 color: Colors.blueGrey,
172                 ),
173             ),
174             onPressed: () {
175                 next_page.clickNextQuestion(context);
176             },
177             ),
178             IconButton(
179             icon: const Icon(Icons.navigate_next),
180             tooltip: 'Next Question',
181             onPressed: () {
182                 next_page.clickNextQuestion(context);
183             },
184             ),
185         ],
186         ),
187     );
188     }
```

```
}


//next_page.dart

import 'package:flutter/material.dart';
import 'package:my_first_flutter_app/chap5_widgets/my_stateless_scaffold.dart';
import 'package:my_first_flutter_app/chap6_layout_widgets/questions.dart';
import 'package:my_first_flutter_app/chap6_layout_widgets/snackbar.dart';

void clickNextPage(BuildContext context) {
    Navigator.push(context, MaterialPageRoute(
    builder: (BuildContext context) {
        return Scaffold(
        appBar: AppBar(
            title: const Text('Know Yourself...'),
        ),
        body: const Center(
            child: Text(
            'Dig deep into every layer of your mind to find yourself...',
            style: TextStyle(fontSize: 24),
            textAlign: TextAlign.center,
            ),
        ),
        );
    },
    ));
}

void clickNextQuestion(BuildContext context) {
    Navigator.push(context, MaterialPageRoute(
    builder: (BuildContext context) {

        final questions = [
        Questions(questions: 'Are you impulsive?'),
        Questions(questions: 'Do you get angry easily?'),
        Questions(questions: 'Are you sloth?'),
        Questions(questions: 'Do you cheat others?'),
        ];

        return Scaffold(
        key: scaffoldKey,
        appBar: AppBar(
```

```
232             title: const Text('Know Yourself...'),
233         ),
234         body: Column(
235             mainAxisAlignment: MainAxisAlignment.start,
236             children: <Widget>[
237             Text(
238                 '${questions[1].questions}',
239                 textAlign: TextAlign.center,
240                 style: TextStyle(
241                 fontSize: 25,
242                 ),
243             ),
244             Container(
245                 margin: EdgeInsets.symmetric(
246                 vertical: 10,
247                 horizontal: 15,
248                 ),
249                 decoration: BoxDecoration(
250                 border: Border.all(
251                     color: Colors.black,
252                     width: 1,
253                 ),
254                 ),
255                 padding: EdgeInsets.all(10),
256                 width: 350,
257                 child: Card(
258                 elevation: 20,
259                 color: Colors.red,
260                 child: RaisedButton(
261                     child: Text(
262                     'No. Not at all...',
263                     style: TextStyle(
264                         fontSize: 22,
265                         color: Colors.black,
266                     ),
267                     ),
268                     disabledColor: Colors.redAccent,
269                     onPressed: () {
270                     scaffoldKey.currentState.showSnackBar(snackBarFour);
271                     },
272                 ),
273                 ),
274             ),
```

```
275             Container(
276                 margin: EdgeInsets.symmetric(
277                 vertical: 10,
278                 horizontal: 15,
279                 ),
280                 decoration: BoxDecoration(
281                 border: Border.all(
282                     color: Colors.black,
283                     width: 1,
284                 ),
285                 ),
286                 padding: EdgeInsets.all(10),
287                 width: 350,
288                 child: Card(
289                 elevation: 20,
290                 color: Colors.red,
291                 child: RaisedButton(
292                     child: Text(
293                     'I try to control it...',
294                     style: TextStyle(
295                         fontSize: 22,
296                         color: Colors.black,
297                     ),
298                     ),
299                     disabledColor: Colors.redAccent,
300                     onPressed: () {
301                     scaffoldKey.currentState.showSnackBar(snackBarFive);
302                     },
303                 ),
304                 ),
305             ),
306             Container(
307                 margin: EdgeInsets.symmetric(
308                 vertical: 10,
309                 horizontal: 15,
310                 ),
311                 decoration: BoxDecoration(
312                 border: Border.all(
313                     color: Colors.black,
314                     width: 1,
315                 ),
316                 ),
317                 padding: EdgeInsets.all(10),
```

```
318                 width: 350,
319                 child: Card(
320                 elevation: 20,
321                 color: Colors.red,
322                 child: RaisedButton(
323                     child: Text(
324                     'I cannot control it.',
325                     style: TextStyle(
326                         fontSize: 22,
327                         color: Colors.black,
328                     ),
329                     ),
330                     disabledColor: Colors.redAccent,
331                     onPressed: () {
332                     scaffoldKey.currentState.showSnackBar(snackBarSix);
333                     },
334                 ),
335                 ),
336             ),
337             RaisedButton(
338                 child: Text(
339                 'Next Question',
340                 style: TextStyle(
341                     fontSize: 22,
342                     color: Colors.blueGrey,
343                 ),
344                 ),
345                 onPressed: () {
346                 clickNextQuestion(context);
347                 },
348             ),
349             IconButton(
350                 icon: const Icon(Icons.navigate_next),
351                 tooltip: 'Next Question',
352                 onPressed: () {
353                 clickNextQuestion(context);
354                 },
355             ),
356             ],
357         ),
358         );
359     },
360     ));
```

```
361 }
```

To go to the next question, we need these widgets. We could have used any one of them – either RaisedButton or Icon. We have used both, to get an idea how it works.

RaisedButton(child: Text('Next Question', style: TextStyle(fontSize: 22, color: Colors.blueGrey,),), onPressed: () { next_page.clickNextQuestion(context); },), IconButton(icon: const Icon(Icons.navigate_-next), tooltip: 'Next Question', onPressed: () { next_page.clickNextQuestion(context); },),

Now, after answering the first question, if we press the 'Next Question' button, or press the arrow Icon below, we reach to the next page, where we get the second question. In the 'next_page.dart' file this part of the 'body' widget is important.

final questions = [Questions(questions: 'Are you impulsive?'), Questions(questions: 'Do you get angry easily?'), Questions(questions: 'Are you sloth?'), Questions(questions: 'Do you cheat others?'),];

```
    return Scaffold(
      key: scaffoldKey,
      appBar: AppBar(
        title: const Text('Know Yourself...'),
      ),
      body: Column(
        mainAxisAlignment: MainAxisAlignment.start,
        children: <Widget>[
          Text(
            '${questions[1].questions}',
            textAlign: TextAlign.center,
            style: TextStyle(
              fontSize: 25,
            ),
          ),
```

...

The code is incomplete for brevity. However, we have got the second question for these two lines:

Questions(questions: 'Do you get angry easily?'), '${questions[1].questions}',

Any list index starts with 0. We have tried to get the second element, that is why we have passed the index number 1.

Incidentally, it takes us to the next question, in the next page. The layout mechanism of the next page, or the second question is a little bit simple. We have not worked on the AppBar class widget, this time.

Because it gives us an idea, we have skipped that part. While practicing, and modifying the code, we can of course add more layout capabilities.

The next screenshot will show us how it looks like.

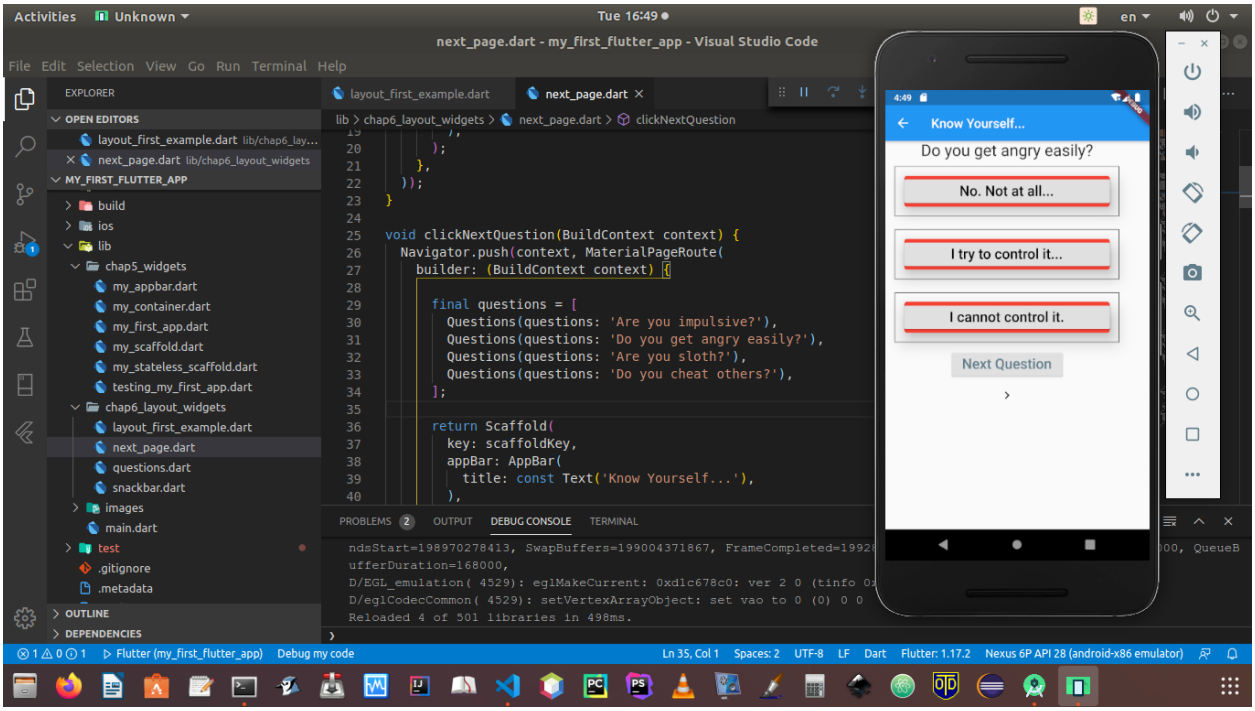


Figure 6.6 – Reaching to the second question

In the next code snippets, we are going to use another widget, Card class widget.

Library of layout widgets

There is a rich library of layout widgets in Flutter. A few of them are commonly used, such as Container, Card, Stack, GridView, ListView, etc.

In Flutter's official documentation the Widget catalog we will get the complete list. There are two categories of layout widgets. We can get the standard widgets from the widgets library, and the specialized widgets from the Material library. One of the most commonly used is Container class widget, where besides using padding, margins, borders or width, we can change the device's background by changing the background color or image. Because we are going to use the Stack class widget in the next code snippets, we will try to understand how this layout widget functions.

When we use Stack widget, the first one is base widget. It could be an image, or any type of colored shapes, where we can use icons, etc. The second one always overlaps the first one. The third one overlaps the second one. Things go on like this, just like a Stack.

The Stack is a list of children of which the first one is the base widget. We keep on adding the subsequent widgets as children on top of the base widget.

Let us see the final code snippets of this chapter. It will give us an idea of how we can manipulate

the layout mechanism with different types of layout widgets. Let us first see the full code first, after that we will see the screenshot of the UI that has been produced by this code snippets.

//code 6.6 //my_new_layout.dart

import 'package:flutter/material.dart'; import 'package:my_first_flutter_app/chap6_layout_widgets/snackbar_and_page.dart';

class MyNewLayout extends StatelessWidget { MyNewLayout({Key key}) : super(key: key);

```
Widget build(BuildContext context) {
  return Scaffold(
    key: scaffoldKey,
    appBar: AppBar(
      actions: <Widget>[
        IconButton(
          icon: const Icon(Icons.add_alert),
          tooltip: 'Show Snackbar',
          onPressed: () {
            scaffoldKey.currentState.showSnackBar(snackBarOne);
          },
        ),
        IconButton(
          icon: Icon(Icons.search),
          tooltip: 'Search',
          onPressed: () {
            scaffoldKey.currentState.showSnackBar(snackBarTwo);
          },
        ),
        IconButton(
          icon: const Icon(Icons.navigate_next),
          tooltip: 'Next page',
          onPressed: () {
            clickNextPage(context);
          },
        ),
      ],
      leading: IconButton(
        icon: Icon(Icons.menu),
        tooltip: 'Navigation menu',
        onPressed: () {
          scaffoldKey.currentState.showSnackBar(snackBarThree);
        },
      ),
      title: Text(
```

```
36          'War Quiz App',
37          style: TextStyle(
38            fontSize: 25.00,
39            fontStyle: FontStyle.normal,
40          ),
41        ),
42        backgroundColor: Colors.redAccent,
43      ),
44      body: Center(
45        child: Column(
46          mainAxisAlignment: MainAxisAlignment.start,
47          children: <Widget>[
48            Stack(
49              alignment: Alignment.topCenter,
50              children: <Widget>[
51                Container(
52                  margin: EdgeInsets.only(top: 25.00),
53                  height: 60,
54                  width: 60,
55                  decoration: BoxDecoration(
56                    borderRadius: BorderRadius.circular(100.00),
57                    color: Colors.redAccent,
58                  ),
59                  child: Icon(Icons.landscape, color: Colors.brown),
60                ),
61                Container(
62                  margin: EdgeInsets.only(top: 70.00, right: 50.00),
63                  height: 60,
64                  width: 60,
65                  decoration: BoxDecoration(
66                    borderRadius: BorderRadius.circular(100.00),
67                    color: Colors.green,
68                  ),
69                  child: Icon(Icons.keyboard_arrow_down, color: Colors.black),
70                ),
71                Container(
72                  margin: EdgeInsets.only(left: 50.00, top: 70.00),
73                  height: 60,
74                  width: 60,
75                  decoration: BoxDecoration(
76                    borderRadius: BorderRadius.circular(100.00),
77                    color: Colors.blueAccent,
78                  ),
```

```
79                  child: Icon(Icons.keyboard_arrow_up, color: Colors.black),
80                ),
81              ],
82            ),
83            Row(
84              mainAxisAlignment: MainAxisAlignment.center,
85              children: <Widget>[
86                Text(
87                  'Take a Quick War Quiz!',
88                  style: TextStyle(
89                    fontSize: 35.00,
90                    fontStyle: FontStyle.normal,
91                  ),
92                ),
93              ],
94            ),
95            Column(
96              children: [
97                Text(
98                  'Answer a few Questions to test your Knowledge, Scores will decide..\
99  .',
100                 textAlign: TextAlign.center,
101                 style: TextStyle(
102                   fontSize: 25,
103                 ),
104               ),
105               Text(
106                 '...EITHER...',
107                 style: TextStyle(
108                   fontSize: 22,
109                   fontStyle: FontStyle.italic,
110                   color: Colors.deepOrangeAccent,
111                 ),
112               ),
113               RaisedButton(
114                 child: Text(
115                   'You are a War Expert!',
116                   style: TextStyle(
117                     fontSize: 22,
118                     color: Colors.white,
119                   ),
120                 ),
121                 disabledColor: Colors.redAccent,
```

```
122                 onPressed: null,
123               ),
124               Icon(
125                 Icons.favorite,
126                 color: Colors.pink,
127                 size: 24.0,
128               ),
129               Text(
130                 '...OR ...',
131                 style: TextStyle(
132                   fontSize: 22,
133                   fontStyle: FontStyle.italic,
134                   color: Colors.deepOrangeAccent,
135                 ),
136               ),
137               RaisedButton(
138                 child: Text(
139                   'You are a Learned Person!',
140                   style: TextStyle(
141                     fontSize: 22,
142                     color: Colors.white,
143                   ),
144                 ),
145                 disabledColor: Colors.redAccent,
146                 onPressed: null,
147               ),
148               Icon(
149                 Icons.audiotrack,
150                 color: Colors.green,
151                 size: 30.0,
152               ),
153               Text(
154                 '...FINALLY...',
155                 style: TextStyle(
156                   fontSize: 22,
157                   fontStyle: FontStyle.italic,
158                   color: Colors.deepOrangeAccent,
159                 ),
160               ),
161               RaisedButton(
162                 child: Text(
163                   'You need to Study More!',
164                   style: TextStyle(
```

```
                    fontSize: 22,
                    color: Colors.white,
                  ),
                ),
                disabledColor: Colors.redAccent,
                onPressed: null,
              ),
              Row(
                mainAxisAlignment: MainAxisAlignment.center,
                children: <Widget>[
                  Expanded(
                    child: Padding(
                      padding: EdgeInsets.all(20.00),
                      child: Container(
                        alignment: Alignment.center,
                        height: 40.00,
                        decoration: BoxDecoration(
                          color: Colors.blueGrey,
                          borderRadius: BorderRadius.circular(30.00),
                        ),
                        child: RaisedButton(
                          child: Text(
                            'Let\'s Start...',
                            style: TextStyle(
                              fontSize: 22,
                              color: Colors.blue,
                            ),
                          ),
                          //disabledColor: Colors.redAccent,
                          onPressed: (){
                            clickNextPage(context);
                          },
                        ),
                      ),
                    ),
                  ),
                ],
              ),
            ],
          ),
        ),
      ),
```

```
208     );
209   }

}
```

// snackbar_and_page.dart

import ‘package:flutter/material.dart’;

final GlobalKey<ScaffoldState> scaffoldKey = GlobalKey<ScaffoldState>(); final SnackBar snackBarOne = const SnackBar(content: Text(‘Alert has been pressed!’, style: TextStyle(fontSize: 30),)); final SnackBar snackBarTwo = const SnackBar(content: Text(‘Search has been pressed!’, style: TextStyle(fontSize: 30),)); final SnackBar snackBarThree = const SnackBar(content: Text(‘Navigation has been pressed!’, style: TextStyle(fontSize: 30),));

void clickNextPage(BuildContext context) { Navigator.push(context, MaterialPageRoute(builder: (BuildContext context) { return Scaffold(appBar: AppBar(title: const Text(‘Know Yourself…’),), body: const Center(child: Text(‘Dig deep into every layer of your mind to find yourself…’, style: TextStyle(fontSize: 24), textAlign: TextAlign.center,),),); },)); }

The next screenshot shows us how this layout widgets work (Figure 6.7). We have not used any image. We created the logo of the page simply by using Stack and Icons widget classes.

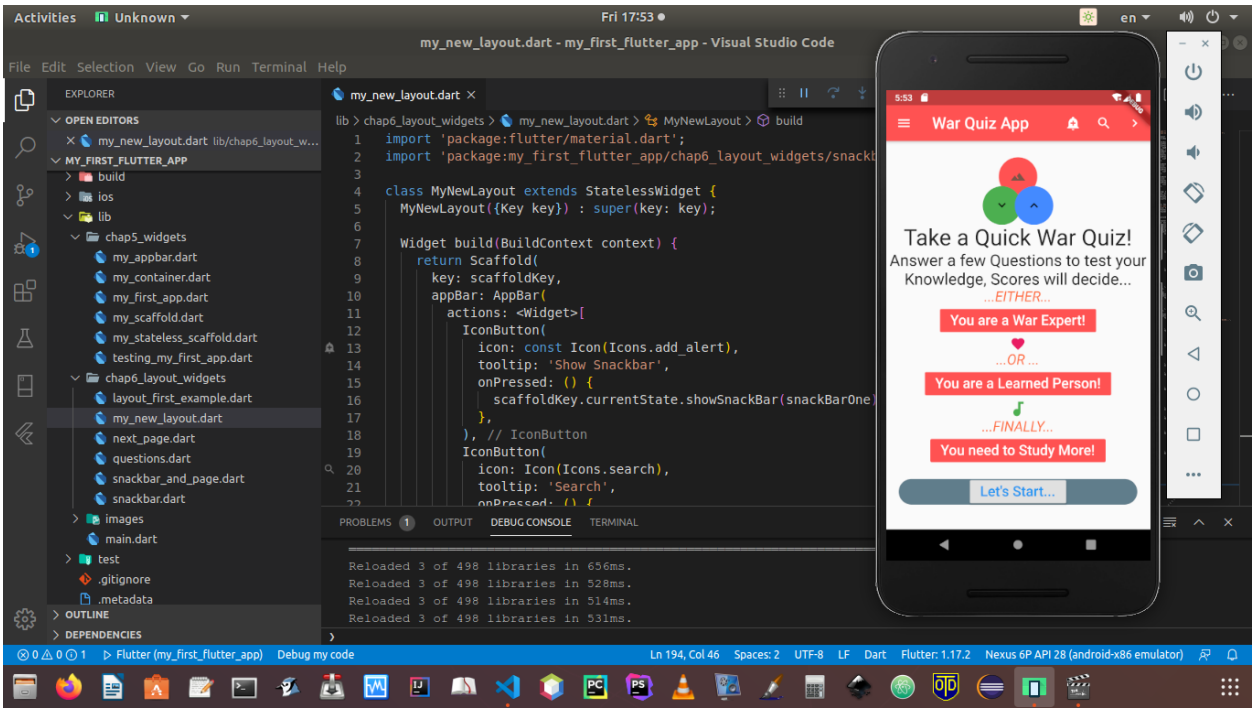


Figure 6.7 – A complete new look of our application

We will take a look at some particular spots in our ‘body’ widget, where we have designed the logo, and after that in a series of Row and Column widget classes we have added more layout capabilities.

body: Center(child: Column(mainAxisAlignment: MainAxisAlignment.start, children: <Widget>[Stack(alignment: Alignment.topCenter, children: <Widget>[Container(margin: EdgeInsets.only(top: 25.00), height: 60, width: 60, decoration: BoxDecoration(borderRadius: BorderRadius.circular(100.00), color: Colors.redAccent,), child: Icon(Icons.landscape, color: Colors.brown),), Container(margin: EdgeInsets.only(top: 70.00, right: 50.00), height: 60, width: 60, decoration: BoxDecoration(borderRadius: BorderRadius.circular(100.00), color: Colors.green,), child: Icon(Icons.keyboard_arrow_down, color: Colors.black),), Container(margin: EdgeInsets.only(left: 50.00, top: 70.00), height: 60, width: 60, decoration: BoxDecoration(borderRadius: BorderRadius.circular(100.00), color: Colors.blueAccent,), child: Icon(Icons.keyboard_arrow_up, color: Colors.black),),],), Row(mainAxisAlignment: MainAxisAlignment.center, children: <Widget>[Text('Take a Quick War Quiz!', style: TextStyle(fontSize: 35.00, fontStyle: FontStyle.normal,),),],), Column(children: [Text('Answer a few Questions to test your Knowledge, Scores will decide…', textAlign: TextAlign.center, style: TextStyle(fontSize: 25,),), Text('…EITHER…', style: TextStyle(fontSize: 22, fontStyle: FontStyle.italic, color: Colors.deepOrangeAccent,),), RaisedButton(child: Text('You are a War Expert!', style: TextStyle(fontSize: 22, color: Colors.white,),), disabledColor: Colors.redAccent, onPressed: null,), Icon(Icons.favorite, color: Colors.pink, size: 24.0,), …

The code has been shortened as we need not repeat the same snippets again and again. Especially watch the Stack class widget, which uses children widget of Container class widgets. We have used three Containers, and by modifying the padding, margins, etc.

Take a close look at one Container class widget that has been placed inside the Stack class widget.

Container(margin: EdgeInsets.only(left: 50.00, top: 70.00), height: 60, width: 60, decoration: BoxDecoration(borderRadius: BorderRadius.circular(100.00), color: Colors.blueAccent,), child: Icon(Icons.keyboard_arrow_up, color: Colors.black),),

By changing the 'EdgeInsets' class widget we can control the position of the Icons. Although we are not going to use this layout when we will create the final application, yet hopefully this layout mechanism gives us an idea of how we can add or modify different type of layout capabilities.

Abstract Class and Methods

Let us look back to some Dart programming concepts again to understand a few more capabilities of Flutter layout mechanism.

An abstract class is something where we define an interface but leaving its implementation up to other classes. Quite naturally, abstract methods can only exist in abstract classes. In abstract methods, we just leave a semicolon (;) at the end of the method name. We don't define the method body.

