Android Studio Jellyfish Essentials

Kotlin Edition

Payload
publishing

Android Studio Jellyfish Essentials

Kotlin Edition

Android Studio Jellyfish Essentials – Kotlin Edition

ISBN: 978-1-951442-92-7

Rev: 1.0

https://www.payloadbooks.com

Contents

Table of Contents

Chapter 1

1. Introduction

This book, fully updated for Android Studio Jellyfish (2023.3.1) and the new UI, teaches you how to develop Android-based applications using the Kotlin programming language.

This book begins with the basics and outlines how to set up an Android development and testing environment, followed by an introduction to programming in Kotlin, including data types, control flow, functions, lambdas, and object-oriented programming. Asynchronous programming using Kotlin coroutines and flow is also covered in detail.

Chapters also cover the Android Architecture Components, including view models, lifecycle management, Room database access, content providers, the Database Inspector, app navigation, live data, and data binding.

More advanced topics such as intents are also covered, as are touch screen handling, gesture recognition, and the recording and playback of audio. This book edition also covers printing, transitions, and foldable device support.

The concepts of material design are also covered in detail, including the use of floating action buttons, Snackbars, tabbed interfaces, card views, navigation drawers, and collapsing toolbars.

Other key features of Android Studio and Android are also covered in detail, including the Layout Editor, the ConstraintLayout and ConstraintSet classes, MotionLayout Editor, view binding, constraint chains, barriers, and direct reply notifications.

Chapters also cover advanced features of Android Studio, such as App Links, Gradle build configuration, in-app billing, and submitting apps to the Google Play Developer Console.

Assuming you already have some programming experience, are ready to download Android Studio and the Android SDK, have access to a Windows, Mac, or Linux system, and have ideas for some apps to develop, you are ready to get started.

1.1 Downloading the Code Samples

The source code and Android Studio project files for the examples contained in this book are available for download at:

https://www.payloadbooks.com/product/jellyfishkotlin/

The steps to load a project from the code samples into Android Studio are as follows:

1. From the Welcome to Android Studio dialog, click on the Open button option.
2. In the project selection dialog, navigate to and select the folder containing the project to be imported and click on OK.

1.2 Feedback

We want you to be satisfied with your purchase of this book. If you find any errors in the book, or have any comments, questions or concerns please contact us at *info@payloadbooks.com*.

1.3 Errata

While we make every effort to ensure the accuracy of the content of this book, it is inevitable that a book covering a subject area of this size and complexity may include some errors and oversights. Any known issues with the book will be outlined, together with solutions, at the following URL:

https://www.payloadbooks.com/jellyfishkotlin

If you find an error not listed in the errata, please let us know by emailing our technical support team at *info@payloadbooks.com*. They are there to help you and will work to resolve any problems you may encounter.

Chapter 3

3. Creating an Example Android App in Android Studio

The preceding chapters of this book have explained how to configure an environment suitable for developing Android applications using the Android Studio IDE. Before moving on to slightly more advanced topics, now is a good time to validate that all required development packages are installed and functioning correctly. The best way to achieve this goal is to create an Android application and compile and run it. This chapter will cover creating an Android application project using Android Studio. Once the project has been created, a later chapter will explore using the Android emulator environment to perform a test run of the application.

3.1 About the Project

The project created in this chapter takes the form of a rudimentary currency conversion calculator (so simple, in fact, that it only converts from dollars to euros and does so using an estimated conversion rate). The project will also use one of the most basic Android Studio project templates. This simplicity allows us to introduce some key aspects of Android app development without overwhelming the beginner by introducing too many concepts, such as the recommended app architecture and Android architecture components, at once. When following the tutorial in this chapter, rest assured that the techniques and code used in this initial example project will be covered in much greater detail later.