```
//code 6.7
//we cannot instantiate any abstract class
abstract class volume{
//we can declare instance variable
int age;
void increase();
void decrease();
// a normal function
void anyNormalFunction(int age){
    print("This is a normal function to know the $age.");
}
}
class soundSystem extends volume{
void increase(){
    print("Sound is up.");
}
void decrease(){
    print("Sound is down.");
}
//it is optional to override the normal function
void anyNormalFunction(int age){
    print("This is a normal function to know how old the sound system is: $age.");
}
}
main(List<String> arguments){
var newSystem = soundSystem();
newSystem.increase();
newSystem.decrease();
newSystem.anyNormalFunction(10);
}
And here is the output of code 5.1.
Sound is up.
Sound is down.
This is a normal function to know how old the sound system is: 10.
```

We have used the abstract modifier to define an abstract class and it cannot be instantiated. So we can say, the abstract class and methods summarize the main ideas; and we can extend that idea. There are a few things to remember:

1. In abstract class, we can use normal properties and methods.
2. It is optional that we would override the method.
3. We can also define instance variables in the abstract class.

Advantage of Interfaces

In some cases, we need to use reference variables and methods of many classes at the same time. 'Mixins' may come to our help. No doubt. But there is another good feature in Dart, we can also use – interfaces. Let us see the code first then we will discuss it in detail.

```
//code 6.8
// interface in dart is class, but we don't extend, we implement it
class Vehicle{
void steerTheVehicle() {
    print("The vehicle is moving.");
}
}
class Engine{
//in the interface, but only visible when used publicly
final _name;
//not in the interface, since it is a constructor
Engine(this._name);
String lessOilConsumption(){
    return "It consumes less oil.";
}
}
class Car implements Vehicle, Engine{
get _name => "";
void carName(String name) => print("$name");
void steerTheVehicle() {
    print("The car is moving.");
}
String lessOilConsumption(){
    print("This model of car consumes less oil.");
}
void ridingExperience(Engine engine) => print("This car gives good ride, because the\
 engine is ${engine._name}");
}
main(List<String> arguments){
var car = Car();
car.carName("Opel");
car.steerTheVehicle();
car.lessOilConsumption();
car.ridingExperience(Engine("Suzuki"));
}
Here is the output of code 6.1:
```

```
Opel
The car is moving.
This model of car consumes less oil.
This car gives a good ride because the engine is Suzuki
```

When a class implements an interface, it implicitly defines all the instance members of the implemented class. A class implements one or more than one interfaces at a time by declaring the 'implement' clause.

Considering the above code, we see that class Car supports class Vehicle and class Engine's API and for that requirement, the class Car implements class Vehicle and class Engine's interfaces.

A class cannot extend more than one classes. But it can implement more than one interface by declaring the implement clause.

Therefore, a few things to remember:

1. The biggest advantage of the interface is that we can implement multiple classes.
2. We cannot inherit multiple classes through inheritance.

Static Variables and Methods

To implement class-wide variables and methods, we use the static keyword. Static variables are also called class variables. Let us first see a code snippet and after that, we will discuss the advantages and disadvantages of static variables and methods.

```
//code 6.9
// static variables and methods consume less memory
// they are lazily initialized
class Circle{
static const pi = 3.14;
static Function drawACircle(){
    //from static method you cannot call a normal function
    print(pi);
}
Function aNonStaticFunction(){
    //from a normal function ou can call a static meethod
    Circle.drawACircle();
    print("This is normal function.");
}
}
main(List<String> arguments){
var circle = Circle();
```

```
circle.aNonStaticFunction();
Circle.drawACircle();
}
And here is the output:
3.14
This is normal function.
3.14
```

As you see, static variables are useful for class-wide state and constants. So in the main() method we can add this line at the end:

```
main(List<String> arguments){
var circle = Circle();
circle.aNonStaticFunction();
Circle.drawACircle();
print(Circle.pi);
}
```

And get the value of constant 'pi' again. Here, 'Circle.pi' is the class variable. And the class method is: 'Circle.drawACircle();'. The biggest advantage of using static variables and methods is it consumes less memory. An instance variable once instantiated, consumes memory whether it is being used or not. The static variables and methods are not initialized until they are used in the program. It consumes memory when they are used.

A few things to remember:

1. From a normal function, you can call a static method.
2. From a static method, you cannot call a normal function.
3. In a static method, you cannot use the 'this' keyword. It is because the static methods do not operate on an instance and thus do not have access to this.

So, in the end, we can conclude that using static variables and methods depend on the context and situations. In the next part of the book, where we will build native iOS and Android mobile apps with the help of Flutter framework, you will see how and when we use static variables and methods.

The 'Closure' is a Special Function

We can define Closure in two ways. According to the first definition, we can say that Closure is the only function that has access to the parent scope, even after the scope is closed.

To understand this definition, let us see a very short code snippet:

```
//code 6.10
//a closure can modify the parent scope
String message = "Any Parent String";
Function overridingParentScope = (){
String message = "Overriding the parent scope";
print(message);
};
main(List<String> arguments){
print(message);
overridingParentScope();
}
The output is:
Any Parent String
Overriding the parent scope
```

By the second definition, we can say that a Closure is a function object that has access to the variables in its lexical scope, even when the function is used outside of its original scope.

```
//code 6.11
Function show = (){
String pathToImage = "This is an old path.";
Function gettingImage(){
    String path = "This is a new path to image.";
    print(path);
}
return gettingImage;
};
main(List<String> arguments){
var showing = show();
showing();
}
```

Here is the output: This is a new path to image.

It actually returns a function object 'gettingImage' that has accessed the variable in its lexical scope. So at the end of this section, we can summarize the points about Closure.

1. In several other languages, you are not allowed to modify the parent variable.
2. However, within a closure, you can mutate or modify the values of variables present in the parent scope.

Now we will conclude our whole journey to study the nameless functions in one single code base, and we will also see the output.

```
//code 6.12
//Lambda is an anonymous function
class AboutLambdas{
//first way of expressing Lambda or anonymous function
Function addingNumbers = (int a, int b){
    var sum = a + b;
    //print(sum);
    return sum;
};
Function multiplyWithEight = (int num){
    return num * 8;
};
//second way of expressing Lambda by Fat Arrow
Function showName = (String name) => name;
//higher order functions pass function as parameter
int higherOrderFunction(Function myFunction){
    int a = 10;
    int b = 20;
    print(myFunction(a, b));
}
//returning a function
Function returningAFunction(){
    Function showAge = (int age) => age;
    return showAge;
}
//a closure can modify the parent scope
String anyString = "Any Parent String";
Function overridingParentScope = (){
    String message = "Overriding the parent scope";
    print(message);
};
Function show = (){
    String pathToImage = "This is an old path.";
    Function gettingImage(){
    String path = "This is a new path to image.";
    print(path);
    }
    return gettingImage;
};
}
main(List<String> arguments){
var add = AboutLambdas();
var addition = add.addingNumbers(5, 10);
```

```
print(addition);
var mul = AboutLambdas();
var result = mul.multiplyWithEight(4);
print(result);
var name = AboutLambdas();
var myName = name.showName("Sanjib");
print(myName);
var higher = AboutLambdas();
var higherOrder = higher.higherOrderFunction(add.addingNumbers);
higherOrder;
var showAge = AboutLambdas();
var showingAge = showAge.returningAFunction();
print(showingAge(25));
var sayMessage = AboutLambdas();
sayMessage.overridingParentScope();
var image = AboutLambdas();
var imagePath = image.show();
imagePath();
}
```

And in the output we will see how the nameless functions work in different ways.

```
//output
15
32
Sanjib
30
25
Overriding the parent scope
This is a new path to image.
```

Data Structures and Collections

Understanding the concepts of data structures and collections, as a whole, plays a crucial role in your future Dart programming. We will see in a minute that there are four types of data structures in Dart: List, Set,Map, and Queue. In my opinion, Lists and Maps will cover almost everything, so you hardly need the other two types in your programming life, except a few occasions.

Anyway, although we have seen an introduction to collections before, we will learn these data structures concepts exclusively in this chapter. We will cover all the concepts of Dart collections in detail.

In a nutshell, data structures are something that helps you to organize information for storage and later retrieval.

Although Sets are not necessary for day-to-day programming like Lists and Maps, it is good on some occasions, especially, when your data elements are unique. We will see them in a minute. Remember, learning data structures properly will definitely make you a good Dart programmer in the future.

So let us start with Lists.

Lists: Fixed Length and Growable

If you are a complete beginner, you may not have heard about 'array'. Of course, you have heard it, if you have already had a programming background. You won't find the terms like 'array' or 'associative array' in Dart. But Dart collections can be used to duplicate the data structures like an array.

Now, what is List?

The list is a simple ordered group of objects. Creating a List seems easy because Dart core libraries have the necessary support and a List class. There are two types of Lists.

1. Fixed Length List
2. Growable List

In the Fixed Length List, the length of Lists cannot change at run-time; however, in the second type, Growable List, the length can change at run-time. We will see them separately.

```
1  //code 6.13
2  int listFunction(){
3  List<int> nameOfTest = List(3);
4  nameOfTest[0] = 1;
5  nameOfTest[1] = 2;
6  nameOfTest[2] = 3;
7  //there are three methods to capture the list
8  //1. method
9  for(int element in nameOfTest){
10     print(element);
11 }
12 print("-----------");
13 //2. method
14 nameOfTest.forEach((v) => print('${v}'));
15 print("-----------");
16 //3. method
```

```
for(int i = 0; i < nameOfTest.length; i++){
    print(nameOfTest[i]);
}
}
main(List<String> arguments){
listFunction();
}
```

As you see this is an ordered list of 3 numbers. We are getting the output by using three methods, very simple and straight forward.

```
//output
1
2
3
-----------
1
2
3
-----------
1
2
3
```

Now we will see an example of Growable List.

```
//code 6.14
Function growableList(){
//1. method
List<String> names = List();
names.add("Mana");
names.add("Babu");
names.add("Gopal");
names.add("Pota");
//there are two methods to capture the list
print("-----------");
//1. method
names.forEach((v) => print('${v}'));
print("-----------");
//2. method
for(int i = 0; i < names.length; i++){
    print(names[i]);
```

```
}
}
main(List<String> arguments){
growableList();
}
```

It is also very straight forward, we have not passed any number through the List() and keeping it open lets us add any number of elements into it. Here we have added a few names. And we can capture the List elements through two methods, instead of three. The output is quite expected.

```
//output
-----------
Mana
Babu
Gopal
Pota
-----------
Mana
Babu
Gopal
Pota
```

So it is evident from the output and the code that Growable Lists are dynamic in nature. We can dynamically add any number of elements and we can also remove it by a simple method: ‘ names.remove(“any name”)’. We can also use the key; as this ordered list starts from 0. So we can remove the first name just by passing this key value: ‘ names.removeAt(0)’. We use the ‘removeAt(key)’ method for that operation. We can also clear the Lists just by typing: ‘ names.clear()’.

Set: An Unordered Collections of Unique Items

The headline says everything. A Set represents a collection of objects in which each object can occur only once; it literally stands for the uniqueness of the items. In the dart core library, there is a Set class that supports to achieve this criterion.

Since Set is an unordered collection of unique items, you cannot get element1s by the INDEX. There is a concept called ‘HashSet’ that actually implements the unordered Set and it is based on hash-table based Set implementation. We will look into those features in a minute.

```
//code 6.15
void setFunction(){
//set is an unordered collections of unique items
//cannot get elements by INDEX since the items are unordered
//1. method of creating Set
Set<String> countries = Set.from(['India', 'England', 'US']);
Set<int> numbers = Set.from([1, 45, 58]);
Set<int> moreNumbers = Set();
moreNumbers.add(178);
moreNumbers.add(568);
moreNumbers.add(569);
//1. method
for(int element in numbers){
    print(element);
}
print("-----------");
//2. method
countries.forEach((v) => print('${v}'));
print("-----------");
for(int element in moreNumbers){
    if(moreNumbers.lookup(178) == 178){
    print(moreNumbers);
    break;
    }
}
//set
var fruitCollection = {'Mango', 'Apple', 'Jack fruit'};
print(fruitCollection.lookup('Something Else'));
//it gives null
//lists
List fruitCollections = ['Mango', 'Apple', 'Jack fruit'];
var myIntegers = [1, 2, 3, 'non-integer object'];
print(myIntegers[3]);
print(fruitCollections[0]);
}
main(List<String> arguments){
setFunction();
}
```

Let us see the output first, then we will be able to understand what happens.

```
//output
1
45
58
-----------
India
England
US
-----------
{178, 568, 569}
null
non-integer object
Mango
```

We have created Set of ‘countries’, ‘numbers’ and ‘morenumbers’; finally we have created a List at the end to distinguish between the characters of Lists and Sets. These three methods have created Lists:

```
Set<String> countries = Set.from(['India', 'England', 'US']);
Set<int> numbers = Set.from([1, 45, 58]);
Set<int> moreNumbers = Set();
```

We get the output of the first one we have by this method:

```
countries.forEach((v) => print('${v}'));
The second List has been retrieved by this method:
for(int element in numbers){
print(element);
}
```

And the values of the third Set we have captured using this method:

```
for(int element in moreNumbers){
if(moreNumbers.lookup(178) == 178){
    print(moreNumbers);
    break;
}
}
```

To manipulate a Set there are lots of methods available in Dart core libraries. You can use ‘moreNumbers.contains(value)’, ‘moreNumbers.remove(value)’ or ‘moreNumbers.isEmpty’ etc.

In this code snippet, the return value is ‘null’, since there is no such value present in the Set.

```
//set
var fruitCollection = {'Mango', 'Apple', 'Jack fruit'};
print(fruitCollection.lookup('Something Else'));
```

We need to remember one thing, when the Set type is integer, it is easier to use 'for' loop to loop over the elements. Otherwise, it will be wise to use 'foreach' as we have used in the above code:

```
countries.forEach((v) => print('${v}'));
```

In the next section we will see how Map in Dart works.

Maps: the Key, Value Pair

An unordered collection of Key-Value pair is known as Map in Dart. The main advantage of Map is the Key-Value pair can be of any type. In the next chapter, where we will discuss state management in Flutter, we will use Map, and the key-value pair.

To begin with, let us start with some points that we should remember while working with Map.

1. Each Key in a Map should be unique.
2. The value can be repeated.
3. The Map can commonly be called hash or dictionary.
4. Size of a Map is not fixed, it can either increase or decrease as per the number of elements. In other words, Maps can grow or shrink at runtime.
5. HashMap is an implementation of a Map and it is based on a Hash table.

Let us see a code snippet to understand how a Map works in Dart.

```
//code 6.16
void mapFunction(){
//unordered collection of key=>value pair
Map<String, String> countries = Map();
countries['India'] = "Asia";
countries["German"] = "Europe";
countries["France"] = "Europe";
countries["Brazil"] = "South America";
//1. method we can obtain key or value
for(var key in countries.keys){
    print("Countries' name: $key");
}
print("-----------");
```

```
for(String value in countries.values){
    print("Continents' name: $value");
}
//2. method
countries.forEach((key, value) => print("Country: $key and Continent: $value"));
//we can update any map very easily
if(countries.containsKey("German")){
    countries.update("German", (value) => "European Union");
    print("Updated country German.");
    countries.forEach((key, value) => print("Country: $key and Continent: $value"));
}
//we can remove any country
countries.remove("Brazil");
countries.forEach((key, value) => print("Country: $key and Continent: $value"));
print("Barzil has been removed successfully.");
print("-----------");
//3. method of creating a map
Map<String, int> telephoneNumbersOfCustomers = {
    "John" : 1234,
    "Mac" : 7534,
    "Molly" : 8934,
    "Plywod" : 1275,
    "Hagudu" : 2534
};
telephoneNumbersOfCustomers.forEach((key, value) => print("Customer: $key and Contac\
t NUmber: $value"));
}
main(List<String> arguments){
mapFunction();
}

And here is the output
Countries' name: India
Countries' name: German
Countries' name: France
Countries' name: Brazil
-----------
Continents' name: Asia
Continents' name: Europe
Continents' name: Europe
Continents' name: South America
Country: India and Continent: Asia
Country: German and Continent: Europe
```

```
Country: France and Continent: Europe
Country: Brazil and Continent: South America
Updated country German.
Country: India and Continent: Asia
Country: German and Continent: European Union
Country: France and Continent: Europe
Country: Brazil and Continent: South America
Country: India and Continent: Asia
Country: German and Continent: European Union
Country: France and Continent: Europe
Barzil has been removed successfully.
-----------
Customer: John and Contact NUmber: 1234
Customer: Mac and Contact NUmber: 7534
Customer: Molly and Contact NUmber: 8934
Customer: Plywod and Contact NUmber: 1275
Customer: Hagudu and Contact NUmber: 2534
```

There are three methods that we use to retrieve the values of a Map.

```
//1. method we can obtain key or value
for(var key in countries.keys){
print("Countries' name: $key");
}
print("-----------");
//2. Method
for(String value in countries.values){
print("Continents' name: $value");
}
//3. method
countries.forEach((key, value) => print("Country: $key and Continent: $value"));
```

Besides, there are several methods to add, update or remove the elements in a Map. As we progress and build apps in native iOS or Android, we will see more features of Map. Lastly we will see another collection feature in Map, which is called Queue.

Queue is Open-Ended

The queue is useful when you try to build a collection that can be added from one end and can be deleted from another end. The values are removed or read in the order of their insertion.

Consider this code:

```
//code 6.16
import 'dart:collection';
main(List<String> arguments){
Queue myQueue = new Queue();
print("Default implementation ${myQueue.runtimeType}");
myQueue.add("Sanjib");
myQueue.add(54);
myQueue.add("Howrah");
myQueue.add("sanjib12sinha@gmail.com");
for(var allTheQueues in myQueue){
    print(allTheQueues);
}
print("----------");
print("We are removing the first element ${myQueue.elementAt(0)}.");
myQueue.removeFirst();
for(var allTheQueues in myQueue){
    print(allTheQueues);
}
print("----------");
print("We are removing the last element ${myQueue.elementAt(2)}.");
myQueue.removeLast();
for(var allTheQueues in myQueue){
    print(allTheQueues);
}
}
```

The output is as expected; it gives us the full lists of what we have added in the Queue. After that we have removed the first and the last element1.

```
//output
Default implementation ListQueue<dynamic>
Sanjib
54
Howrah
sanjib12sinha@gmail.com
----------
We are removing the first element Sanjib.
54
Howrah
sanjib12sinha@gmail.com
----------
We are removing the last element sanjib12sinha@gmail.com.
```

```
54
Howrah
```

In most cases, as I have said earlier at the beginning of Data Structures chapter, we can handle with Lists and Maps. So Queue is an option that you may need some time; but not very often.

Callable Classes

It is a very interesting feature in Dart, where we can call a Class like a function. All we need to do is just implement the call() function.

```
//code 6.17
//when dart class is callable like a function, use call() function
class Person{
String name;
String call(String message, [name]){
    return "This message: '$message', has been passed to the person $name.";
}
}
main(List<String> arguments){
var John = Person();
John.name = "John Smith";
String name = John.name;
String msgAndName = John("Hi John how are you?", name);
print(msgAndName);
}
And here is the output:
This message: 'Hi John how are you?', has been passed to the person John Smith.
```

Here, 'John' is the instance variable and the 'Person()' is the class object. The class 'Person' is called like a function because we have implemented the call() function, through which we have passed two parameters:'String message' and the optional parameter 'name'. Finally, we have passed both and captured the value through 'msgAndName'.

Exception Handling

During the execution of any program, if Exception occurs, the program is disrupted. The normal flow of the program gets disturbed.

For the complete beginners, these concepts may seem a little bit tough. Seasoned programmers will understand how to catch the exceptions and display them in a nice formatted way. From the complete

beginners' perspective, we can say, there are some errors that usually disrupt the flow of program automatically.

Suppose you want to divide a number by zero.

It is an impossible task and will disrupt the flow resulting in some errors. However, you cannot control a user's behavior, so you need to take every precaution to avoid getting such ugly errors.

Dart programmers have thought about it and they have included many built-in exceptions. One of them is: 'IntegerDivisionByZeroException'; it is thrown when a number is divided by zero. Likewise, when a scheduled timeout happens while waiting for an 'async' result, the 'Timeout' exception occurs. If deferred libraries fail to load, there is 'DeferredLoadException' happens.

Suppose a string data cannot be parsed because it does not have the proper format. In that case, 'FormatException' exception occurs. Any input and output related exceptions are captured through 'IOException' class.

In Dart, everything is an Object and behind an object, there must be a class. In the exception handling cases, the class 'Exception' plays the main role to prevent the application from terminating abruptly.

Let us see some code snippets so that we can understand easily how we can catch the exceptions.

```
1  //code 6.18
2  main(List<String> arguments){
3  try{
4      int result = 10 ~/ 0;
5      print("The result is $result");
6  } on IntegerDivisionByZeroException{
7      print("We cannot divide by zero");
8  }
9  try{
10     int result = 10 ~/ 0;
11     print("The result is $result");
12 } catch(e){
13     print(e);
14 }
15 try{
16     int result = 10 ~/ 0;
17     print("The result is $result");
18 } catch(e){
19     print("The exception is : $e");
20 } finally{
21     print("This is Finally and it always is executed.");
22 }
23 }
24
```

```
//Here is the output:
We cannot divide by zero
IntegerDivisionByZeroException
The exception is : IntegerDivisionByZeroException
This is Finally and it always is executed.
```

As you see in the output, there are several methods through which we can catch the exceptions. If we know the type of exception, we can use try/on. As we have used in the above code:

```
try{
int result = 10 ~/ 0;
print("The result is $result");
} on IntegerDivisionByZeroException{
print("We cannot divide by zero");
}
```

In this case, we did know what type of exception could be generated. So we have used “try/on”. But what happens, when we do not know the exception? The syntax of handling exception is as given below:

```
try{
int result = 10 ~/ 0;
print("The result is $result");
} catch(e){
print(e);
}
```

The catch block is used when the handler needs the exception object.

The try block may be followed by finally block after the catch block. We have used the same thing in the above code:

```
try{
int result = 10 ~/ 0;
print("The result is $result");
} catch(e){
print("The exception is : $e");
} finally{
print("This is Finally and it always is executed.");
}
```

The final block will be executed at the end, whatever be the outcome:

```
The exception is : IntegerDivisionByZeroException
This is Finally and it always is executed.
```

In case, an exception occurs in the try block, the control goes to the catch block; and at the end, the final block gives the output.

Dart Packages and Libraries

Dart programmings very heavily rely on libraries. There are several common libraries that serve many purposes while we build any Dart application. So far you have seen many built-in functions that we have used in many user-defined functions; such as 'dart:core' libraries provide assistance for numbers, string-specific operations or collections. With the help of 'dart:math' we can do many types of mathematical operations quite easily.

We can also build our own libraries. In fact, as you progress you will feel the necessity of creating your own libraries. Besides, you can get additional libraries by importing them from packages.

We should also know why we need libraries? To create a modular and shareable code base, we need a good organization of the code base. It is an essential part of object-oriented programming. Libraries not only provide support for modular, object-oriented programming, it also gives you a kind of privacy in your own code.

Identifiers, starting with (_) underscore, are only visible in your libraries.

Libraries also give you good support to avoid name conflicts which is an essential part of coding. How it does so? Let us see an example to clarify those points. First, we have created a 'Relational-Operators.dart' file inside the 'lib' folder.

```
//code 6.19
//lib/ RelationalOperators.dart
class TrueOrFalse{
int firstNum = 40;
int secondNum = 40;
int thirdNum = 74;
int fourthNum = 56;
void BetweenTrueOrFalse(){
    if (firstNum == secondNum || thirdNum == fourthNum){
    print("If choice between 'true' or 'false', in this case the 'TRUE' gets the pre\
cedence. $firstNum is equal to $secondNum");
    } else print("Nothing happens.");
}
void BetweenTrueAndFalse(){
    if (firstNum == secondNum && thirdNum == fourthNum){
    print("It will go to else clause");
```

```
    } else print("If choice between 'true' and 'false', in this case the 'FALSE' get\
s the precedence. $thirdNum is not equal to $fourthNum");
}
}
```

Next, we create a file 'PowProject.dart' inside the 'lib' folder.

```
//code 6.20
//lib/PowProject.dart
class PowProject{
void MultiplyByAGivenNumber(int fixedNumber, int givenNumber){
    int result = fixedNumber * givenNumber;
    print(result);
}
void pow(int x, int y){
    int addition = x + y;
    print(addition);
}
}
```

Now take a look at the 'main()' function body:

```
//code 6.21
import 'dart:math' as math;
import 'package:IdeaProjects/PowProject.dart';
import 'package:IdeaProjects/RelationalOperators.dart' as Relation;
main(List<String> arguments){
print("Prinitng 2 to the power 5 using Dart's built-in 'dart:math' library.");
var int = math.pow(2, 5);
print(int);
print("Now we are going to use another 'pow()' function from our own library.");
var anotherPowObject = PowProject();
anotherPowObject.MultiplyByAGivenNumber(4, 3);
anotherPowObject.pow(2, 12);
print("Now we are going to use another library to test the relational operators.");
var trueOrFalse = Relation.TrueOrFalse();
trueOrFalse.BetweenTrueOrFalse();
trueOrFalse.BetweenTrueAndFalse();
}
```

In the 'lib' or libraries folder we have created two classes. One of them has a function called: 'pow()'. But we know that the built-in 'dart:math' library has a function of the same name: 'pow()'. By any

way, we cannot use those same-name functions consecutively. It will give us errors. So to avoid the name conflict what we have done,we created our own library and defined it inside the class. Quite naturally, for the book's sake, our created 'pow()' function is doing something different than calculating the power of a number.

Look at the top of the main() function.

```
import 'dart:math' as math;
import 'package:IdeaProjects/PowProject.dart';
import 'package:IdeaProjects/RelationalOperators.dart' as Relation;
```

We have used the keyword 'import' to specify how our libraries, besides the core libraries can be used. After the 'import' we need to pass an argument which is nothing but a URI (Uniform Resource Identifier) specifying the libraries. For any built-in libraries, the URI has the special 'dart:...' scheme. For other libraries, you can use the file system path or the 'package:...' scheme. We have used the 'package:...' scheme; it is easy and it is provided b the package manager such as the 'pub' tool. When we directly use the libraries, we use a normal line like this:

```
import 'package:IdeaProjects/PowProject.dart';
```

In that case, we can directly create the class object that belongs to that particular library, such as:

```
var anotherPowObject = PowProject();
```

However, there is another good method; we can call any library by a name, like this:

```
import 'package:IdeaProjects/RelationalOperators.dart' as Relation;
```

The advantage is, now we can create any class object belonging to that library, such as:

```
var trueOrFalse = Relation.TrueOrFalse();
```

These prefixes basically are used to avoid name conflicts. You can write same-name classes in libraries and you can use them by giving them any name.

There are a few good built-in libraries that come with Dart; you need not write them again. Here are some of them:

1. 'dart:core' : It gives us many core functionalities. It is automatically imported into every Dart program.
2. 'dart:math' : You have seen how we have used the core mathematical libraries in our program. We can do many types of mathematical operations using that library; we can generate random numbers.

3. ‘dart:convert’ : Converting between different data representations is made easy through this library; this conversion includes JSON and UTF-8.
4. ‘dart:async’ : This library is beyond the scope of this book. I have written a separate book on Dart advanced programming. Please consult that book to know how Dart helps asynchronous programming, with classes such as Future and Stream.

Before concluding this book, we will have a look at the user-defined libraries again, that time for Flutter libraries. Usually, you can create any package libraries inside the ‘/lib’ directory and ‘import’ them, as we have done. Besides, you can also create sub-folders inside ‘/lib’ and create the hierarchies as needed.

Want to read more Flutter related Articles and resources?

For more Flutter related Articles and Resources[7]

[7] https://zerodotone.net

7. Introduction to State Management and Form Validation in Flutter and Dart

Some form of user interaction with the system is needed in any application, be it a mobile application or web application. If you already have some web development experience, you may have managed state in one way or other. Because of 'state', we have persistent data all along any application. Therefore, state is some kind of app data, which is persistent all over its life cycle.

State of any application can be managed by various ways; in Flutter also, it is true. Although there are several state management patterns in flutter, we cannot dive deep into every pattern here, for many reasons. Because this book is aimed for absolute beginners, we will try to stick around the basic. We want to understand what state management is, and how it works in flutter.

As we have already said that state represents some kind of user interactions. A user presses a button and it gives her some output. She presses again, it gives another output; it might keep going on until the output list is exhausted.

It cannot be achieved by a 'StateLess' widget. By using a 'StateLess' widget we can represent a 'RaisedButton', however, it won't work or won't give some output, until we 'set state'. Setting state can only be done by the 'StateFul' widget. With the help of 'Stateful' widget we can build from simple to complex user interactions. Probably we have noticed that whenever we create a flutter application it comes up with a 'StateFul' widget counter, where pressing the 'plus' button increases the number.

This is Google's BLoC pattern. The advantage of this pattern is we do not need any type of outside libraries. Therefore, we can use this pattern for any type of application, simple or complex.

Flutter renders the widgets by building the 'element' trees. Just picture this: a Container widget has a child element, which has a Padding widget, which has a child element that has a Row widget, which has a children widget,and so on.

Now inside a 'StateLess' widget, we can have a RaisedButton widget, which passes a parameter called 'onPress'; it is an anonymous void function that could call another named function that returns a value.

Now in that named function, we can set the state. When we create a 'StateFulWidget', its variables are immutable just like any 'StateLessWidget'; you can only declare any 'final' property, which is immutable. However, 'StateFulWidget' creates an associated state object by 'createState()' method.

State is mutable

Let us first see an image, which represents a simple button.

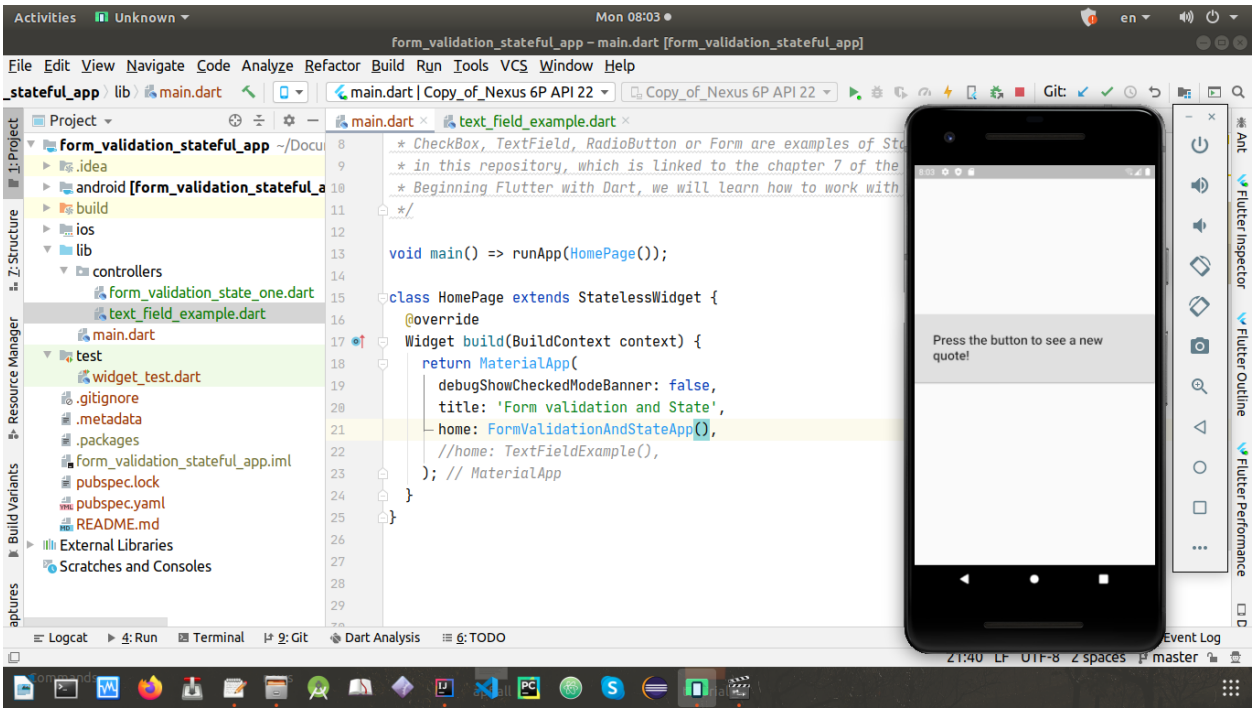


Figure 7.1 – A simple RaisedButton

If we press this button, it will give us different quotes, until the list is exhausted. The state object holds to the app data persistently throughout its life-cycle. Let us see the full code and after that we will dissect the code and try to understand how it works.

```
//code 7.1
import 'package:flutter/material.dart';

/**
* inheriting StatefulWidget makes a class immutable
*/

class FormValidationAndStateApp extends StatefulWidget {
/**
* the widget returns the state in createState() method
*/
@override
_FormValidationAndStateAppState createState() => _FormValidationAndStateAppState();
}
```

```
/**
* a simple example of state where user presses a button to see a list of quotes
*/
class _FormValidationAndStateAppState extends State<FormValidationAndStateApp> {

var _quotes = [
    '',
    'Life is a Tragedy',
    'Life is Beautiful',
    'Life consists of problems to solve',
];

int _questionIndex = 0;

int _answerQuestions() {
    /**
    * calling the setState() method makes the change and redraw the widget
    */
    setState(() {
    _questionIndex = _questionIndex + 1;
    });
    if(_questionIndex == 4) {
    _questionIndex = 0;
    }
    return _questionIndex;
}

@override
Widget build(BuildContext context) {
    return Scaffold(
    body: Container(
        alignment: Alignment.center,
        child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        //crossAxisAlignment: CrossAxisAlignment.start,
        children: <Widget>[
            RaisedButton(
            padding: EdgeInsets.all(32.0),
            child: Text(
                'Press the button to see a new quote!',
                style: TextStyle(
                fontSize: 22,
                //color: Colors.blue,
```

```
                ),
            ),
            onPressed: () {
                _answerQuestions();
            },
            disabledColor: Colors.blueAccent,
            ),
            SizedBox(height: 10.0,),
            Text(
            '${_quotes[_questionIndex]}',
            style: TextStyle(
                fontSize: 40.0,
            ),
            textAlign: TextAlign.center,
            ),
        ],
        ),
    ),
    );
}
}


import 'package:flutter/material.dart';
import 'package:form_validation_stateful_app/controllers/form_validation_state_one.d\
art';
import 'package:form_validation_stateful_app/controllers/text_field_example.dart';

/**
* the state is mutable and might change during the lifetime of the widget
* each time it redraws the widget whenever the state is changed
* CheckBox, TextField, RadioButton or Form are examples of Stateful widgets
* in this repository, which is linked to the chapter 7 of the book
* Beginning Flutter with Dart, we will learn how to work with these widgets
*/

void main() => runApp(HomePage());

class HomePage extends StatelessWidget {
@override
Widget build(BuildContext context) {
    return MaterialApp(
    debugShowCheckedModeBanner: false,
```

```
    title: 'Form validation and State',
    home: FormValidationAndStateApp(),
    );
}
}
```

Now we are going to study the above code part by part, so that we could understand how it works.

```
/**
* inheriting StatefulWidget makes a class immutable
*/

class FormValidationAndStateApp extends StatefulWidget {
/**
* the widget returns the state in createState() method
*/
@override
_FormValidationAndStateAppState createState() => _FormValidationAndStateAppState();
}
```

Our 'FormValidationAndStateApp' extends a 'StateFulWidget'; and it makes the class immutable. However, at the same time it creates a state object. The next step is to use that state object.

```
/**
* a simple example of state where user presses a button to see a list of quotes
*/
class _FormValidationAndStateAppState extends State<FormValidationAndStateApp> { }
```

In between the curly braces, we will now write our code. To hold the state and iterate the list values using its index, we need to use the 'setState()' method.

```
var _quotes = [
    '',
    'Life is a Tragedy',
    'Life is Beautiful',
    'Life consists of problems to solve',
];

int _questionIndex = 0;

int _answerQuestions() {
    /**
```

```
    * calling the setState() method makes the change and redraw the widget
    */
    setState(() {
    _questionIndex = _questionIndex + 1;
    });
    if(_questionIndex == 4) {
    _questionIndex = 0;
    }
    return _questionIndex;
}
```

We have increased the question index value by 1, and finally when the index value reaches 4, it comes back to 0 again. Using a simple trick we have used a blank string to start with. We could have used a 'reset' button. We will see such examples later in this chapter.

With the help of two 'StateLessWidget' - the RaisedButton and Text, we press the button and catch the value.

```
RaisedButton(
            padding: EdgeInsets.all(32.0),
            child: Text(
                'Press the button to see a new quote!',
                style: TextStyle(
                fontSize: 22,
                //color: Colors.blue,
                ),
            ),
            onPressed: () {
                _answerQuestions();
            },
            disabledColor: Colors.blueAccent,
            ),
            SizedBox(height: 10.0,),
            Text(
            '${_quotes[_questionIndex]}',
            style: TextStyle(
                fontSize: 40.0,
            ),
            textAlign: TextAlign.center,
            ),
```

The next image shows the above example, where a user presses the button and gets the quote.

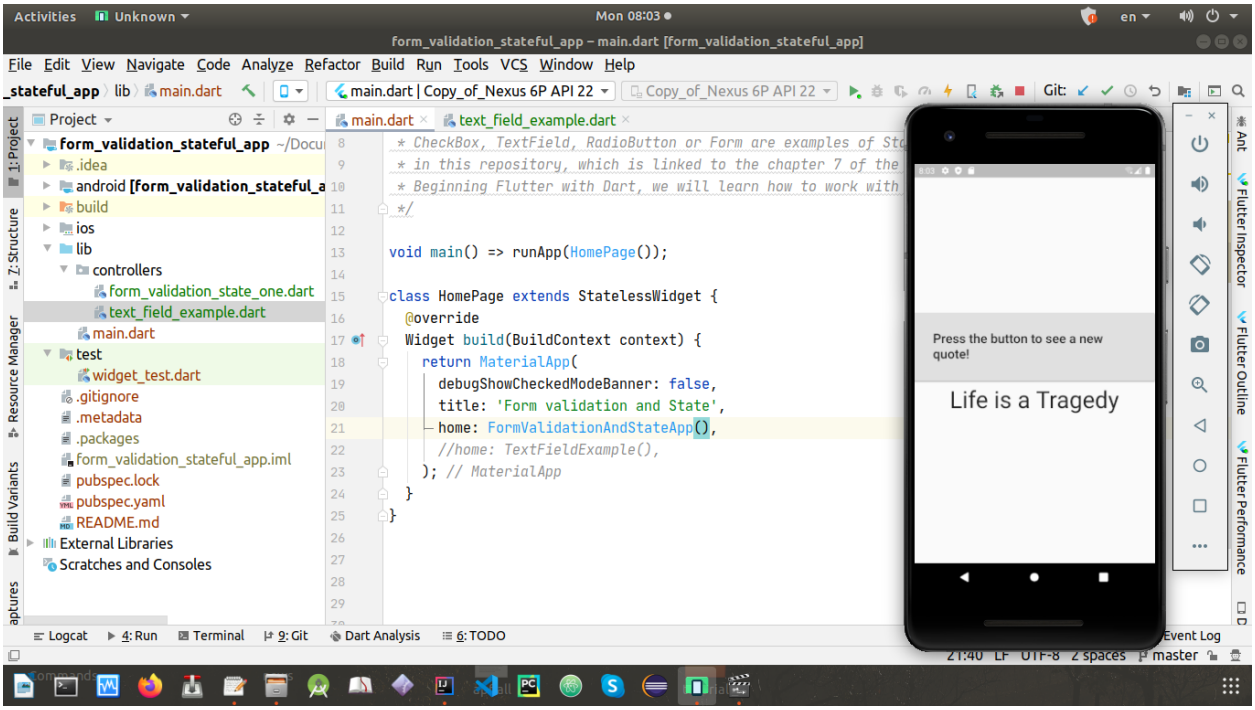


Figure 7.2 – Pressing the button gives an output

The state object is associated with the StateFulWidget's life-cycle. Flutter manages this complex process internally and with each sequence it re-draws the screen.

That never happens with the Stateless widget.

Life cycle of State

The state is mutable and it changes with the life-cycle of the widget. Each time the state changes, Flutter re-draws the widget.

The next examples will make the abstraction clearer than before. The TexField widget is by default a Stateful widget. It has a property 'onChanged'; the named parameter points to the anonymous function that passes the string value which the user types on the mobile screen. Whenever some text is is being typed through the 'TextField', the text is reflected on the screen.

```
/// it reflects the text input on the screen while typing
            onChanged: (String name) {
                setState(() {
                yourName = name;
                });
            },
            ),
```

However, it has also another property ‘onSubmitted’, which takes the input and gives the output on the screen.

```
onSubmitted: (String name) {
                setState(() {
                yourName = name;
                });
            },
```

Let us see how it looks like. The next image shows us how it looks like on the screen.

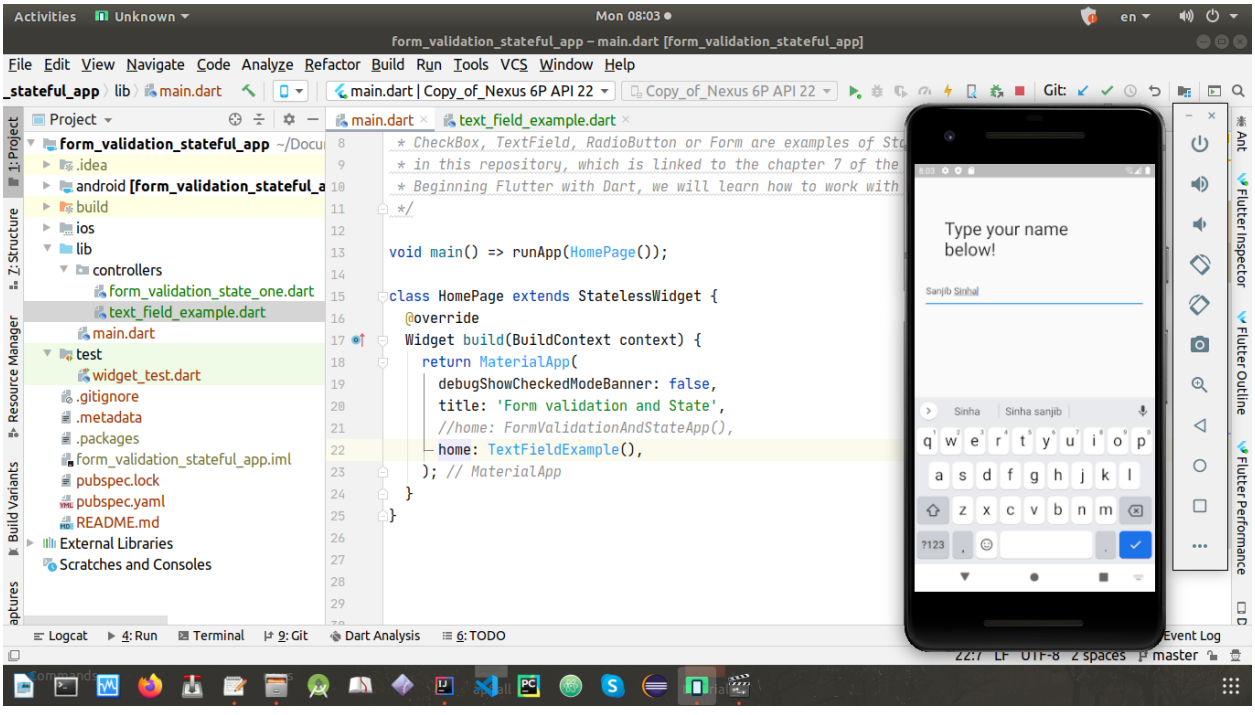


Figure 7.3 – The TextField Stateful widget

There is nothing fancy in the following code snippets. A simple TextField widget, where we would write any name and pressing the blue button below on the mobile keypad gives us an output.