3.2 Creating a New Android Project

The first step in the application development process is to create a new project within the Android Studio environment. Begin, therefore, by launching Android Studio so that the "Welcome to Android Studio" screen appears as illustrated in Figure 3-1:

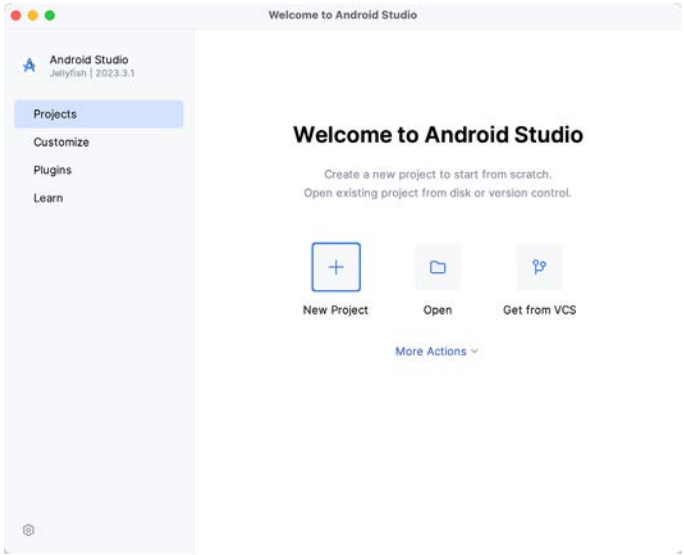

Figure 3-1

Once this window appears, Android Studio is ready for a new project to be created. To create the new project, click on the *New Project* option to display the first screen of the *New Project* wizard.

3.3 Creating an Activity

The next step is to define the type of initial activity to be created for the application. Options are available to create projects for Phone and Tablet, Wear OS, Television, or Automotive. A range of different activity types is available when developing Android applications, many of which will be covered extensively in later chapters. For this example, however, select the *Phone and Tablet* option from the Templates panel, followed by the option to create an *Empty Views Activity.* The Empty Views Activity option creates a template user interface consisting of a single TextView object.

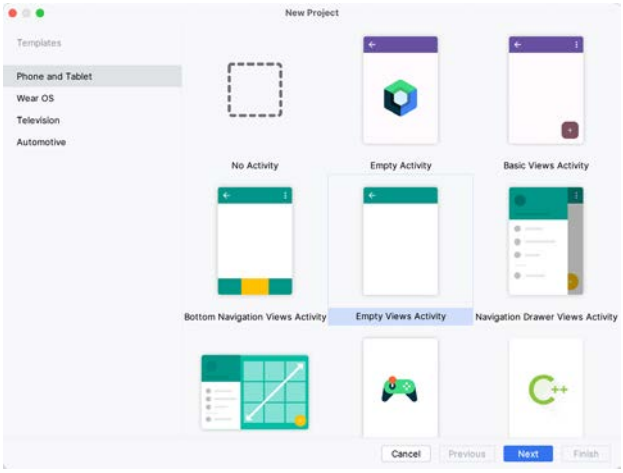

Figure 3-2

With the Empty Views Activity option selected, click *Next* to continue with the project configuration.

3.4 Defining the Project and SDK Settings

In the project configuration window (Figure 3-3), set the *Name* field to *AndroidSample*. The application name is the name by which the application will be referenced and identified within Android Studio and is also the name that would be used if the completed application were to go on sale in the Google Play store.

The *Package name* uniquely identifies the application within the Android application ecosystem. Although this can be set to any string that uniquely identifies your app, it is traditionally based on the reversed URL of your domain name followed by the application's name. For example, if your domain is *www.mycompany.com*, and the application has been named *AndroidSample*, then the package name might be specified as follows:

```
com.mycompany.androidsample
```

If you do not have a domain name, you can enter any other string into the Company Domain field, or you may use *example.com* for testing, though this will need to be changed before an application can be published:

```
com.example.androidsample
```

The *Save location* setting will default to a location in the folder named *AndroidStudioProjects* located in your home directory and may be changed by clicking on the folder icon to the right of the text field containing the current path setting.