```dart
//code 7.2

import 'package:flutter/material.dart';

class TextFieldExample extends StatefulWidget {
@override
_TextFieldExampleState createState() => _TextFieldExampleState();
}
/**
* a simple example of TextField, where one can type the name
* and see the output below; each time user types a String data type or text, a state\
ful widget
* is created, however, after that when she re-type another text, the set state is ca\
lled, which
* tells the framework to redraw the TextFieldExampleState widget and it's created ag\
ain
*/

class _TextFieldExampleState extends State<TextFieldExample> {

String yourName = '';

@override
Widget build(BuildContext context) {
    return Scaffold(
    body: Container(
        margin: EdgeInsets.all(20.0),
        child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        crossAxisAlignment: CrossAxisAlignment.start,
        children: <Widget>[
            Padding(
            padding: const EdgeInsets.all(32.0),
            child: Text(
                'Type your name below!',
                style: TextStyle(
                fontSize: 30.0,
                ),
            ),
            ),
            TextField(
            /*
            onSubmitted: (String name) {
```

```
                setState(() {
                yourName = name;
                });
            },
            */
            /// it reflects the text input on the screen while typing
            onChanged: (String name) {
                setState(() {
                yourName = name;
                });
            },
            ),
            Padding(
            padding: const EdgeInsets.all(32.0),
            child: Text(
                yourName,
                style: TextStyle(
                fontSize: 30.0,
                ),
            ),
            ),
        ],
        ),
    ),

    );
}
}


import 'package:flutter/material.dart';
import 'package:form_validation_stateful_app/controllers/form_validation_state_one.d\
art';
import 'package:form_validation_stateful_app/controllers/text_field_example.dart';

void main() => runApp(HomePage());

class HomePage extends StatelessWidget {
@override
Widget build(BuildContext context) {
    return MaterialApp(
    debugShowCheckedModeBanner: false,
    title: 'Form validation and State',
```

```
87      home: TextFieldExample(),
88      );
89  }
90  }
```

In the comments section we have written what is happening under the hood. Whenever we type some text and press the button, it gives us the output. Moreover, when we re-type any text, the set state is called, which tells the framework to redraw the assigned widget and it's created again.

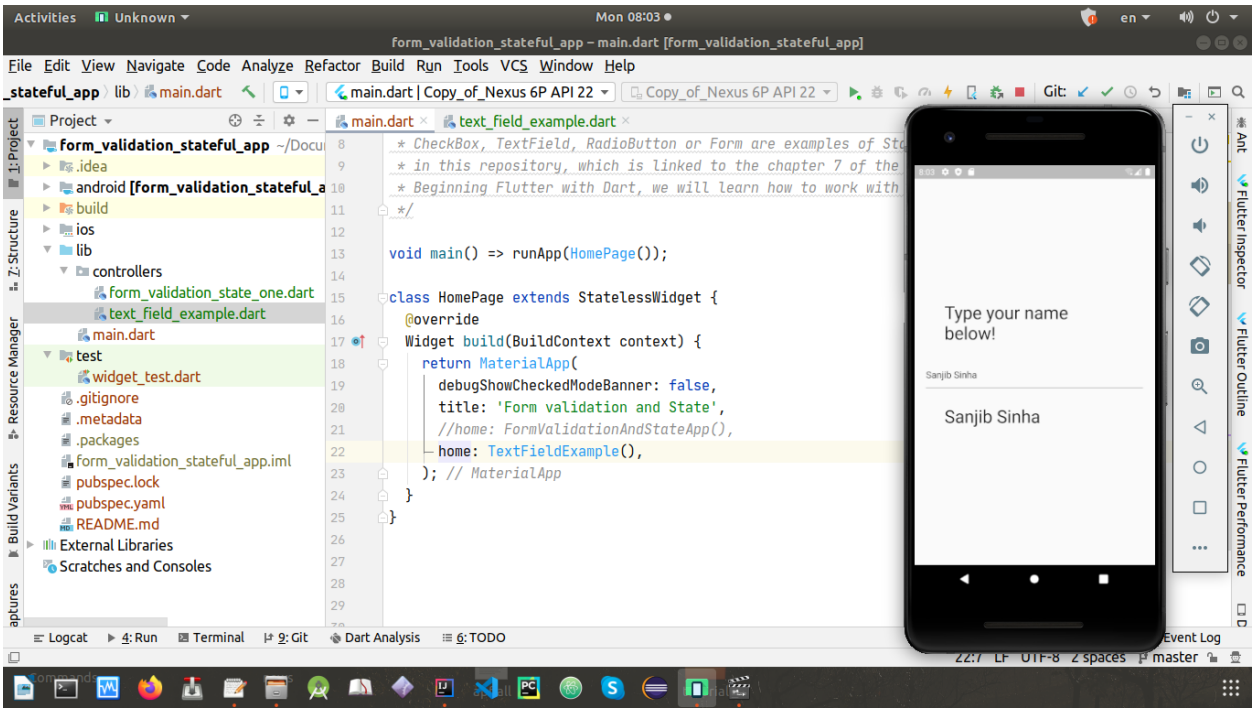


Figure 7.4 – The TextField widget gives us an output

As we have learned before, changing the 'onSubmitted' property to 'onChanged' gives us a visual representation of what we are typing on the screen. The next image shows us how this magical activity takes place directly on the screen.

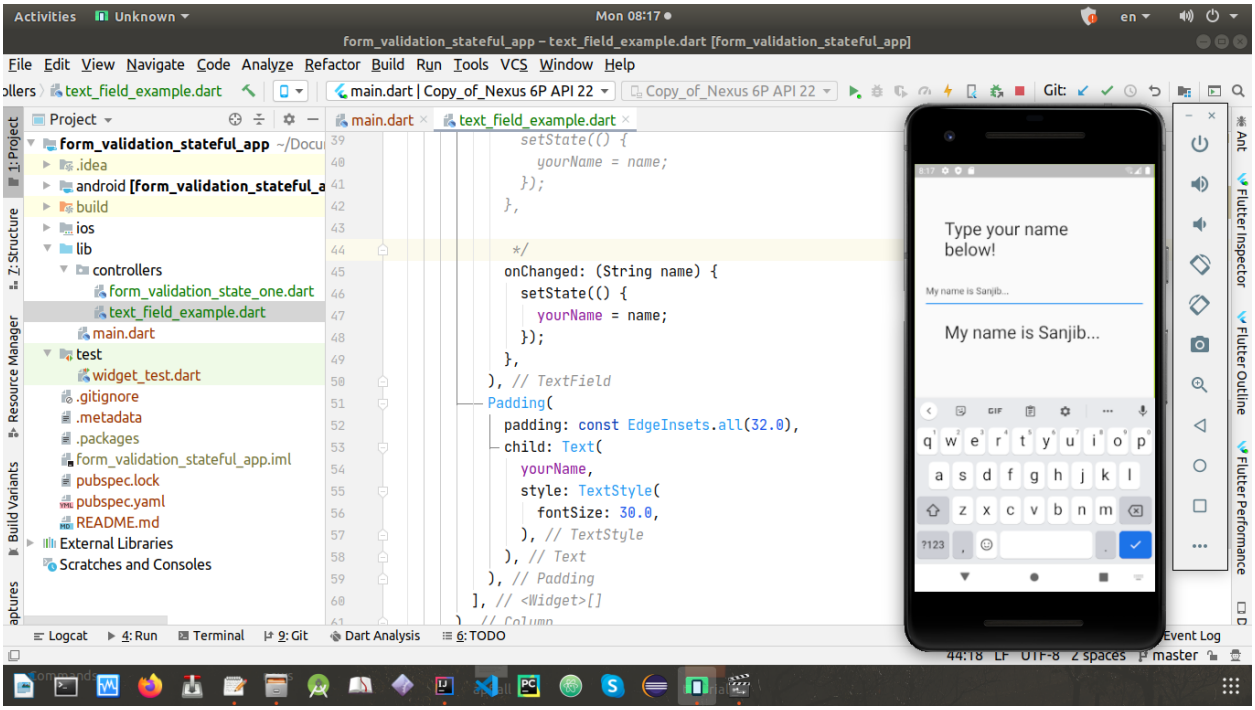


Figure 7.5 – While typing using TextField widget the text is being shown on the screen

If we want to use any button click to get the output, we can use the TextField widget as well. The next code snippet will give you an idea.

```
//code 7.3

import 'package:flutter/material.dart';

class TextFieldApp extends StatefulWidget {
@override
_TextFieldAppState createState() => _TextFieldAppState();
}

class _TextFieldAppState extends State<TextFieldApp> {

TextEditingController textOfName = TextEditingController();

String _displayText = '';

String displayAllSelectedValue() {
    String name = textOfName.text;
    String result = 'Name is: ${name}';
    return result;
}
```

```
21
22
23  @override
24  Widget build(BuildContext context) {
25      return Scaffold(
26      body: Container(
27          alignment: Alignment.center,
28          child: Padding(
29          padding: EdgeInsets.all(20.0),
30          child: ListView(
31              children: <Widget>[
32              TextField(
33                  keyboardType: TextInputType.text,
34                  controller: textOfName,
35                  style: TextStyle(
36                  fontSize: 16.0,
37                  color: Colors.blue,
38                  ),
39                  decoration: InputDecoration(
40                  labelText: 'Your Nmae',
41                  hintText: 'In text...',
42                  labelStyle: TextStyle(
43                      fontSize: 17.0,
44                      color: Colors.red,
45                  ),
46                  border: OutlineInputBorder(
47                      borderRadius: BorderRadius.circular(5.0),
48                  ),
49                  ),
50              ),
51              SizedBox(height: 10.0,),
52              Row(
53                  children: <Widget>[
54                  Container(
55                      width: 150.0,
56                      child: RaisedButton(
57                      color: Colors.white24,
58                      textColor: Colors.redAccent,
59                      child: Text('Press'),
60                      onPressed: () {
61                          setState(() {
62                          this._displayText = displayAllSelectedValue();
63                          });
```

```
                    },
                    ),
                ),
                ],
            ),
            SizedBox(height: 10.0,),
            Text(
                '${_displayText}',
                style: TextStyle(
                fontSize: 20.0,
                color: Colors.redAccent,
                ),
            ),
            ],
        ),
        ),
    ),
    );
}
}
```

Role of Controller in TextField Widget

However, there is a big change in the TextField widget; as we have to use a new property, a named parameter called ‘controller’. Now this controller will check what type of input we type using the TextField widget.

```
TextField(
                keyboardType: TextInputType.text,
                controller: textOfName,
                style: TextStyle(
                fontSize: 16.0,
                color: Colors.blue,
                ),
                decoration: InputDecoration(
                labelText: 'Your Nmae',
                hintText: 'In text...',
                labelStyle: TextStyle(
                    fontSize: 17.0,
                    color: Colors.red,
                ),
```

```
                border: OutlineInputBorder(
                    borderRadius: BorderRadius.circular(5.0),
                ),
                ),
            ),
```

Visibly there is a lot of change when we have described the TextField widget. We have added a controller name and we have added some styling. The function that we have used to display the text, used that controller object. TextEditingController textOfName = TextEditingController();

```
String _displayText = '';

String displayAllSelectedValue() {
    String name = textOfName.text;
    String result = 'Name is: ${name}';
    return result;
}
```

The above code shows us that because we use the controller object, we are in a better position to control the nature of user inputs. Instead only text, we can handle integer data type also.

The next code snippets and the associated images will give us a good idea of how it is being done. We have also used inside the code a new Stateful widget 'DropdownButton' that uses String value to give users a chance to select the correct input. Let us see the code and the associated images; after that we will discuss the code snippets in detail.

Let us first see the image, where the user is asked to give some inputs, such as name, age using the TextField widget. Besides, the user will choose the name of the city where she lives currently. For that we have used the 'DropdownButton'.

Above each TextField there are label text and inside it, there is hint text also.

As we have clicked the button, the user will be given the required output.

Beside the submit button, we have also placed a button called 'reset'; whenever it is pressed, every output will disappear from the screen and Flutter re-draws every widget. It is a little bit complex examples of state management, where we have not only used the 'setState()' but also override the 'initState()' function for the first time to clean the DropdownButton widget.

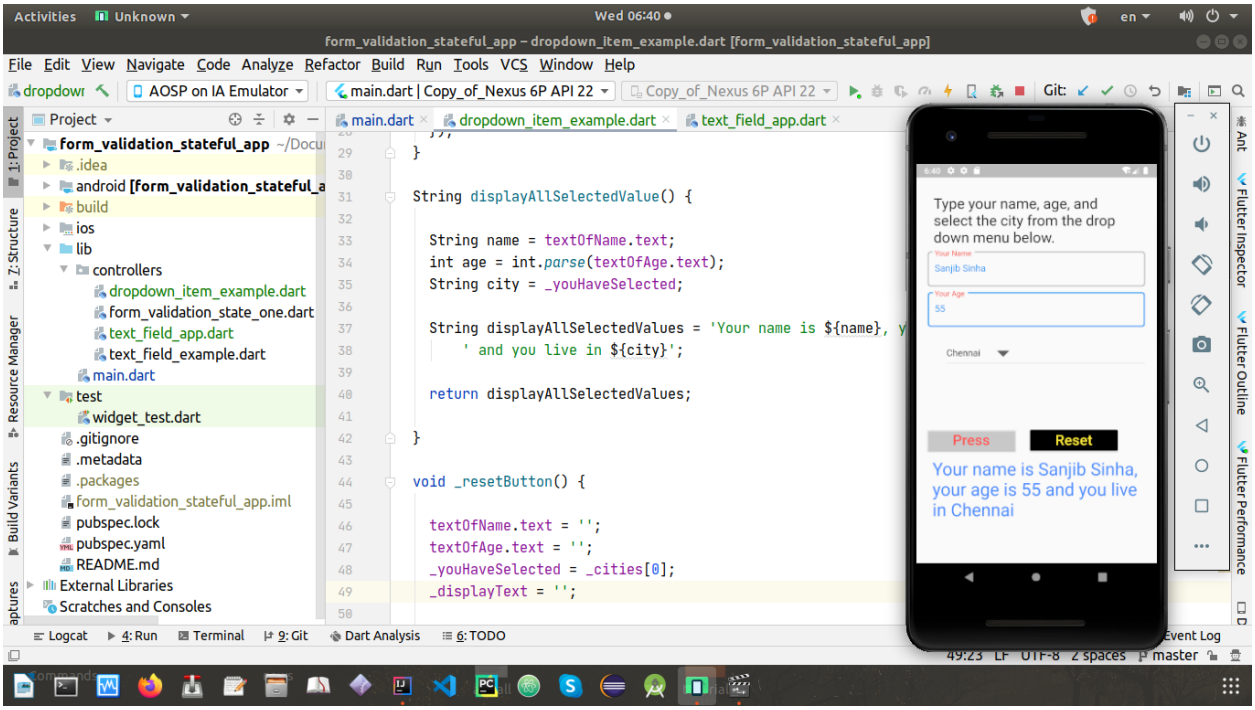


Figure 7.6 - A complex state management example

Let us see the code snippets first, and afterward we will see the next image where the reset button has been pressed.

```
//code 7.4

import 'package:flutter/material.dart';

class DropDownExample extends StatefulWidget {
@override
_DropDownExampleState createState() => _DropDownExampleState();
}

class _DropDownExampleState extends State<DropDownExample> {

String yourName = '';
String yourAge = '';
var _cities = ["Calcutta", "Delhi", "Mumbai", "Chennai", "Bangalore"];
String _youHaveSelected = '';
String _displayText = '';

@override
void initState() {
    super.initState();
```

```
    _youHaveSelected = _cities[0];
}

TextEditingController textOfAge  = TextEditingController();
TextEditingController textOfName  = TextEditingController();

void selectedDropDownItem(String theValueSelected) {
    setState(() {
    this._youHaveSelected = theValueSelected;
    });
}

String displayAllSelectedValue() {

    String name = textOfName.text;
    int age = int.parse(textOfAge.text);
    String city = _youHaveSelected;

    String displayAllSelectedValues = 'Your name is ${name}, your age is ${age}'
        ' and you live in ${city}';

    return displayAllSelectedValues;

}

void _resetButton() {

    textOfName.text = '';
    textOfAge.text = '';
    _youHaveSelected = _cities[0];
    _displayText = '';

}

@override
Widget build(BuildContext context) {
    return Scaffold(
    //resizeToAvoidBottomPadding: false,
    body: Container(
        margin: EdgeInsets.all(20.0),
        child: ListView(
        children: <Widget>[
            Padding(
```

```
64              padding: const EdgeInsets.all(10.0),
65              child: Text(
66                  'Type your name, age, and select the city from the drop down menu be\
67  low.',
68                  style: TextStyle(
69                  fontSize: 25.0,
70                  ),
71              ),
72              ),
73              TextField(
74              keyboardType: TextInputType.text,
75              controller: textOfName,
76              style: TextStyle(
77                  fontSize: 16.0,
78                  color: Colors.blue,
79              ),
80              decoration: InputDecoration(
81                  labelText: 'Your Name',
82                  hintText: 'In text...',
83                  labelStyle: TextStyle(
84                  fontSize: 17.0,
85                  color: Colors.red,
86                  ),
87                  border: OutlineInputBorder(
88                  borderRadius: BorderRadius.circular(5.0),
89                  ),
90              ),
91              ),
92              SizedBox(height: 10.0,),
93              TextField(
94              keyboardType: TextInputType.text,
95              controller: textOfAge,
96              style: TextStyle(
97                  fontSize: 16.0,
98                  color: Colors.blue,
99              ),
100             decoration: InputDecoration(
101                 labelText: 'Your Age',
102                 hintText: 'In number...',
103                 labelStyle: TextStyle(
104                 fontSize: 17.0,
105                 color: Colors.red,
106                 ),
```

```
107                 border: OutlineInputBorder(
108                 borderRadius: BorderRadius.circular(5.0),
109                 ),
110             ),
111             ),
112             SizedBox(height: 10.0,),
113             Padding(
114             padding: const EdgeInsets.only(left: 32.0, top: 10.0),
115             child: DropdownButton<String>(
116                 items: _cities.map((String nameOfCities) {
117                 return DropdownMenuItem<String>(
118                     value: nameOfCities,
119                     child: Text(nameOfCities),
120                 );
121                 }).toList(),
122                 onChanged: (String theValueSelected) {
123                 selectedDropDownItem(theValueSelected);
124                 },
125                 value: _youHaveSelected,
126                 iconSize: 50.0,
127             ),
128             ),
129             SizedBox(height: 100.0,),
130             Row(
131             children: <Widget>[
132                 Container(
133                 width: 150.0,
134                 child: RaisedButton(
135                     color: Colors.white24,
136                     textColor: Colors.redAccent,
137                     child: Text('Press', style: TextStyle(fontSize: 25.0),),
138                     onPressed: () {
139                     setState(() {
140                         this._displayText = displayAllSelectedValue();
141                     });
142                     },
143                 ),
144                 ),
145                 Container(width: 25.0,),
146                 Container(
147                 width: 150.0,
148                 child: RaisedButton(
149                     color: Colors.black,
```

```
                    textColor: Colors.yellow,
                    child: Text('Reset', style: TextStyle(fontSize: 25.0),),
                    onPressed: () {
                    setState(() {
                        _resetButton();
                    });
                    },
                ),
                ),
            ],
            ),
            Padding(
            padding: const EdgeInsets.all(8.0),
            child: Text(
                '${_displayText}',
                style: TextStyle(
                fontSize: 30.0,
                color: Colors.blueAccent,
                ),
            ),
            ),
        ],
        ),
    ),
    );
}
}
```

Let us see the initial state when the reset button has been pressed.

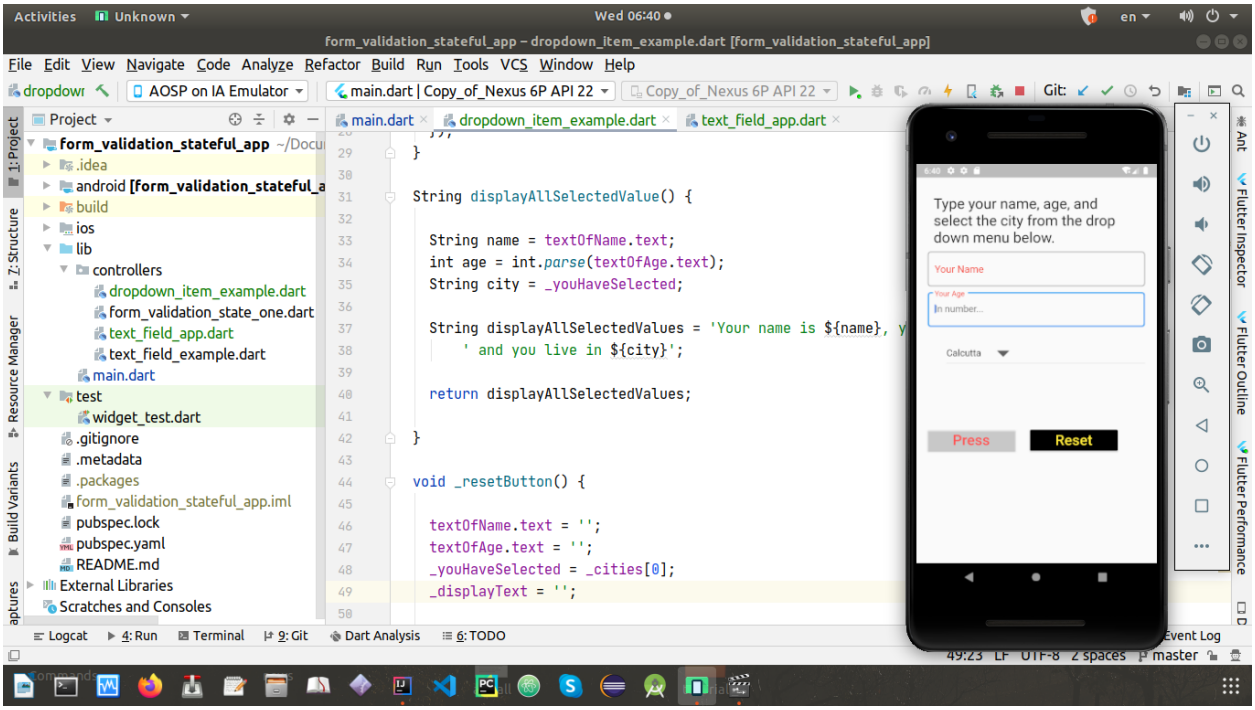


Figure 7.7 – After the reset button has been pressed

The following part of the code snippets has helped us to maintain this complex state management successfully. We have used the concepts of list and later in the

How List and Map used in StateFul DropdownButton Widget

DropdownButton widget used the concepts of map to get the Drop down Menu Item.

```
String yourName = '';
String yourAge = '';
var _cities = ["Calcutta", "Delhi", "Mumbai", "Chennai", "Bangalore"];
String _youHaveSelected = '';
String _displayText = '';

@override
void initState() {
    super.initState();
    _youHaveSelected = _cities[0];
}

TextEditingController textOfAge  = TextEditingController();
```

```
14 TextEditingController textOfName  = TextEditingController();
15
16 void selectedDropDownItem(String theValueSelected) {
17     setState(() {
18     this._youHaveSelected = theValueSelected;
19     });
20 }
21
22 String displayAllSelectedValue() {
23
24     String name = textOfName.text;
25     int age = int.parse(textOfAge.text);
26     String city = _youHaveSelected;
27
28     String displayAllSelectedValues = 'Your name is ${name}, your age is ${age}'
29         ' and you live in ${city}';
30
31     return displayAllSelectedValues;
32
33 }
34
35 void _resetButton() {
36
37     textOfName.text = '';
38     textOfAge.text = '';
39     _youHaveSelected = _cities[0];
40     _displayText = '';
41
42 }
43 …
44 child: DropdownButton<String>(
45                 items: _cities.map((String nameOfCities) {
46                 return DropdownMenuItem<String>(
47                     value: nameOfCities,
48                     child: Text(nameOfCities),
49                 );
50                 }).toList(),
51                 onChanged: (String theValueSelected) {
52                 selectedDropDownItem(theValueSelected);
53                 },
54                 value: _youHaveSelected,
55                 iconSize: 50.0,
56             ),
```

```
57              ),
```

As we have seen, managing state is not difficult, but we need to be careful to follow the correct design patterns. Google's bloc pattern is simple and user friendly. We can easily set the state and when necessary, intiState() method can be overridden to make some complex operations possible.

Finally, in this chapter, we will learn how to use validation, so the user will be prompted to fill the required fields properly.

How to Valiadate a Form using State Management

Let us see the code snippets, after that we will see the associated image; once that is done, we can discuss the code in parts.

```
1  //code 7.5
2  //form_validation_app.dart
3
4  import 'package:flutter/material.dart';
5
6  class FormValidationApp extends StatefulWidget {
7  @override
8  _FormValidationAppState createState() => _FormValidationAppState();
9  }
10
11 class _FormValidationAppState extends State<FormValidationApp> {
12
13 //we have initialized the form key with super class FormState
14 //in future, this key will be used to identify the form instance
15 var _formKey = GlobalKey<FormState>();
16
17 String yourName = '';
18 String yourAge = '';
19 var _cities = ["Calcutta", "Delhi", "Mumbai", "Chennai", "Bangalore"];
20 String _youHaveSelected = '';
21 String _displayText = '';
22
23 @override
24 void initState() {
25     super.initState();
26     _youHaveSelected = _cities[0];
27 }
28
```

```
TextEditingController textOfAge  = TextEditingController();
TextEditingController textOfName  = TextEditingController();

void selectedDropDownItem(String theValueSelected) {
    setState(() {
    this._youHaveSelected = theValueSelected;
    });
}

String displayAllSelectedValue() {

    String name = textOfName.text;
    int age = int.parse(textOfAge.text);
    String city = _youHaveSelected;

    String displayAllSelectedValues = 'Your name is ${name}, your age is ${age}'
        ' and you live in ${city}';

    return displayAllSelectedValues;

}

void _resetButton() {

    textOfName.text = '';
    textOfAge.text = '';
    _youHaveSelected = _cities[0];
    _displayText = '';

}

@override
Widget build(BuildContext context) {
    return Scaffold(
    //resizeToAvoidBottomPadding: false,
    //we have changed the previous container widget to Form
    //since Form does not allow margin, we need to add some padding around ListView
    body: Form(
        //later this key will act as an identifier
        //and it will let us know the current status of the form
        key: _formKey,
        child: Padding(
        padding: const EdgeInsets.all(8.0),
```

```
72          child: ListView(
73              children: <Widget>[
74              Padding(
75                  padding: const EdgeInsets.all(10.0),
76                  child: Text(
77                  'Type your name, age, and select the city from the drop down menu be\
78  low.',
79                  style: TextStyle(
80                      fontSize: 25.0,
81                  ),
82                  ),
83              ),
84              //we will change the TextField to TextFormField so that we can use the v\
85  alidator
86              TextFormField(
87                  keyboardType: TextInputType.text,
88                  controller: textOfName,
89                  validator: (String validationValue) {
90                  if (validationValue.isEmpty) {
91                      return 'Please fill up the form with correct input!';
92                  }
93                  },
94                  style: TextStyle(
95                  fontSize: 16.0,
96                  color: Colors.blue,
97                  ),
98                  //because we are using TextFormField, and use the validation
99                  // we can use customize the error style
100                 decoration: InputDecoration(
101                 labelText: 'Your Name',
102                 hintText: 'In text...',
103                 labelStyle: TextStyle(
104                     fontSize: 17.0,
105                     color: Colors.red,
106                 ),
107                 border: OutlineInputBorder(
108                     borderRadius: BorderRadius.circular(5.0),
109                 ),
110                 errorStyle: TextStyle(
111                     color: Colors.deepPurple,
112                     fontSize: 20.0,
113                 ),
114                 ),
```

```
115             ),
116             SizedBox(height: 10.0,),
117             TextFormField(
118                 keyboardType: TextInputType.text,
119                 controller: textOfAge,
120                 validator: (String validationValue) {
121                 if (validationValue.isEmpty) {
122                     return 'Please fill up the form with correct input!';
123                 }
124                 },
125                 style: TextStyle(
126                 fontSize: 16.0,
127                 color: Colors.blue,
128                 ),
129                 decoration: InputDecoration(
130                 labelText: 'Your Age',
131                 hintText: 'In number...',
132                 labelStyle: TextStyle(
133                     fontSize: 17.0,
134                     color: Colors.red,
135                 ),
136                 border: OutlineInputBorder(
137                     borderRadius: BorderRadius.circular(5.0),
138                 ),
139                 errorStyle: TextStyle(
140                     color: Colors.deepPurple,
141                     fontSize: 20.0,
142                 ),
143                 ),
144             ),
145             SizedBox(height: 10.0,),
146             Padding(
147                 padding: const EdgeInsets.only(left: 32.0, top: 10.0),
148                 child: DropdownButton<String>(
149                 items: _cities.map((String nameOfCities) {
150                     return DropdownMenuItem<String>(
151                     value: nameOfCities,
152                     child: Text(nameOfCities),
153                     );
154                 }).toList(),
155                 onChanged: (String theValueSelected) {
156                     selectedDropDownItem(theValueSelected);
157                 },
```

```
158                 value: _youHaveSelected,
159                 iconSize: 50.0,
160                 ),
161             ),
162             SizedBox(height: 100.0,),
163             Row(
164                 children: <Widget>[
165                 Container(
166                     width: 150.0,
167                     child: RaisedButton(
168                     color: Colors.white24,
169                     textColor: Colors.redAccent,
170                     child: Text('Press', style: TextStyle(fontSize: 25.0),),
171                     onPressed: () {
172                         setState(() {
173                         //if the form's current state validates, only then proceed
174                         if (_formKey.currentState.validate()) {
175                             this._displayText = displayAllSelectedValue();
176                         }
177                         });
178                     },
179                     ),
180                 ),
181                 Container(width: 25.0,),
182                 Container(
183                     width: 150.0,
184                     child: RaisedButton(
185                     color: Colors.black,
186                     textColor: Colors.yellow,
187                     child: Text('Reset', style: TextStyle(fontSize: 25.0),),
188                     onPressed: () {
189                         setState(() {
190                         _resetButton();
191                         });
192                     },
193                     ),
194                 ),
195                 ],
196             ),
197             Padding(
198                 padding: const EdgeInsets.all(8.0),
199                 child: Text(
200                 '${_displayText}',
```

```
                style: TextStyle(
                    fontSize: 30.0,
                    color: Colors.blueAccent,
                ),
                ),
            ),
            ],
        ),
        ),
    ),
    );
}
}
```

We have changed our old code (7.4) a little bit; all we have done is we have added some some extra functionalities and changed the Container widget to the Form widget. This Form widget has many other features, without which we could not have achieved what we wanted to do.

Let us see the image, and we will have an idea how this code works.

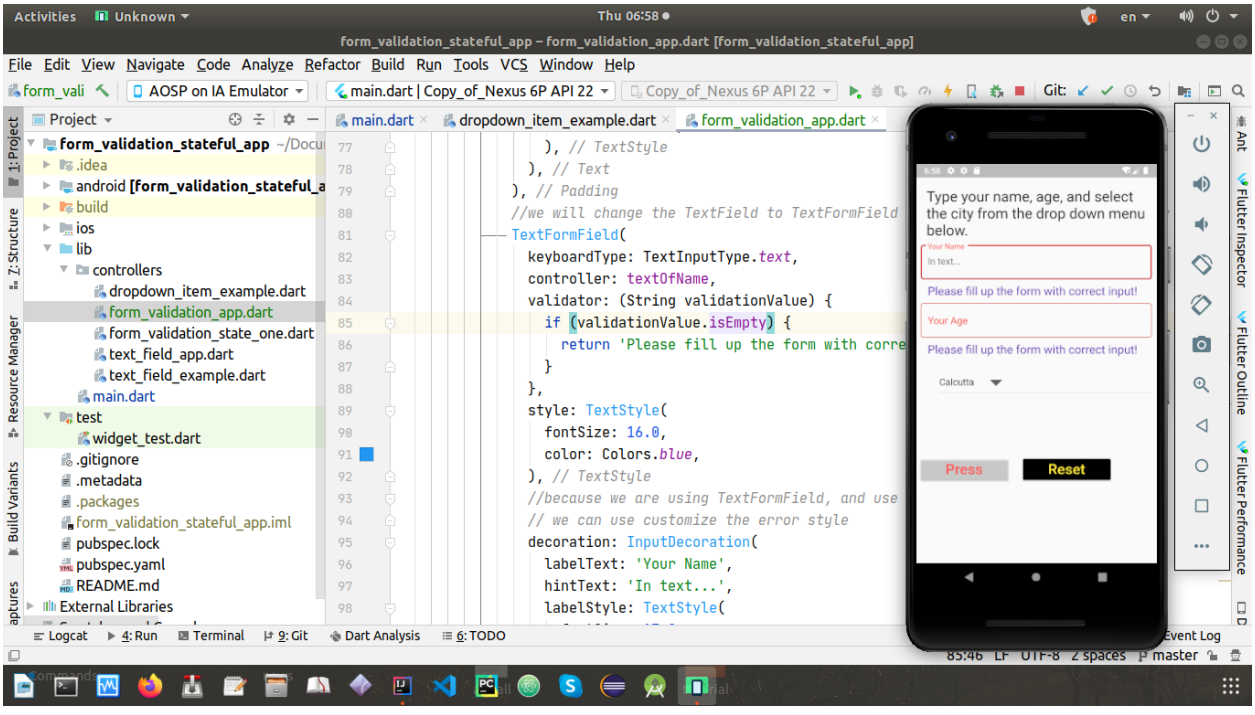


Figure 7.8 – How form validation works in state management

The major change is in the following section:

```
//we have changed the previous container widget to Form
    //since Form does not allow margin, we need to add some padding around ListView
    body: Form(
        //later this key will act as an identifier
        //and it will let us know the current status of the form
        key: _formKey,
…
```

The next big change in the TextField section.

```
//we will change the TextField to TextFormField so that we can use the validator
            TextFormField(
                keyboardType: TextInputType.text,
                controller: textOfName,
                validator: (String validationValue) {
                if (validationValue.isEmpty) {
                    return 'Please fill up the form with correct input!';
                }
                },
                style: TextStyle(
                fontSize: 16.0,
                color: Colors.blue,
                ),
                //because we are using TextFormField, and use the validation
                // we can use customize the error style
                decoration: InputDecoration(
                labelText: 'Your Name',
                hintText: 'In text...',
                labelStyle: TextStyle(
                    fontSize: 17.0,
                    color: Colors.red,
                ),
                border: OutlineInputBorder(
                    borderRadius: BorderRadius.circular(5.0),
                ),
                errorStyle: TextStyle(
                    color: Colors.deepPurple,
                    fontSize: 20.0,
                ),
                ),
            ),
….
```

In the comments we have mentioned why we are doing these changes. Every change is purposeful here, because we want to validate the Form properly.

The Form key plays a major role here. It holds on to the state of the form; for that reason, this following part is extremely important.

```
Form(
        //later this key will act as an identifier
        //and it will let us know the current status of the form
        key: _formKey,
….
```

It is clearly stated why we are using the key. Later, inside the RaisedButton widget, it plays the key role. Watch this code snippets:

```
child: RaisedButton(
                    color: Colors.white24,
                    textColor: Colors.redAccent,
                    child: Text('Press', style: TextStyle(fontSize: 25.0),),
                    onPressed: () {
                        setState(() {
                        //if the form's current state validates, only then proceed
                        if (_formKey.currentState.validate()) {
                            this._displayText = displayAllSelectedValue();
                        }
                        });
                    },
                    ),
                ),
...
```

Here the logic is also quite clear. If the form key's current state validates, only then the output will be given on the screen. Otherwise, it will give us errors that we have customized.

However, Google announced at Google I/O '19 that Provider is now its preferred package for state management. Provider is a package written in 2018 by Remi Rousselet. Because it is simple and flexible, we will also learn how to use Provider to manage state without rebuilding the whole UI widget tree.

Of course, you can still use others, because there are others, ans some of them is really good. But, keep in mind that Google recommends going with Provider.

In the next chapter, we will find the simplicity and flexibility of using Provider to manage state in a better way. We will also learn how to use Model-View-Controller design pattern to implement Provider.

Want to read more Flutter related Articles and resources?

For more Flutter related Articles and Resources[8]

[8] https://zerodotone.net

8. Provider: A recommended approach to manage State and Model-View-Controller Pattern

When an app is running, if we want something to exist in memory, we can call it 'state'. In the previous chapter, we have already discussed 'state' and learned a few tricks to manage it. However, that is an introduction. We need to understand the concept of 'state', because it is extremely important to build any type of complex app, that handles multiple screens, different variables, user sessions, etc. State can include anything – the app's assets, as we said, all the variables that the Flutter framework keeps about the UI, user sessions that can be shared in different parts of the app, etc.

Whenever we design an app, and start building it, we don't have to manage every state. Flutter framework takes care of a large sections, like textures. Despite that, we need some data to rebuild our UI at any moment in time. The simplest example is we press a button and the text changes on the screen. Again we press the restore button, and the text disappears. We need to provide the business logic so that it happens.

Consider a complex example, where a user adds an item to cart and that item remains at that cart as long as user is logged in. Notwithstanding, state is of two types – ephemeral and app state.

We know the meaning of the word ephemeral, it means short-lived. Some kind of state is very short-lived. We may contain it in a single widget. That is why it is also called local state. Suppose we want to show the current progress of a complex animation. Once it is done, the UI is rebuilt, and we don't want it anymore.

For that reason, we don't have to need any specialized state management techniques like 'Provider' for that. There are many other techniques as well, but in this chapter we will only learn Provider, because Google recommends it.

For ephemeral state management using setState() and a field inside the StatefulWidget's State class is enough, because, a single widget needs it, no other part of the device can access its single private variable. An app state or application state is not like that. We want to share the app state across many parts of our app, not only that, we may want it to keep between user sessions. In like manner, we can call it shared state.

To manage app state we can opt for several options. Nevertheless, Google recommends Provider, we will have a brief look at other options as well.

Different approaches to state management

As we have said before, Provider is the recommended approach. Provider helps you to manage state efficiently, in a very simple and it has great flexibility. We will learn that techniques in a minute.

Before that, let us see other approaches to manage state. Using setState() and a field inside the StatefulWidget's State class is another approach; yet that is good and recommended for the ephemeral state. This lower-level approach is made up when we create a new Flutter application.

InheritedWidget & InheritedModel approach is another lower-level approach that communicates between ancestors and children in the widget tree.

Redux is another approach that is familiar to the web developers. It is a state container approach, which is also very popular among Flutter developers.

BLoC is another stream and observable based patterns, in fact before Provider has stepped in, BLoC was very popular. Still the flutter community adores BLoC.

Otherwise we might use MobX or GetX approach; the first one is a popular library based conceptualization on observables and reactions, and the second one is a simplified reactive state management solution.

There are plenty of open source resources available to learn any one of them, thoroughly. In this chapter, we will learn only Provider, the state management recommended by Google, creator of Dart programming language and Flutter framework.

A Step by Step guide to use Provider

First thing first, we have add the dependency on provider to our 'pubspec.yaml' file.

```
// pubspec.yaml
# ...

dependencies:
flutter:
    sdk: flutter

provider: ^4.0.0
```

At the time of writing this book, provider package is 4 and above. We will always check the latest version.

The app state is something that we need to modify from many different places, and to do that we have to pass around a lot of callbacks; for a complex widget tree, it will be suicidal to replace several widgets again and again. To understand this mechanism we need to find out a solution that will not

disturb the widget tree as a whole, yet the app state will modify a few widgets deep down the tree. Suppose we need to modify one widget that has hundred widgets on top of it. Without disturbing top hundred widgets, we can successfully handle the app state using Provider.

Flutter has in-built mechanisms for widgets to provide data and services to their distant descendants, it means not just the immediate children, but any widgets below them.

Provider makes it possible to forget the callbacks and InheritedWidgets. We need to understand three primary concepts:

```
ChangeNotifier
ChangeNotifierProvider
Consumer
```

ChangeNotifier is an in-built class included in the Flutter SDK, this class notifies the listeners when any change in the state of the ChangeNotifier class takes place. Any widget having hundreds widgets on the top, can subscribe to its changes.

ChangeNotifierProvider, unlike ChangeNotifier, comes from the Provider package and it provides an instance of a ChangeNotifier to the widgets, which have already subscribed to it.

Where we should place the ChangeNotifierProvider? Just above the widgets that need to access it.

```
void main() {
runApp(
    ChangeNotifierProvider(
    create: (context) => AnyModel(),
    child: HomeApp(),
    ),
);
}
```

Or we can even use MultiProvider, if we want to use multiple classes.

```
void main() {
runApp(
    MultiProvider(
    providers: [
        ChangeNotifierProvider(create: (context) => FirstModel()),
        Provider(create: (context) => SecondClass()),
    ],
    child: HomeApp(),
    ),
);
}
```

Once our designed Model is provided to the desired widgets in our app through the ChangeNotifier-Provider declaration at the top, the Consumer widgets that have subscribed to the notifications can use it.

```
return Consumer<FirstModel>(
builder: (context, value, child) {
    return Text("The value : ${value.firstModelVariable}");
},
);
```

The first rule of using Consumer widget is we need to be specific about the type of the model that we want to access. Suppose, we want 'FirstModel', so we write Consumer<FirstModel>. If the generic type <FirstModel> is not specified, the Provider package cannot help us. The Provider package is based on 'type'. Therefore, we must mention the type.

The second most important rule is we must supply the 'builder' argument of the Consumer widget. This is the only required argument of the Consumer widget. Whenever in the model class ChangeNotifier changes, the builder argument is called. Let us try to understand what is happening. Whenever the ChangeNotifier changes, the method notifyListeners() is called, and at the same time, all the builder methods of all the corresponding Consumer widgets are called.

The 'builder' is called with three arguments, the first one is quite familiar, 'context'; we get it in every build method. The second argument 'value' is the instance of the ChangeNotifier. Using that instance we can define the app state, and along with it, we can also use the data in the model according to our requirement.

The role of the third argument 'child' is quite interesting. Suppose we have a large widget subtree under our Consumer that does not change when our model changes. We can also get it through the builder argument 'child'. We have done enough talking, tried to understand the interaction between Provider, and Consumer. Nonetheless, we won't understand this concepts unless we try to implement them.

Let us start with a very simple counter model. Through Provider, we will change the counter number. We have two buttons – Increase and Decrease (Figure 8.1). Imagine a number line, using these buttons, we can either move towards the right side (positive), or towards the left side (negative).

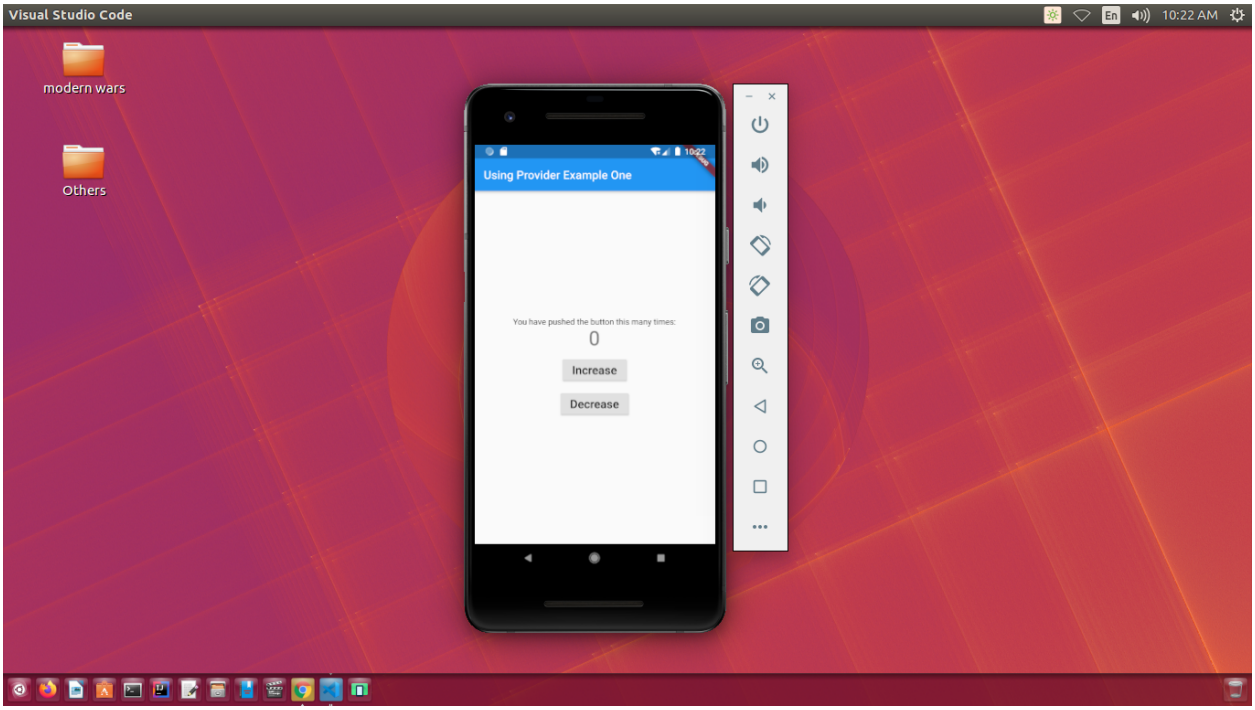


Figure 8.1 – Simple Provider example

The next two images will show you how we have increased the value and decreased the value tapping these two buttons respectively.

But before that we need to see the code and try to understand how we have used the Provider package.