Set the minimum SDK setting to API 26 (Oreo; Android 8.0). This minimum SDK will be used in most projects created in this book unless a necessary feature is only available in a more recent version. The objective here is to

build an app using the latest Android SDK while retaining compatibility with devices running older versions of Android (in this case, as far back as Android 8.0). The text beneath the Minimum SDK setting will outline the percentage of Android devices currently in use on which the app will run. Click on the *Help me choose* button (highlighted in Figure 3-3) to see a full breakdown of the various Android versions still in use:

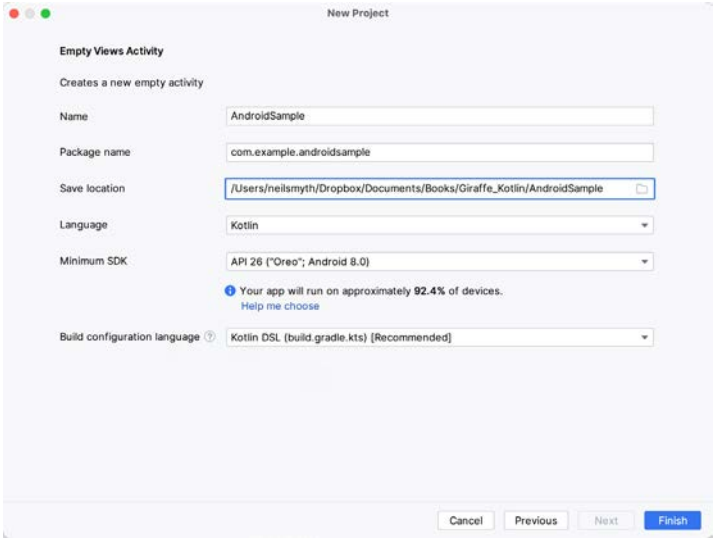

Figure 3-3

Finally, change the *Language* menu to *Kotlin* and select *Kotlin DSL (build.gradle.kts)* as the build configuration language before clicking *Finish* to create the project.

3.5 Modifying the Example Application

Once the project has been created, the main window will appear containing our AndroidSample project, as illustrated in Figure 3-4 below:

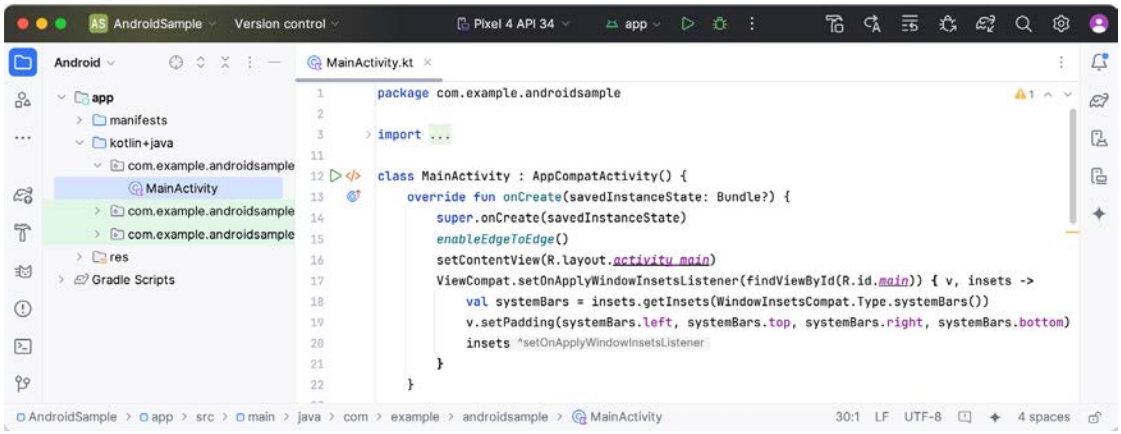

Figure 3-4

The newly created project and references to associated files are listed in the *Project* tool window on the left side of the main project window. The Project tool window has several modes in which information can be displayed. By default, this panel should be in *Android* mode. This setting is controlled by the menu at the top of the panel as highlighted in Figure 3-5. If the panel is not currently in Android mode, use the menu to switch mode:

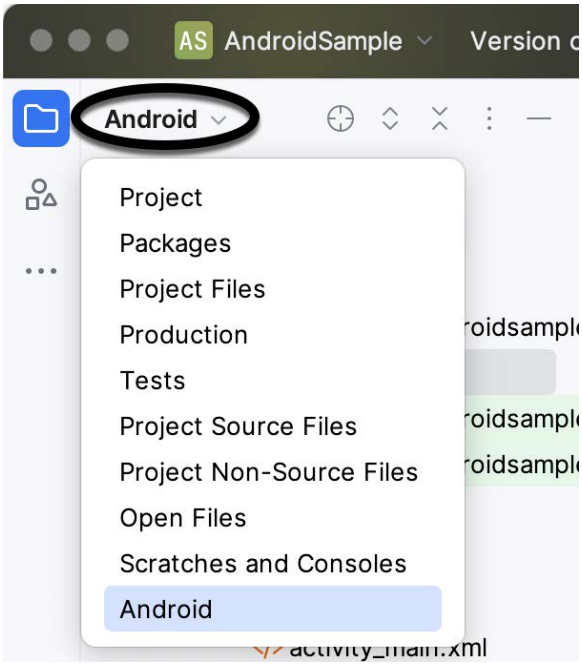

Figure 3-5

3.6 Modifying the User Interface

The user interface design for our activity is stored in a file named *activity_main.xml* which, in turn, is located under *app -> res -> layout* in the Project tool window file hierarchy. Once located in the Project tool window, double-click on the file to load it into the user interface Layout Editor tool, which will appear in the center panel of the Android Studio main window:

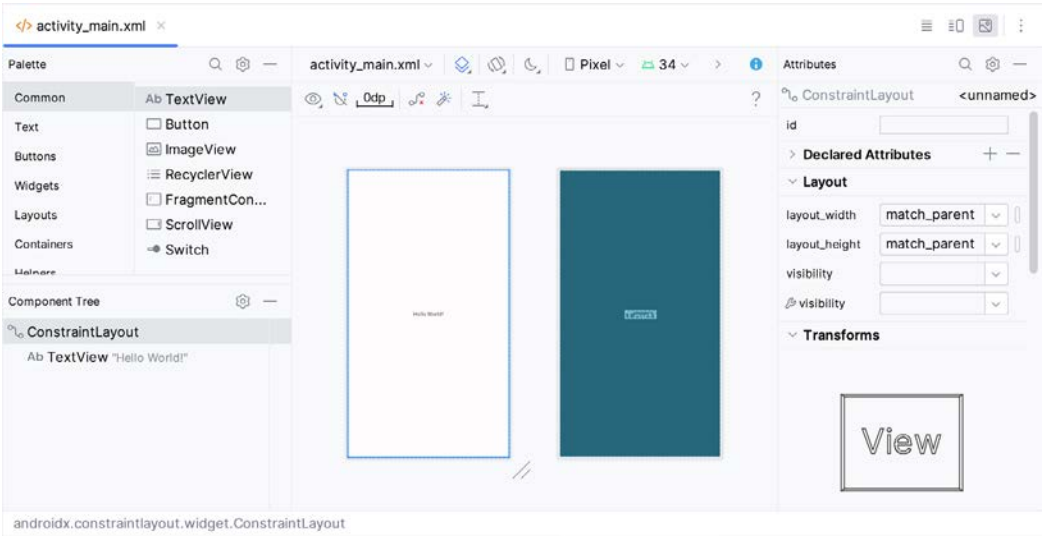

Figure 3-6

In the toolbar across the top of the Layout Editor window is a menu (currently set to *Pixel* in the above figure) which is reflected in the visual representation of the device within the Layout Editor panel. A range of other

device options are available by clicking on this menu.