```
// code 8.1

import 'package:flutter/widgets.dart';

/// using the mixin concept of dart that we have discussed
/// in our previous chapter
class CountingTheNumber with ChangeNotifier {
int value = 0;
void incrementTheValue() {
    value++;
    notifyListeners();
}

void decreaseValue() {
    value--;
    notifyListeners();
}
```

```
}
```

The above code snippets is quite simple. This is our model class through which we want to manage the state of the counter in a ChangeNotifier.

Next, we need to use the ChangeNotifierProvider in the right place.

Because we need to call two methods, using Consumer is wasteful. We don't want to change the whole UI with the help of our model data.

That is why we will use another concept - 'Provider.of', instead of using Consumer.

```
// code 8.2

import 'package:flutter/cupertino.dart';
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';

import 'counter_class.dart';

class MyApp extends StatelessWidget {
// This widget is the root of your application.
@override
Widget build(BuildContext context) {
    return MaterialApp(
    title: 'Flutter Demo',
    theme: ThemeData(
        primarySwatch: Colors.blue,
        visualDensity: VisualDensity.adaptivePlatformDensity,
    ),
    home: ChangeNotifierProvider<CountingTheNumber>(
        // it will not redraw the whole widget tree anymore
        create: (BuildContext context) => CountingTheNumber(),
        child: MyHomePage()),
    );
}
}

class MyHomePage extends StatelessWidget {
/*
MyHomePage({Key key, this.title}) : super(key: key);

final String title;
*/
```

```
33
34  @override
35  Widget build(BuildContext context) {
36      final counter = Provider.of<CountingTheNumber>(context);
37      return Scaffold(
38      appBar: AppBar(
39          title: Text('Using Provider Example One'),
40      ),
41      body: Center(
42          child: Column(
43          mainAxisAlignment: MainAxisAlignment.center,
44          children: <Widget>[
45              Text(
46              'You have pushed the button this many times:',
47              ),
48              // only Text widget listens to the notification
49              Text(
50              '${counter.value}',
51              style: Theme.of(context).textTheme.headline4,
52              ),
53              SizedBox(
54              height: 10.0,
55              ),
56              RaisedButton(
57              onPressed: () => counter.incrementTheValue(),
58              child: Text(
59                  'Increase',
60                  style: TextStyle(
61                  fontSize: 20.0,
62                  ),
63              ),
64              ),
65              SizedBox(
66              height: 10.0,
67              ),
68              RaisedButton(
69              onPressed: () => counter.decreaseValue(),
70              child: Text(
71                  'Decrease',
72                  style: TextStyle(
73                  fontSize: 20.0,
74                  ),
75              ),
```

```
            ),
        ],
        ),
    ),
    // This trailing comma makes auto-formatting nicer for build methods.
    );
}
}
```

Now, we can run the app and by tapping two buttons change the value. Before that, let us have a close look at some parts of the above code.

```
final counter = Provider.of<CountingTheNumber>(context);
```

'Provider.of', just like Consumer needs to know the type of the model. We need to specify the model 'CountingTheNumber'. Now using the 'counter' we have accessed the model data.

```
Text(
            '${counter.value}',
            style: Theme.of(context).textTheme.headline4,
            ),
…
RaisedButton(
            onPressed: () => counter.incrementTheValue(),
            child: Text(
                'Increase',
                style: TextStyle(
                fontSize: 20.0,
                ),
            ),
            ),
…
RaisedButton(
            onPressed: () => counter.decreaseValue(),
            child: Text(
                'Decrease',
                style: TextStyle(
                fontSize: 20.0,
                ),
            ),
            ),
```

The next step is running the app.

```
// code 8.3

import 'package:flutter/material.dart';
import 'utilities/first_provider_example.dart';

void main() {
runApp(MyApp());
}
```

Now we can tap the increase button (Figure 8.2).

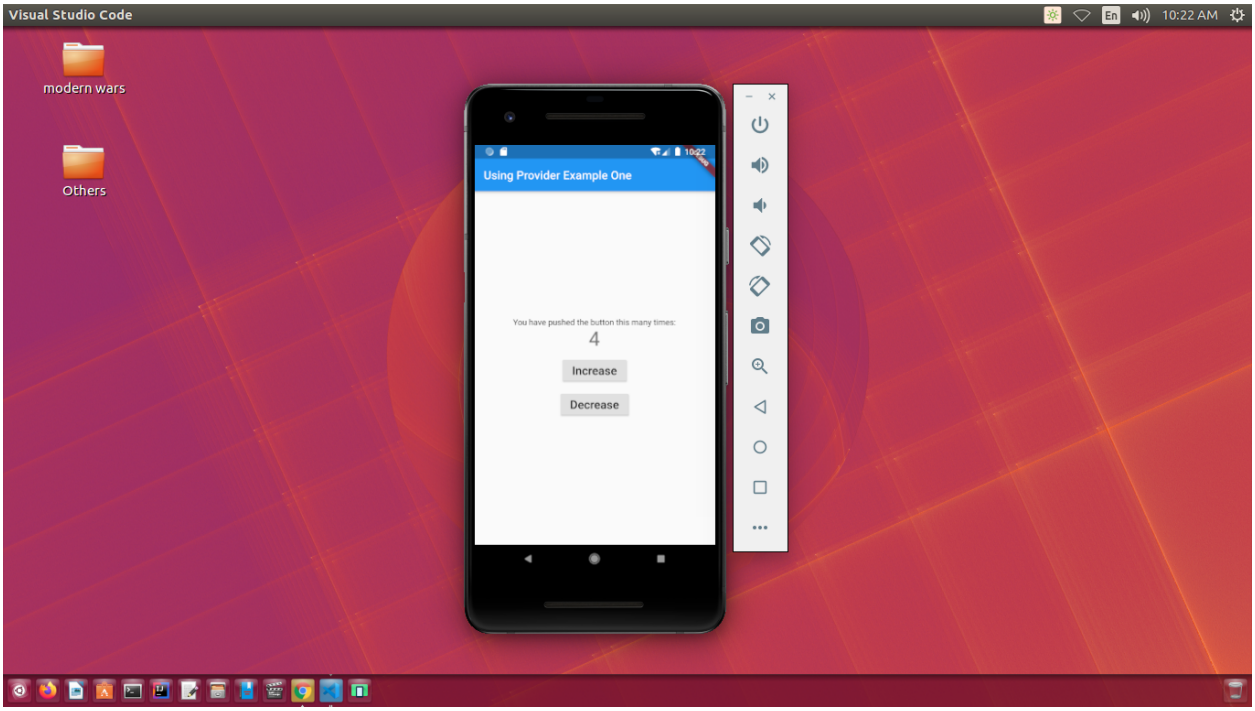


Figure 8.2 – We have tapped the increase button 4 times

After that, we can run the app once again, and it turns the counter value to 0. Now, we can test the decrease button (Figure 8.3).

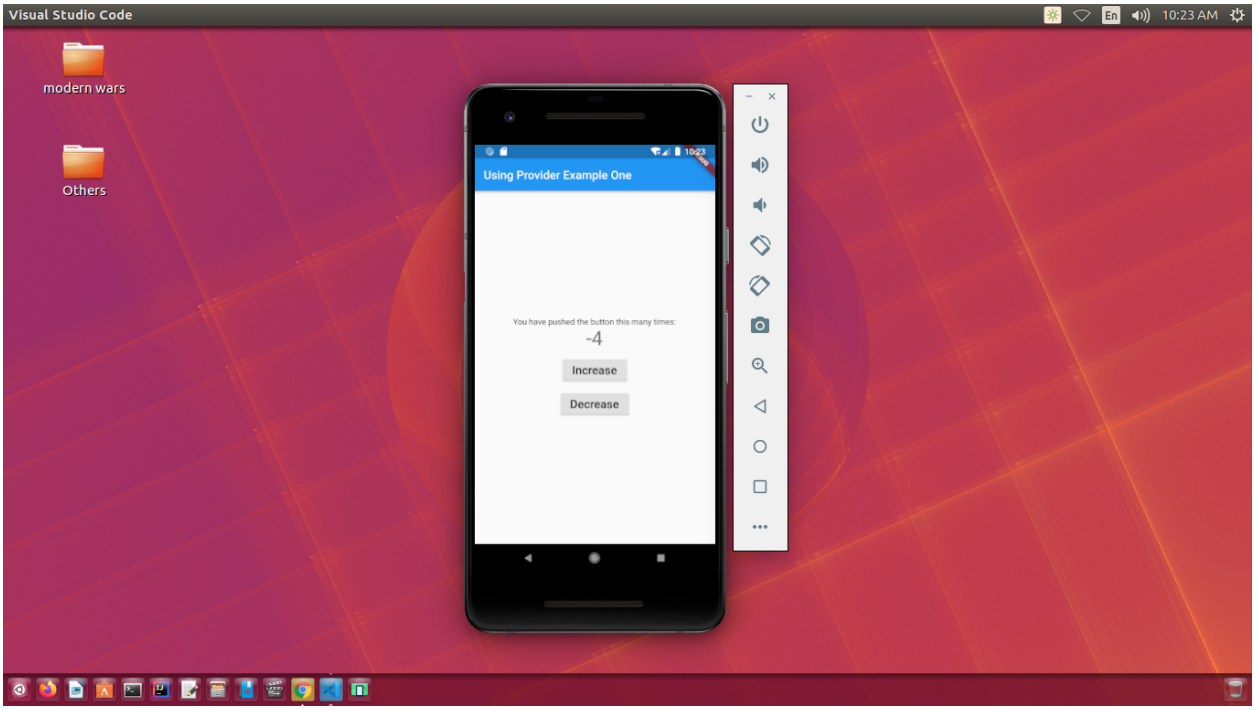


Figure 8.3 – Tapping the decrease button

The above code snippets give us an idea of how Provider package works.

Now we will take closer looks and use multi Providers and multi models to understand this process. This time we will use Consumer.

Let us start with an image. We have extended our old code added a few more generic models.

Now we can press the counter button, and besides, we will press a button to change the text below. After that, we can also press the restore button to clear that data and give an output of that.

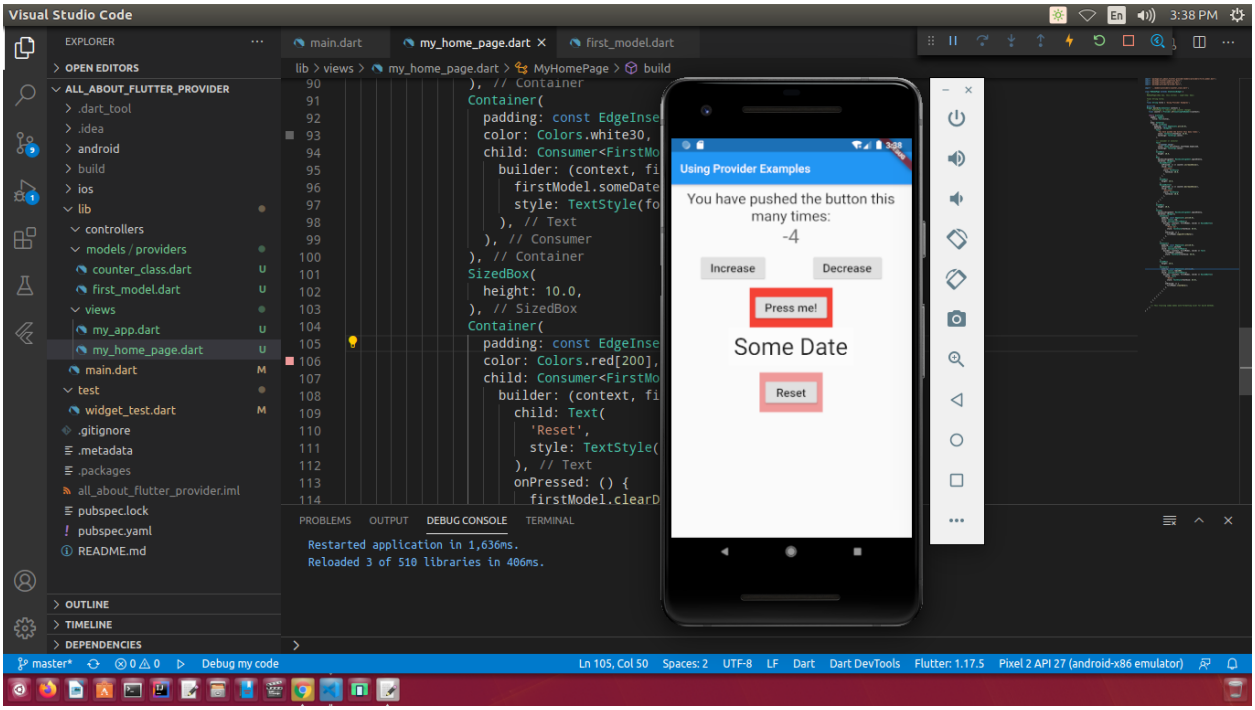


Figure 8.4 – Provider example with many models

In the above image, it is evident that we have pressed the decrease button 4 times, however, the default text data - 'Some Data', has not been affected.

Let us see the code:

```
//code 8.4

//main.dart

import 'models/providers/first_model_provider.dart';

import 'models/providers/counter_model_provider.dart';
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import 'models/providers/second_model_provider.dart';
import 'views/my_app.dart';

void main() {
runApp(MultiProvider(
    providers: [
    ChangeNotifierProvider(
        create: (context) => CountingTheNumber(),
    ),
```

```
    ChangeNotifierProvider(
        create: (context) => FirstModelProvider(),
    ),
    ],
    child: MyApp(),
));
}


// first_model_provider.dart

import 'package:flutter/widgets.dart';

class FirstModelProvider with ChangeNotifier {
String someDate = 'Some Date';

void supplyFirstData() {
    someDate = 'Data Changed!';
    print(someDate);
    notifyListeners();
}

void clearData() {
    someDate = 'Data Cleared!';
    print(someDate);
    notifyListeners();
}
}


// my_home_page.dart

import 'package:all_about_flutter_provider/models/providers/first_model_provider.dar\
t';
import 'package:all_about_flutter_provider/models/providers/second_model_provider.da\
rt';
import 'package:flutter/cupertino.dart';
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';

import '../models/providers/counter_model_provider.dart';

class MyHomePage extends StatelessWidget {
```

```
/*
MyHomePage({Key key, this.title}) : super(key: key);

final String title;
*/
final String title = 'Using Provider Examples';

@override
Widget build(BuildContext context) {
    /// MyHomePage is rebuilt when counter changes
    final counter = Provider.of<CountingTheNumber>(context);

    return Scaffold(
    appBar: AppBar(
        title: Text(title),
    ),
    body: SafeArea(
        child: ListView(
        padding: const EdgeInsets.all(10.0),
        children: <Widget>[
            Text(
            'You have pushed the button this many times:',
            style: TextStyle(fontSize: 25.0),
            textAlign: TextAlign.center,
            ),

            /// consumer or selector
            Text(
            '${counter.value}',
            style: Theme.of(context).textTheme.headline4,
            textAlign: TextAlign.center,
            ),
            SizedBox(
            height: 10.0,
            ),
            Row(
            mainAxisAlignment: MainAxisAlignment.spaceEvenly,
            children: <Widget>[
                RaisedButton(
                onPressed: () => counter.increaseValue(),
                child: Text(
                    'Increase',
                    style: TextStyle(
```

```
                    fontSize: 20.0,
                    ),
                ),
                ),
                SizedBox(
                height: 10.0,
                ),
                RaisedButton(
                onPressed: () => counter.decreaseValue(),
                child: Text(
                    'Decrease',
                    style: TextStyle(
                    fontSize: 20.0,
                    ),
                ),
                ),
            ],
            ),
            SizedBox(
            height: 10.0,
            ),
            Column(
            mainAxisAlignment: MainAxisAlignment.spaceEvenly,
            children: <Widget>[
                Container(
                padding: const EdgeInsets.all(10.0),
                color: Colors.red,
                child: Consumer<FirstModelProvider>(
                    builder: (context, firstModelProvider, child) =>
                        RaisedButton(
                    child: Text(
                        'Press me!',
                        style: TextStyle(fontSize: 20.0),
                    ),
                    onPressed: () {
                        firstModelProvider.supplyFirstData();
                    },
                    ),
                ),
                ),
                Container(
                padding: const EdgeInsets.all(10.0),
                color: Colors.white30,
```

```
148                 child: Consumer<FirstModelProvider>(
149                     builder: (context, firstModelProvider, child) => Text(
150                     firstModelProvider.someDate,
151                     style: TextStyle(fontSize: 40.0),
152                     ),
153                 ),
154                 ),
155                 SizedBox(
156                 height: 10.0,
157                 ),
158                 Container(
159                 padding: const EdgeInsets.all(10.0),
160                 color: Colors.red[200],
161                 child: Consumer<FirstModelProvider>(
162                     builder: (context, firstModelProvider, child) =>
163                         RaisedButton(
164                     child: Text(
165                         'Reset',
166                         style: TextStyle(fontSize: 20.0),
167                     ),
168                     onPressed: () {
169                         firstModelProvider.clearData();
170                     },
171                     ),
172                 ),
173                 ),
174             ],
175             ),
176         ],
177         ),
178     ),
179
180     /// This trailing comma makes auto-formatting nicer for build methods.
181     );
182 }
183 }
```

In the above code, we have used two Providers, inside the main() function.

```
runApp(MultiProvider(
    providers: [
    ChangeNotifierProvider(
        create: (context) => CountingTheNumber(),
    ),
    ChangeNotifierProvider(
        create: (context) => FirstModelProvider(),
    ),
    ],
    child: MyApp(),
));
```

Along with the 'CountingTheNumber' model, we have used a new 'ChangeNotifier' model - 'FirstModelProvider' class. And finally, inside the 'MyHomePage' widget, we have used the Consumer concepts.