Use the System UI Mode button ( ) to turn Night mode on and off for the device screen layout. To change the orientation of the device representation between landscape and portrait, use the drop-down menu showing the icon.

As we can see in the device screen, the content layout already includes a label that displays a "Hello World!" message. Running down the left-hand side of the panel is a palette containing different categories of user interface components that may be used to construct a user interface, such as buttons, labels, and text fields. However, it should be noted that not all user interface components are visible to the user. One such category consists of *layouts*. Android supports a variety of layouts that provide different levels of control over how visual user interface components are positioned and managed on the screen. Though it is difficult to tell from looking at the visual representation of the user interface, the current design has been created using a ConstraintLayout. This can be confirmed by reviewing the information in the *Component Tree* panel, which, by default, is located in the lower left-hand corner of the Layout Editor panel and is shown in Figure 3-7:

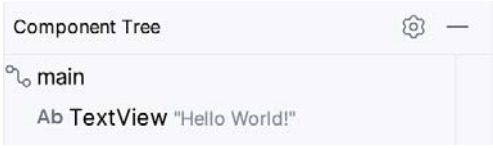

Figure 3-7

As we can see from the component tree hierarchy, the user interface layout consists of a ConstraintLayout parent called *main* and a TextView child object.

Before proceeding, check that the Layout Editor's Autoconnect mode is enabled. This means that as components are added to the layout, the Layout Editor will automatically add constraints to ensure the components are correctly positioned for different screen sizes and device orientations (a topic that will be covered in much greater detail in future chapters). The Autoconnect button appears in the Layout Editor toolbar and is represented by a U-shaped icon. When disabled, the icon appears with a diagonal line through it (Figure 3-8). If necessary, re-enable Autoconnect mode by clicking on this button.

Figure 3-8

The next step in modifying the application is to add some additional components to the layout, the first of which will be a Button for the user to press to initiate the currency conversion.

The Palette panel consists of two columns, with the left-hand column containing a list of view component categories. The right-hand column lists the components contained within the currently selected category. In Figure 3-9, for example, the Button view is currently selected within the Buttons category:

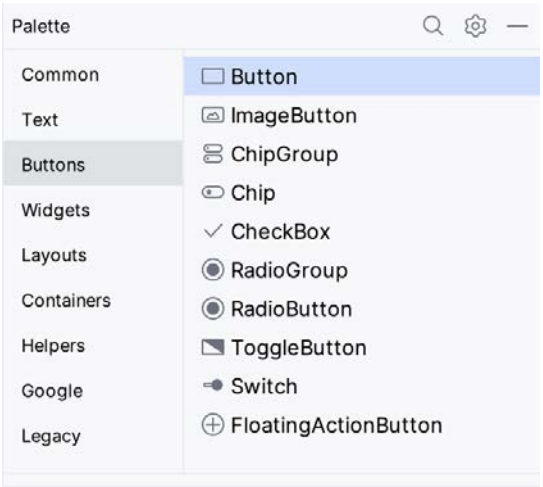

Figure 3-9

Click and drag the *Button* object from the Buttons list and drop it in the horizontal center of the user interface design so that it is positioned beneath the existing TextView widget:

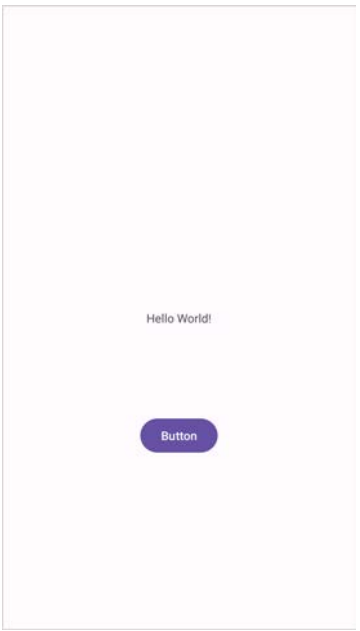

Figure 3-10

The next step is to change the text currently displayed by the Button component. The panel located to the right of the design area is the Attributes panel. This panel displays the attributes assigned to the currently selected component in the layout. Within this panel, locate the *text* property in the Common Attributes section and change the current value from "Button" to "Convert", as shown in Figure 3-11:

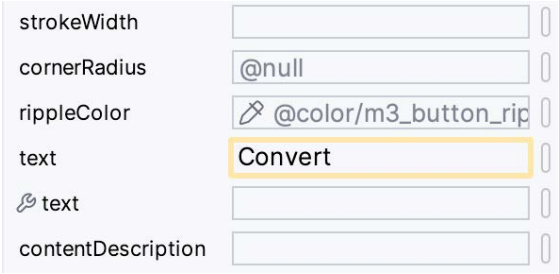

Figure 3-11

The second text property with a wrench next to it allows a text property to be set, which only appears within the Layout Editor tool but is not shown at runtime. This is useful for testing how a visual component and the layout will behave with different settings without running the app repeatedly.

Just in case the Autoconnect system failed to set all of the layout connections, click on the Infer Constraints button (Figure 3-12) to add any missing constraints to the layout:

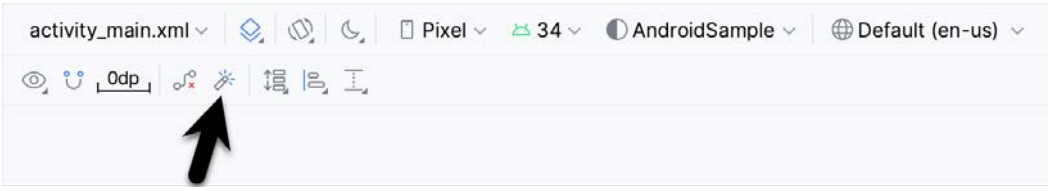

Figure 3-12

It is important to explain the warning button in the top right-hand corner of the Layout Editor tool, as indicated in Figure 3-13. This warning indicates potential problems with the layout. For details on any problems, click on the button:

Figure 3-13

When clicked, the Problems tool window (Figure 3-14) will appear, describing the nature of the problems:

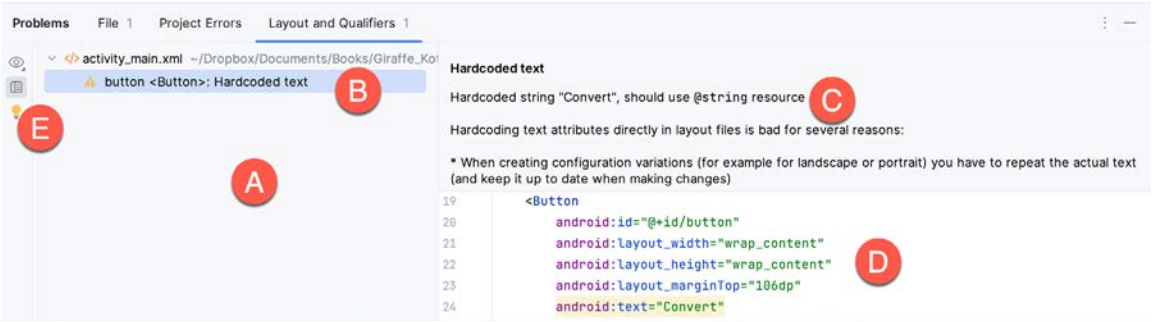

Figure 3-14

This tool window is divided into two panels. The left panel (marked A in the above figure) lists issues detected

within the layout file. In our example, only the following problem is listed:

```
button <Button>: Hardcoded text
```

When an item is selected from the list (B), the right-hand panel will update to provide additional detail on the problem (C). In this case, the explanation reads as follows:

```
Hardcoded string "Convert", should use @string resource
```

The tool window also includes a preview editor (D), allowing manual corrections to be made to the layout file.