```
child: Consumer<FirstModelProvider>(
                    builder: (context, firstModelProvider, child) =>
                        RaisedButton(
                    child: Text(
                        'Press me!',
                        style: TextStyle(fontSize: 20.0),
                    ),
                    onPressed: () {
                        firstModelProvider.supplyFirstData();
                    },
                    ),
                ),
```

Because this Consumer's builder argument returns a RaisedButton() widget, we have used the onPressed() argument to call one of model methods. It gives us the next figure (Figure 8.5).

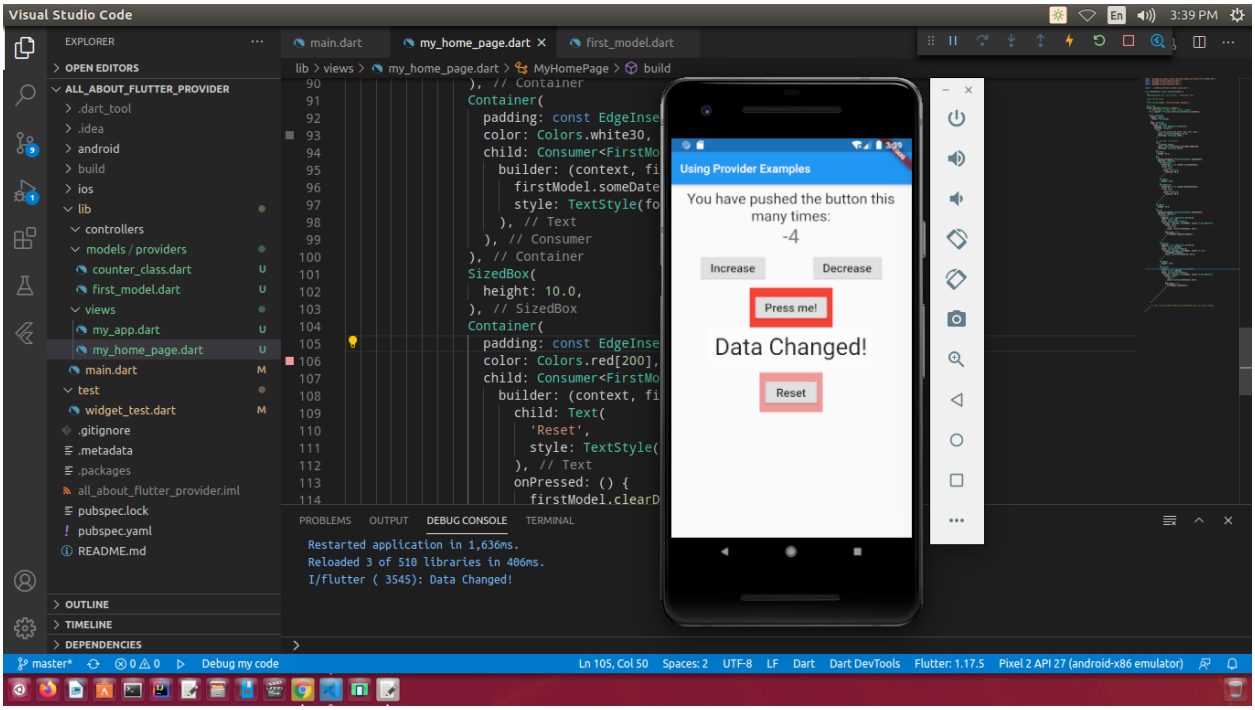


Figure 8.5 – The 'Press me' button has been pressed and the value of the model class has also been changed

If we click the 'Reset' button, the data has been cleared. The following figure (Figure 8.5) shows that display of the screen.

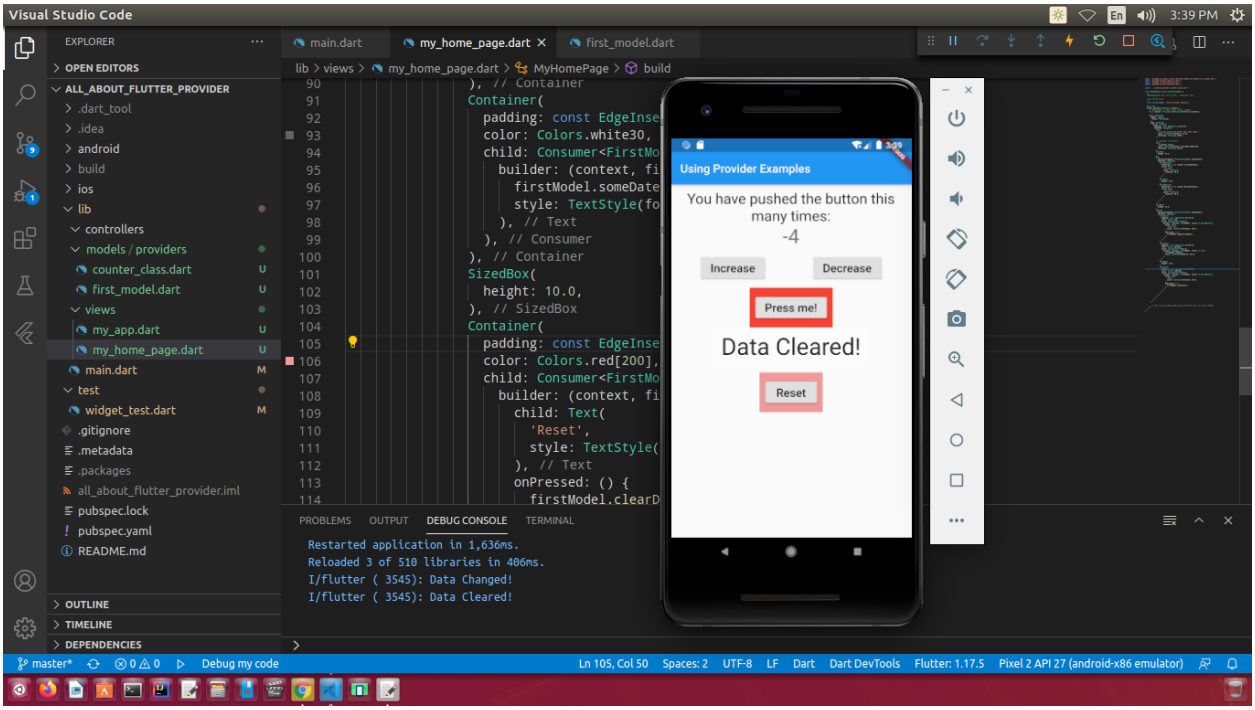


Figure 8.6 – We have pressed the 'Reset' button, and the corresponding value of model class is displayed

Now we are going to add another model class in the next code snippets. It will add another button that will display the first name.

```
//code 8.5

// main.dart

import 'models/providers/first_model_provider.dart';

import 'models/providers/counter_model_provider.dart';
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import 'models/providers/second_model_provider.dart';
import 'views/my_app.dart';

void main() {
runApp(MultiProvider(
    providers: [
    ChangeNotifierProvider(
        create: (context) => CountingTheNumber(),
    ),
    ChangeNotifierProvider(
        create: (context) => FirstModelProvider(),
```

```
    ),
    ChangeNotifierProvider(
        create: (context) => SecondModelProvider(),
    ),
    ],
    child: MyApp(),
));
}


// second_model_provider.dart

import 'package:flutter/widgets.dart';

class SecondModelProvider with ChangeNotifier {
String name = 'Some Name';
int age = 0;

void getFirstName() {
    name = 'Json';
    print(name);
    notifyListeners();
}
}


// my_home_page.dart

import 'package:all_about_flutter_provider/models/providers/first_model_provider.dar\
t';
import 'package:all_about_flutter_provider/models/providers/second_model_provider.da\
rt';
import 'package:flutter/cupertino.dart';
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';

import '../models/providers/counter_model_provider.dart';

class MyHomePage extends StatelessWidget {
/*
MyHomePage({Key key, this.title}) : super(key: key);

final String title;
```

```
64  */
65  final String title = 'Using Provider Examples';
66
67  @override
68  Widget build(BuildContext context) {
69      /// MyHomePage is rebuilt when counter changes
70      final counter = Provider.of<CountingTheNumber>(context);
71
72      return Scaffold(
73      appBar: AppBar(
74          title: Text(title),
75      ),
76      body: SafeArea(
77          child: ListView(
78          padding: const EdgeInsets.all(10.0),
79          children: <Widget>[
80              Text(
81              'You have pushed the button this many times:',
82              style: TextStyle(fontSize: 25.0),
83              textAlign: TextAlign.center,
84              ),
85
86              /// consumer or selector
87              Text(
88              '${counter.value}',
89              style: Theme.of(context).textTheme.headline4,
90              textAlign: TextAlign.center,
91              ),
92              SizedBox(
93              height: 10.0,
94              ),
95              Row(
96              mainAxisAlignment: MainAxisAlignment.spaceEvenly,
97              children: <Widget>[
98                  RaisedButton(
99                  onPressed: () => counter.increaseValue(),
100                 child: Text(
101                     'Increase',
102                     style: TextStyle(
103                     fontSize: 20.0,
104                     ),
105                 ),
106                 ),
```

```
107                 SizedBox(
108                 height: 10.0,
109                 ),
110                 RaisedButton(
111                 onPressed: () => counter.decreaseValue(),
112                 child: Text(
113                     'Decrease',
114                     style: TextStyle(
115                     fontSize: 20.0,
116                     ),
117                 ),
118                 ),
119             ],
120             ),
121             SizedBox(
122             height: 10.0,
123             ),
124             Column(
125             mainAxisAlignment: MainAxisAlignment.spaceEvenly,
126             children: <Widget>[
127                 Container(
128                 padding: const EdgeInsets.all(10.0),
129                 color: Colors.red,
130                 child: Consumer<FirstModelProvider>(
131                     builder: (context, firstModelProvider, child) =>
132                         RaisedButton(
133                     child: Text(
134                         'Press me!',
135                         style: TextStyle(fontSize: 20.0),
136                     ),
137                     onPressed: () {
138                         firstModelProvider.supplyFirstData();
139                     },
140                     ),
141                 ),
142                 ),
143                 Container(
144                 padding: const EdgeInsets.all(10.0),
145                 color: Colors.white30,
146                 child: Consumer<FirstModelProvider>(
147                     builder: (context, firstModelProvider, child) => Text(
148                     firstModelProvider.someDate,
149                     style: TextStyle(fontSize: 40.0),
```

```
150                     ),
151                 ),
152                 ),
153                 SizedBox(
154                 height: 10.0,
155                 ),
156                 Container(
157                 padding: const EdgeInsets.all(10.0),
158                 color: Colors.red[200],
159                 child: Consumer<FirstModelProvider>(
160                     builder: (context, firstModelProvider, child) =>
161                         RaisedButton(
162                     child: Text(
163                         'Reset',
164                         style: TextStyle(fontSize: 20.0),
165                     ),
166                     onPressed: () {
167                         firstModelProvider.clearData();
168                     },
169                     ),
170                 ),
171                 ),
172                 SizedBox(
173                 height: 10.0,
174                 ),
175                 Container(
176                 padding: const EdgeInsets.all(10.0),
177                 color: Colors.white30,
178                 child: Consumer<SecondModelProvider>(
179                     builder: (context, secondModel, child) => Text(
180                     secondModel.name,
181                     style: TextStyle(fontSize: 40.0),
182                     ),
183                 ),
184                 ),
185                 SizedBox(
186                 height: 10.0,
187                 ),
188                 Container(
189                 padding: const EdgeInsets.all(10.0),
190                 color: Colors.red[200],
191                 child: Consumer<SecondModelProvider>(
192                     builder: (context, secondModel, child) => RaisedButton(
```

```
                    child: Text(
                        'Get First Name',
                        style: TextStyle(fontSize: 20.0),
                    ),
                    onPressed: () {
                        secondModel.getFirstName();
                    },
                    ),
                ),
                ),
            ],
            ),
        ],
        ),
    ),

    /// This trailing comma makes auto-formatting nicer for build methods.
    );
}
}
```

This part of the code has handled the Consumer section. Therefore, let us check that part first.

```
Container(
                padding: const EdgeInsets.all(10.0),
                color: Colors.white30,
                child: Consumer<SecondModelProvider>(
                    builder: (context, secondModel, child) => Text(
                    secondModel.name,
                    style: TextStyle(fontSize: 40.0),
                    ),
                ),
                ),
                SizedBox(
                height: 10.0,
                ),
                Container(
                padding: const EdgeInsets.all(10.0),
                color: Colors.red[200],
                child: Consumer<SecondModelProvider>(
                    builder: (context, secondModel, child) => RaisedButton(
                    child: Text(
                        'Get First Name',
```

```
                        style: TextStyle(fontSize: 20.0),
                    ),
                    onPressed: () {
                        secondModel.getFirstName();
                    },
                    ),
                ),
                ),
```

We are able to add another feature of state management through Provider. The second model Provider is a simple class.

```
// second_model_provider.dart

import 'package:flutter/widgets.dart';

class SecondModelProvider with ChangeNotifier {
String name = 'Some Name';
int age = 0;

void getFirstName() {
    name = 'Json';
    print(name);
    notifyListeners();
}
}
```

The next figure (Figure 8.7) will show how Provider and Consumer work together. First, we have pressed the decrease button for 3 times. Next, we have pressed the 'Press me' button, and the 'Data Changed'. After that, finally, we have pressed the 'Get First Name' button, and the name appears on the screen. Each Consumer widget has persisted its state, one button-press does not affect the other. The changed-data stays on the screen.

Before concluding this chapter, we will learn how we can separate business logic, application logic and screen-view.

To do that, we will keep our models inside the 'model' folder and keep our business logic there. We will keep our application logic inside the 'controller' folder, and finally we get the screen-view inside the 'view' folder.

Model-View-Controller Patterns

First of all, we need to update pubspec.yaml, because we want some special fonts to be displayed.

```
//code 8.6

dependencies:
flutter:
    sdk: flutter
provider: ^4.3.2

# To add assets to your application, add an assets section, like this:
assets: [images/]

fonts:
#   - family: Schyler
#     fonts:
#       - asset: fonts/Schyler.ttf
#       - asset: fonts/Schyler-Italic.ttf
#         style: italic
    - family: Trajan Pro
    fonts:
        - asset: fonts/Trajan Pro Regular.ttf
#       - asset: fonts/TrajanPro_Bold.ttf
#         weight: 700
    - family: Sacramento
    fonts:
- asset: fonts/Sacramento-Regular.ttf
```

Next, we need two different models, ChangeNotifier, in our models folder. The first one is the following 'FirstModel' class.

```
//code 8.7

model/first_model.dart

import 'package:flutter/widgets.dart';

class FirstModel with ChangeNotifier {
String name = 'name';
void changeName() {
    name = 'Name Changed!';
    print(name);
    notifyListeners();
}

void clearName() {
```

```
    name = ' ';
    print(name);
    notifyListeners();
}
}
```

And the second model class is the 'MobileModel' that has a list of selected colors of which we will choose one for the background, and another for the mobile. We will display the mobile color on the foreground, and the background will be different. Pressing the icon of the respective mobile will change the color of both – foreground and background. At the same time a text will be displayed to make us aware that foreground and background colors have been changed.

```
//code 8.8

model/mobile_model.dart

import 'package:flutter/material.dart';
import 'package:flutter/widgets.dart';

class MobileModel with ChangeNotifier {
String backgroundColorOfFirst = 'Background';
String mobileColorOfFirst = 'Mobile';
String backgroundColorOfSecond = 'Background';
String mobileColorOfSecond = 'Mobile';
List<Color> selection = [
    Colors.yellow,
    Colors.blue,
    Colors.orange,
    Colors.pinkAccent,
    Colors.green,
    Colors.limeAccent,
];

void changeColorToPurple() {
    backgroundColorOfFirst = 'Background \n Purle';
    mobileColorOfFirst = 'Mobile \n White.';
    selection[0] = Colors.purple;
    selection[4] = Colors.white;
    notifyListeners();
}

void changeColorToRed() {
    backgroundColorOfSecond = 'Background \n Black';
```

```
32      mobileColorOfSecond = 'Mobile \n Red.';
33      selection[1] = Colors.black;
34      selection[5] = Colors.red;
35      notifyListeners();
36  }
37
38  void restoreOldColorOfFirstMobile() {
39      backgroundColorOfFirst = 'Background \n Yellow';
40      mobileColorOfFirst = 'Mobile \n Green.';
41      selection[0] = Colors.yellow;
42      selection[4] = Colors.green;
43      notifyListeners();
44  }
45
46  void restoreOldColorOfSecondMobile() {
47      backgroundColorOfSecond = 'Background \n Blue';
48      mobileColorOfSecond = 'Mobile \n Limeaccent.';
49      selection[1] = Colors.blue;
50      selection[5] = Colors.limeAccent;
51      notifyListeners();
52  }
53  }
```

The model classes are the sources of date. That data should be displayed on the screen-view. Not only that, that data must be changed on the tap of the icon.

Therefore, we need some subscribers or Consumers who will get that data and pass them to the view accordingly. Who will control that? The controllers. The controller will stay between model and view; the controllers' job is simple, it will play the role of the communicator who will manage the communication between model and view.

The data-source or model does not know where its data are going. The view does not know where from the data are coming. The controller knows everything. It controls every operation.

We have many controller widgets that will control different types of operations, such as one will control the foreground color, another will change background color,one controller will manage the text display, another will restore the value again, etc. Even we have some controllers that will also decide what type of text style we will follow.

We have kept those controllers in two separate files inside 'controller' folder. One controller file is mobile specific. Another is page specific. The mobile specific controllers are as follows:

```dart
// code 8.9

// controller/mobile_controller.dart

import 'package:first_flutter_app/model/mobile_model.dart';
import 'package:flutter/material.dart';
import 'package:flutter/widgets.dart';
import 'package:provider/provider.dart';

Widget changeColorButtonToPurple() => Column(
    children: [
        Container(
        padding: const EdgeInsets.all(10.0),
        child: Consumer<MobileModel>(
            builder: (context, value, child) => Container(
            padding: const EdgeInsets.all(15.0),
            child: FloatingActionButton(
                backgroundColor: value.selection[0],
                onPressed: () {
                value.changeColorToPurple();
                },
                child: Icon(
                Icons.mobile_screen_share,
                color: value.selection[4],
                ),
            ),
            ),
        ),
        ),
        Divider(
        thickness: 2.0,
        ),
        Consumer<MobileModel>(
        builder: (context, value, _) => Text(
            value.backgroundColorOfFirst,
            style: TextStyle(
            fontFamily: 'Trajan Pro',
            fontSize: 20.0,
            fontWeight: FontWeight.bold,
            ),
        ),
        ),
        Divider(
```

```
44          thickness: 2.0,
45          ),
46          Consumer<MobileModel>(
47          builder: (context, value, _) => Text(value.mobileColorOfFirst,
48              style: TextStyle(
49                  fontFamily: 'Trajan Pro',
50                  fontSize: 20.0,
51                  fontWeight: FontWeight.bold,
52              )),
53          ),
54      ],
55      );
56
57  Widget changeColorButtonToRed() => Column(
58      children: [
59          Container(
60          padding: const EdgeInsets.all(10.0),
61          child: Consumer<MobileModel>(
62              builder: (context, value, child) => Container(
63              padding: const EdgeInsets.all(15.0),
64              child: FloatingActionButton(
65                  backgroundColor: value.selection[1],
66                  onPressed: () {
67                  value.changeColorToRed();
68                  },
69                  child: Icon(
70                  Icons.mobile_screen_share,
71                  color: value.selection[5],
72                  ),
73              ),
74              ),
75          ),
76          ),
77          Divider(
78          thickness: 2.0,
79          ),
80          Consumer<MobileModel>(
81          builder: (context, value, _) => Text(
82              value.backgroundColorOfSecond,
83              style: TextStyle(
84              fontFamily: 'Trajan Pro',
85              fontSize: 20.0,
86              fontWeight: FontWeight.bold,
```

```
            ),
        ),
        ),
        Divider(
        thickness: 2.0,
        ),
        Consumer<MobileModel>(
        builder: (context, value, _) => Text(value.mobileColorOfSecond,
            style: TextStyle(
                fontFamily: 'Trajan Pro',
                fontSize: 20.0,
                fontWeight: FontWeight.bold,
            )),
        ),
    ],
    );

Widget restoreOldColorOfFirstMobile() => Container(
    padding: const EdgeInsets.all(10.0),
    child: Consumer<MobileModel>(
        builder: (context, value, child) => Container(
        padding: const EdgeInsets.all(10.0),
        child: RaisedButton(
            onPressed: () => value.restoreOldColorOfFirstMobile(),
            child: Text(
            'Restore',
            style: TextStyle(
                fontFamily: 'Sacramento',
                fontSize: 25.0,
                fontWeight: FontWeight.bold,
            ),
            ),
        ),
        ),
    ),
    );

Widget restoreOldColorOfSecondMobile() => Container(
    padding: const EdgeInsets.all(10.0),
    child: Consumer<MobileModel>(
        builder: (context, value, child) => Container(
        padding: const EdgeInsets.all(10.0),
        child: RaisedButton(
```

```
            onPressed: () => value.restoreOldColorOfSecondMobile(),
            child: Text(
            'Restore',
            style: TextStyle(
                fontFamily: 'Sacramento',
                fontSize: 25.0,
                fontWeight: FontWeight.bold,
            ),
            ),
        ),
        ),
    ),
    );
```

If we go through the above code, we will see several Consumers that have subscribed to those model classes. The role of these controllers are simple. They will pass those data to the screen-view pages, which we will see in a minute.

Next goes the page-specific controller file:

```
// code 8.10

// controller/second_home_page_controller.dart

import 'package:first_flutter_app/model/first_model.dart';
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';

Widget textStyleTrajanPro(String trajan) => Text(
    trajan,
    style: TextStyle(
        fontFamily: 'Trajan Pro',
        fontSize: 35.0,
        fontWeight: FontWeight.bold,
    ),
    textAlign: TextAlign.center,
    );

Widget textStyleSacramento(String sacramento) => Text(
    sacramento,
    style: TextStyle(
        fontFamily: 'Sacramento',
        fontSize: 55.0,
```

```
    ),
    textAlign: TextAlign.center,
    );

Widget changeNameButton() => Container(
    padding: const EdgeInsets.all(30.0),
    child: Consumer<FirstModel>(
        builder: (context, value, child) => Container(
        padding: const EdgeInsets.all(25.0),
        child: RaisedButton(
            child: Text(
            'Change Name',
            style: TextStyle(
                fontSize: 35.0,
                fontWeight: FontWeight.bold,
            ),
            ),
            onPressed: () {
            value.changeName();
            },
        ),
        ),
    ),
    );

Widget clearNameButton() => Container(
    padding: const EdgeInsets.all(30.0),
    child: Consumer<FirstModel>(
        builder: (context, value, child) => Container(
        padding: const EdgeInsets.all(25.0),
        child: RaisedButton(
            child: Text(
            'Clear Name',
            style: TextStyle(
                fontSize: 35.0,
                fontWeight: FontWeight.bold,
            ),
            ),
            onPressed: () {
            value.clearName();
            },
        ),
        ),
```

```
67        ),
68        );
```

In the above code, there are one or two Consumers, not as much as the mobile-specific controllers. Before going to read the screen-view code, we will take a look at how our flutter application looks like:

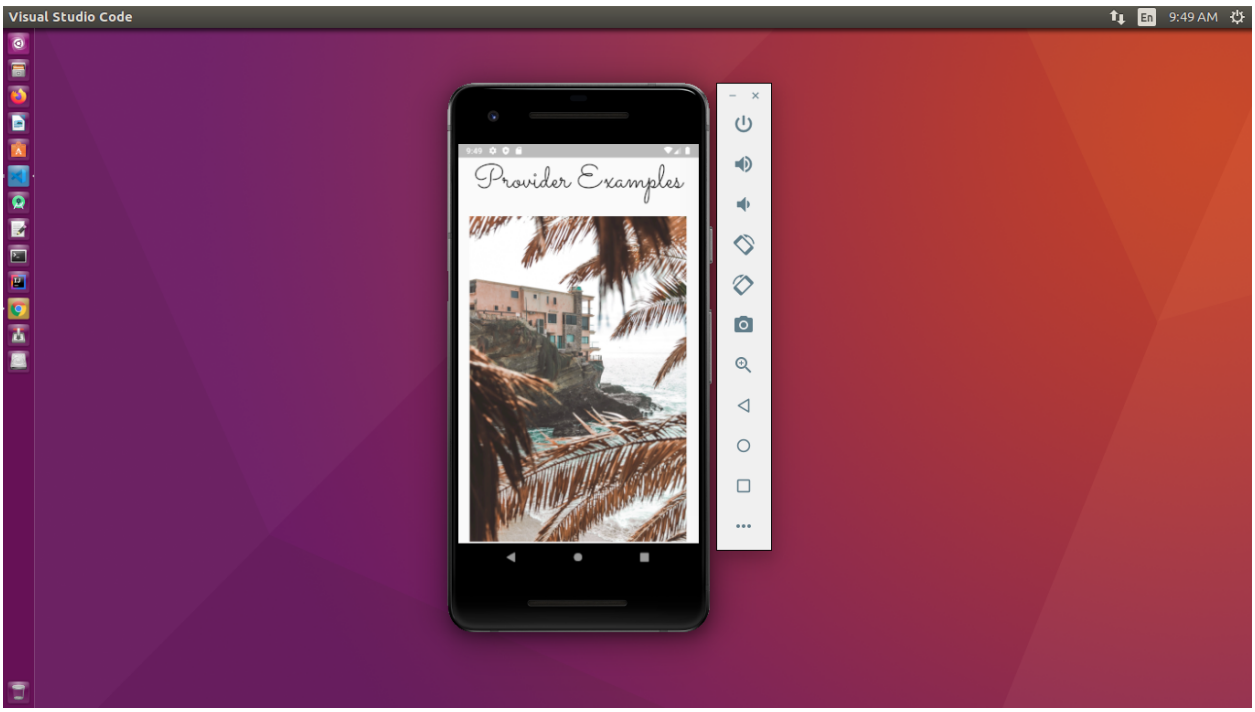


Figure 8.8 – The first look of the application that we are going to build

Now we can scroll down to the bottom and see what are waiting for us. At the bottom part, we have two buttons, and below those buttons, we have two mobile icons and respective text that tells us about the foreground and background colors.

If we click the 'Change Name' button, it will display a text 'Name Changed'. Just below that text we have the 'Clear name' button. Pressing that button will clear the text (Figure 8.9).

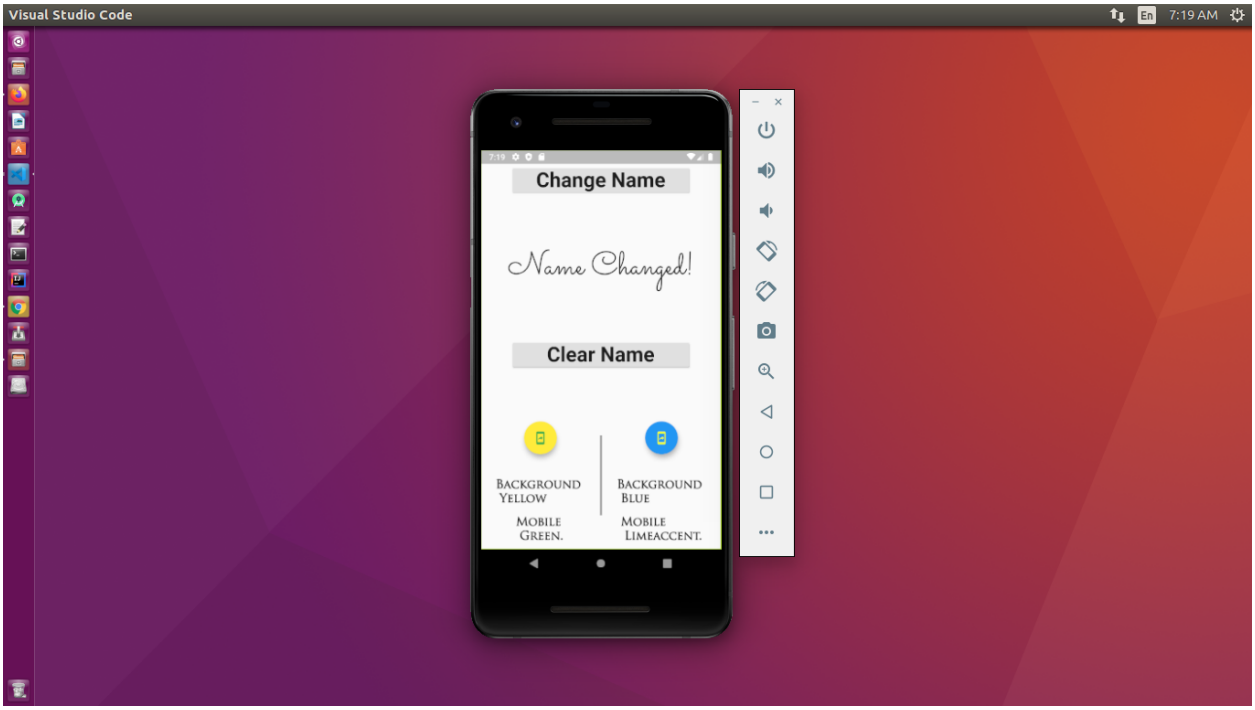


Figure 8.9 – The bottom part of our application

At the very bottom two mobile icons are visible. We can also see the name of the foreground and background color in text. Tapping any icon will change the foreground and background color, and at the same time, the description of color displayed in text will also change.

Finally, let us see the screen-view page and the main method.

```
// code 8.11

// view/second_home_app.dart


import 'package:first_flutter_app/controller/mobile_controller.dart';
import 'package:first_flutter_app/controller/second_home_page_controller.dart';
import 'package:first_flutter_app/model/first_model.dart';
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';

class SecondHomeAppPage extends StatelessWidget {
@override
Widget build(BuildContext context) {
    return MaterialApp(
    debugShowCheckedModeBanner: false,
    title: 'Second Provider Example',
```

```
18      home: Scaffold(
19          body: SafeArea(
20          child: ListView(
21              children: [
22              textStyleSacramento('Provider Examples'),
23              Container(
24                  padding: const EdgeInsets.all(20.0),
25                  child: Image.asset(
26                  'images/sea1.jpg',
27                  width: 300,
28                  ),
29              ),
30              textStyleTrajanPro('We can add humongous widget tree below...'),
31              changeNameButton(),
32              Container(
33                  padding: const EdgeInsets.all(30.0),
34                  child: textStyleSacramento(
35                      Provider.of<FirstModel>(context, listen: true).name),
36              ),
37              clearNameButton(),
38              SizedBox(
39                  height: 10.0,
40              ),
41              Row(
42                  mainAxisAlignment: MainAxisAlignment.spaceEvenly,
43                  children: [
44                  changeColorButtonToPurple(),
45                  VerticalLine(),
46                  changeColorButtonToRed(),
47                  ],
48              ),
49              SizedBox(
50                  height: 10.0,
51              ),
52              Row(
53                  mainAxisAlignment: MainAxisAlignment.spaceEvenly,
54                  children: [
55                  restoreOldColorOfFirstMobile(),
56                  VerticalLine(),
57                  restoreOldColorOfSecondMobile(),
58                  ],
59              ),
60              ],
```

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

class VerticalLine extends StatelessWidget {
const VerticalLine({
    Key key,
}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Center(
    child: Container(
        height: MediaQuery.of(context).size.height * 0.2,
        width: 3,
        color: Colors.black45,
    ),
    );
}
}

class HorizontalLine extends StatelessWidget {
const HorizontalLine({
    Key key,
}) : super(key: key);

@override
Widget build(BuildContext context) {
    return Center(
    child: Container(
        width: MediaQuery.of(context).size.width * 0.2,
        height: 3,
        color: Colors.black45,
    ),
    );
}
}
```

And the main method is as the following where we have used multi Provider :

```
// code 8.12

// main.dart

import 'package:first_flutter_app/model/first_model.dart';
import 'package:first_flutter_app/view/second_home_app.dart';
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';

import 'model/mobile_model.dart';

void main() {
runApp(
    MultiProvider(
    providers: [
        ChangeNotifierProvider(create: (context) => FirstModel()),
        ChangeNotifierProvider(create: (context) => MobileModel()),
    ],
    child: SecondHomeAppPage(),
    ),
);
}
```

Now, we can press the ‘Change Name’ button, and get the text. Let us do that, and take a look at the lower bottom part.

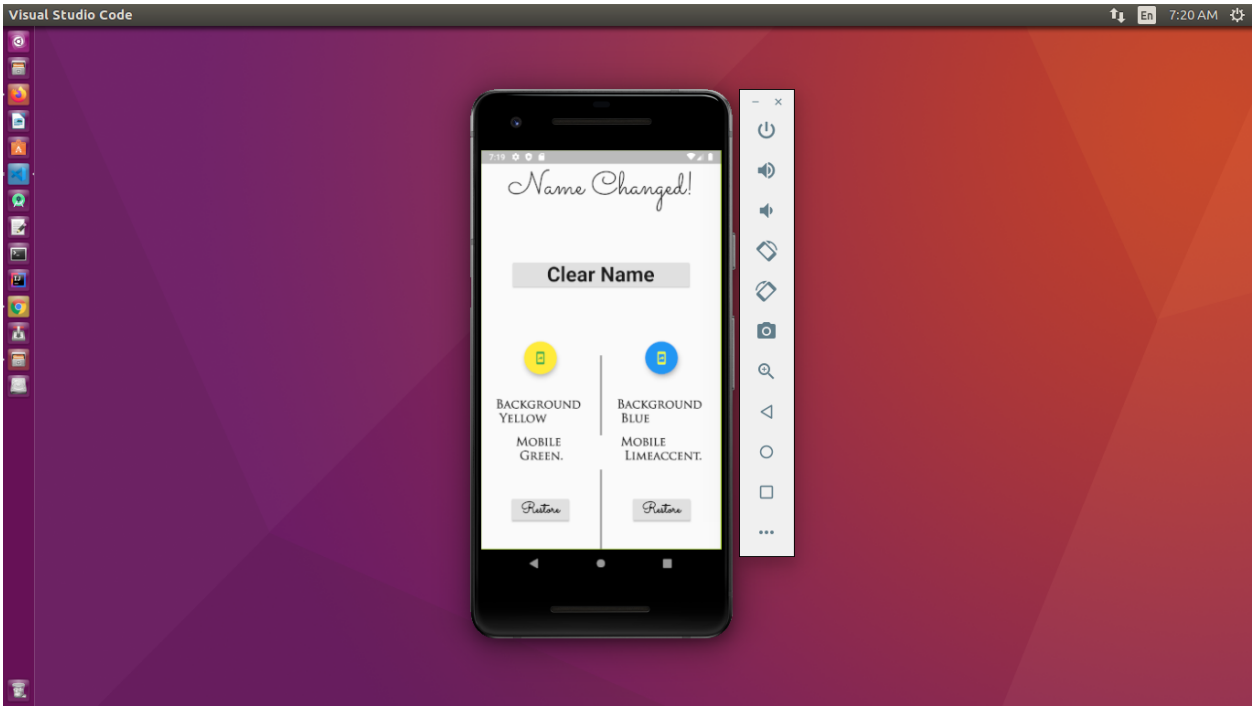


Figure 8.10 – The lower bottom part of our application

Next, we will start operating at the lower bottom part. Remember, we have already pressed the 'Change Name' button,and got the text displayed on the top of the screen-view. Now we are going to change the first mobile icon color, foreground and background, both.

Let us see the image first, after that, we will discuss the code.

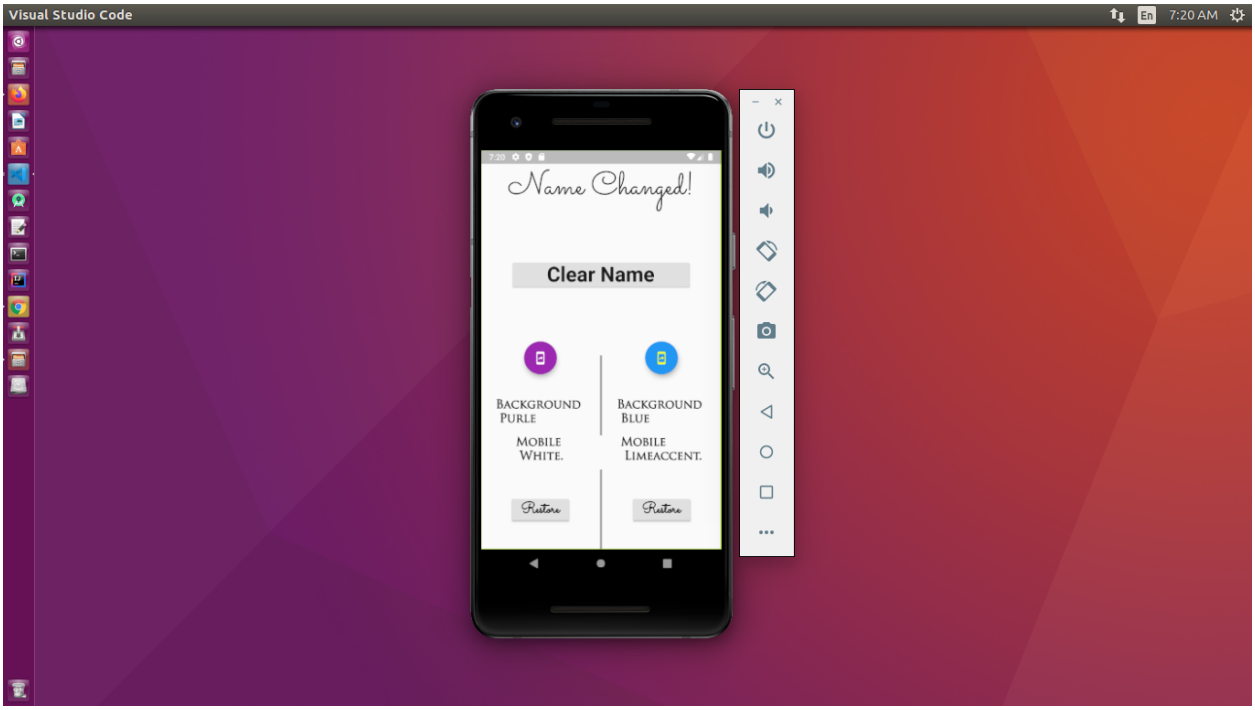


Figure 8.11 – The first mobile icon's foreground and background color have been changed and it has been reflected on the below text

We can clearly watch that the first mobile icon's foreground has been changed to white from green; at the same time the background color has been changed from yellow to purple.

Let us see this coding part in the mobile model class.

```
void changeColorToPurple() {
    backgroundColorOfFirst = 'Background \n Purle';
    mobileColorOfFirst = 'Mobile \n White.';
    selection[0] = Colors.purple;
    selection[4] = Colors.white;
    notifyListeners();
}
```

After that, we will take a look at the related mobile controller's coding part.

```
Widget changeColorButtonToPurple() => Column(
    children: [
        Container(
        padding: const EdgeInsets.all(10.0),
        child: Consumer<MobileModel>(
            builder: (context, value, child) => Container(
            padding: const EdgeInsets.all(15.0),
            child: FloatingActionButton(
                backgroundColor: value.selection[0],
                onPressed: () {
                value.changeColorToPurple();
                },
                child: Icon(
                Icons.mobile_screen_share,
                color: value.selection[4],
                ),
            ),
            ),
        ),
        ),
        Divider(
        thickness: 2.0,
        ),
        Consumer<MobileModel>(
        builder: (context, value, _) => Text(
            value.backgroundColorOfFirst,
            style: TextStyle(
            fontFamily: 'Trajan Pro',
            fontSize: 20.0,
            fontWeight: FontWeight.bold,
            ),
        ),
        ),
        Divider(
        thickness: 2.0,
        ),
        Consumer<MobileModel>(
        builder: (context, value, _) => Text(value.mobileColorOfFirst,
            style: TextStyle(
                fontFamily: 'Trajan Pro',
                fontSize: 20.0,
                fontWeight: FontWeight.bold,
            )),
```

```
44          ),
45      ],
46      );
```

And, finally we can watch the screen-view page part, where we have called this controller.

```
1  Row(
2                  mainAxisAlignment: MainAxisAlignment.spaceEvenly,
3                  children: [
4                  changeColorButtonToPurple(),
5                  VerticalLine(),
6                  changeColorButtonToRed(),
7                  ],
8              ),
```

In the same row, we can call both controllers that will change the foreground and background color.

Therefore, in the next image, we will see that the second mobile icon's foreground and background color have also been changed, because we have tapped the second icon.

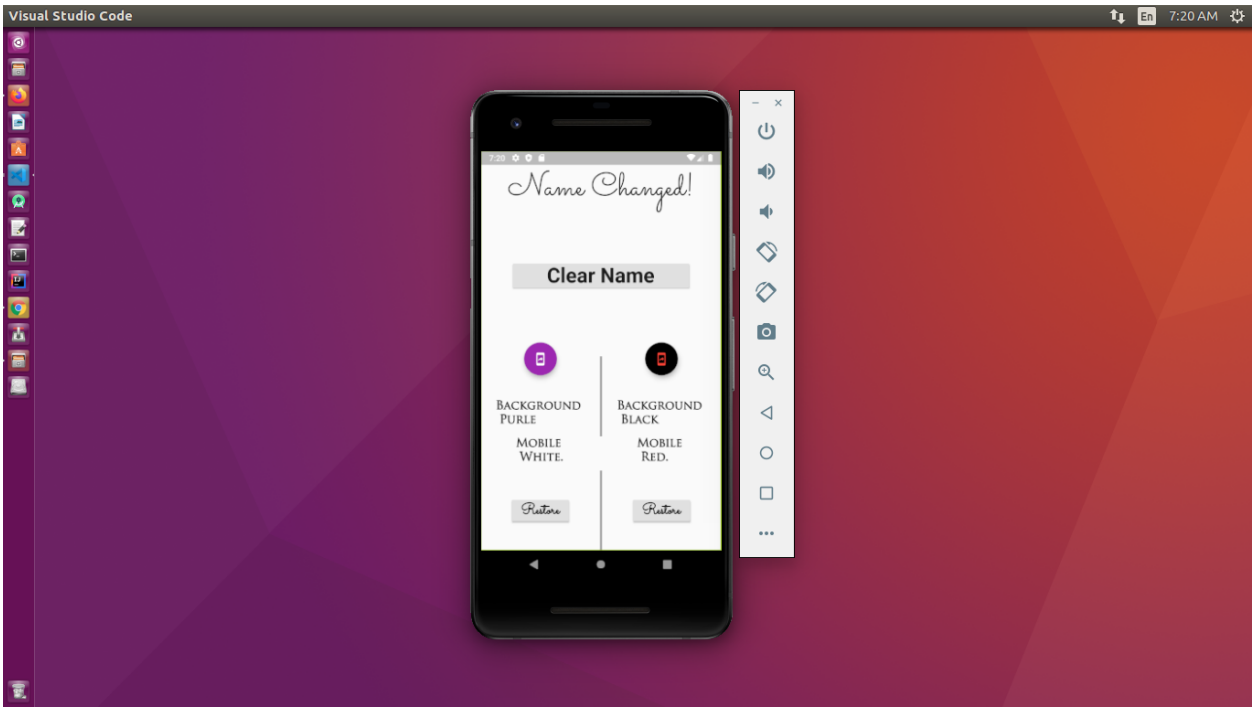


Figure 8.12 – The second mobile icon's foreground and background color have been changed

We can clearly see that the second mobile icon's foreground changed to red, and the background changed to black. The below text has also displayed the name of the color respectively.

One thing is also evident, although two controllers belong to the same Row widget, one change does not affect the other.

Our next step will be to restore the old data. First, we will click the restore button below the first mobile icon. Secondly, we will click the second restore button below the second mobile icon.

And finally, we will click the 'Clear name' button on the upper half of the screen. It will first restore the old color of the first mobile icon, next, it will change the second mobile icon; and finally it will clear the 'name' that was stuck on the upper half of the screen.

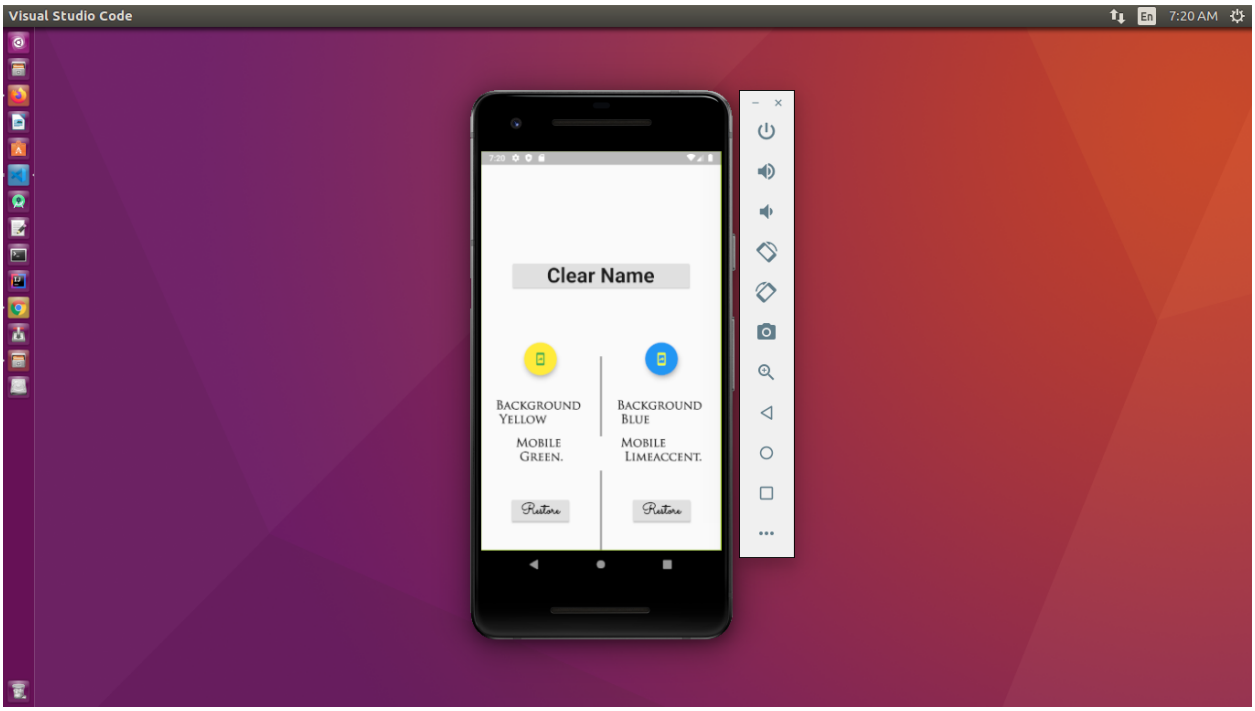


Figure 8.13 – The final screen shot of our application

We have learned how we can use Provider package, and Consumer widget to manage state efficiently. We have also learned how without rebuilding the whole widget tree, we can change and persist state of our application.

In the next chapter, we will learn how to navigate from one screen to others and come back.

Want to read more Flutter related Articles and resources?

For more Flutter related Articles and Resources[9]

[9] https://zerodotone.net

9. What Next...

Although this book is not complete, I hope it gives you an idea about Flutter and Dart work together.

There are lot of things to cover.

We will definitely meet in the next book where we will learn Mobile app building using Flutter again

Till then, for more Flutter tutorials, For more Flutter tutorials please visit[10]

The second code repository for this book[11]

[10] https://zerodotone.net
[11] https://github.com/sanjibsinha/