This I18N message informs us that a potential issue exists concerning the future internationalization of the project ("I18N" comes from the fact that the word "internationalization" begins with an "I", ends with an "N" and has 18 letters in between). The warning reminds us that attributes and values such as text strings should be stored as *resources* wherever possible when developing Android applications. Doing so enables changes to the appearance of the application to be made by modifying resource files instead of changing the application source code. This can be especially valuable when translating a user interface to a different spoken language. If all of the text in a user interface is contained in a single resource file, for example, that file can be given to a translator, who will then perform the translation work and return the translated file for inclusion in the application. This enables multiple languages to be targeted without the necessity for any source code changes to be made. In this instance, we are going to create a new resource named *convert_string* and assign to it the string "Convert".

Begin by clicking on the Show Quick Fixes button (E) and selecting the *Extract string resource* option from the menu, as shown in Figure 3-15:

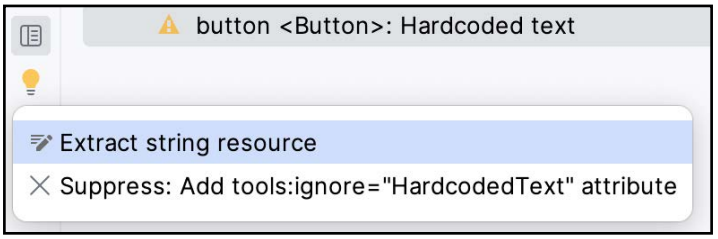

Figure 3-15

After selecting this option, the *Extract Resource* panel (Figure 3-16) will appear. Within this panel, change the resource name field to *convert_string* and leave the resource value set to *Convert* before clicking on the OK button:

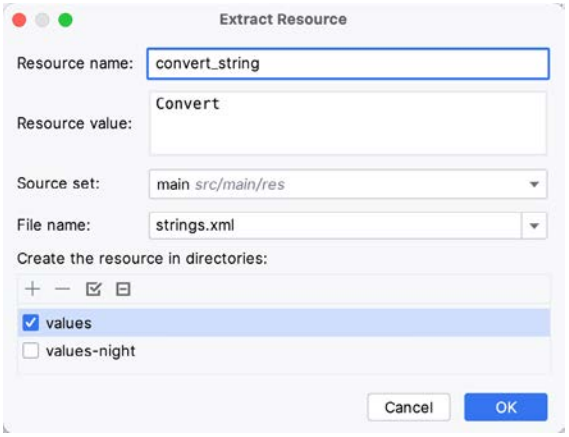

Figure 3-16

The next widget to be added is an EditText widget, into which the user will enter the dollar amount to be converted. From the Palette panel, select the Text category and click and drag a Number (Decimal) component onto the layout so that it is centered horizontally and positioned above the existing TextView widget. With the widget selected, use the Attributes tools window to set the *hint* property to "dollars". Click on the warning icon and extract the string to a resource named *dollars_hint*.

The code written later in this chapter will need to access the dollar value entered by the user into the EditText field. It will do this by referencing the id assigned to the widget in the user interface layout. The default id assigned to the widget by Android Studio can be viewed and changed from within the Attributes tool window when the widget is selected in the layout, as shown in Figure 3-17:

Figure 3-17

Change the id to *dollarText* and, in the Rename dialog, click on the *Refactor* button. This ensures that any references elsewhere within the project to the old id are automatically updated to use the new id:

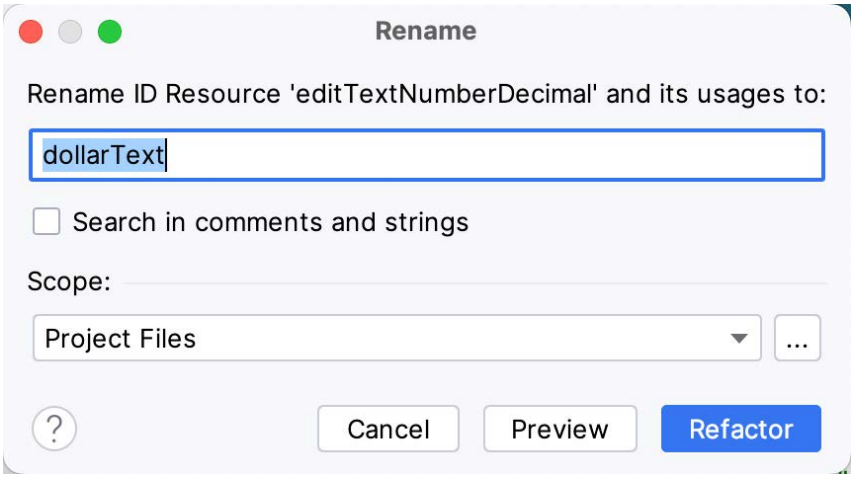

Figure 3-18

Repeat the steps to set the id of the TextView widget to *textView*, if necessary.

Add any missing layout constraints by clicking on the *Infer Constraints* button. At this point, the layout should resemble that shown in Figure 3-19:

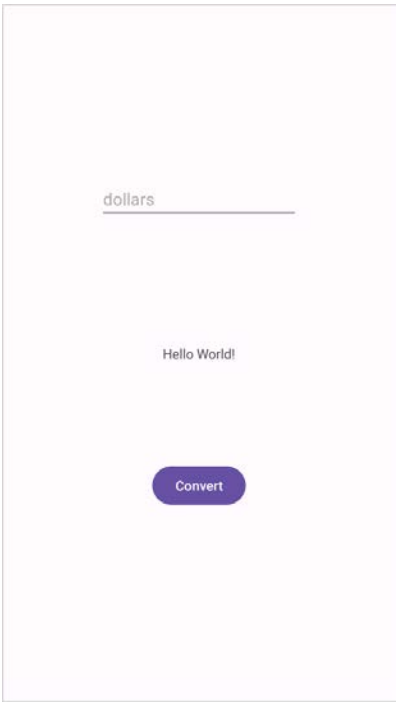

Figure 3-19

3.7 Reviewing the Layout and Resource Files

Before moving on to the next step, we will look at some internal aspects of user interface design and resource handling. In the previous section, we changed the user interface by modifying the *activity_main.xml* file using the Layout Editor tool. In fact, all that the Layout Editor was doing was providing a user-friendly way to edit the underlying XML content of the file. In practice, there is no reason why you cannot modify the XML directly to make user interface changes, and, in some instances, this may actually be quicker than using the Layout Editor tool. In the top right-hand corner of the Layout Editor panel are the View Modes buttons marked A through C in Figure 3-20 below:

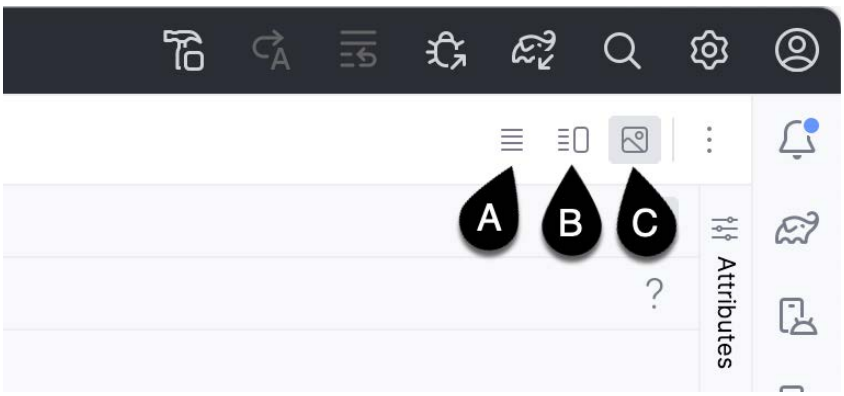

Figure 3-20

By default, the editor will be in *Design* mode (button C), whereby only the visual representation of the layout is displayed. In *Code* mode (A), the editor will display the XML for the layout, while in *Split* mode (B), both the layout and XML are displayed, as shown in Figure 3-21:

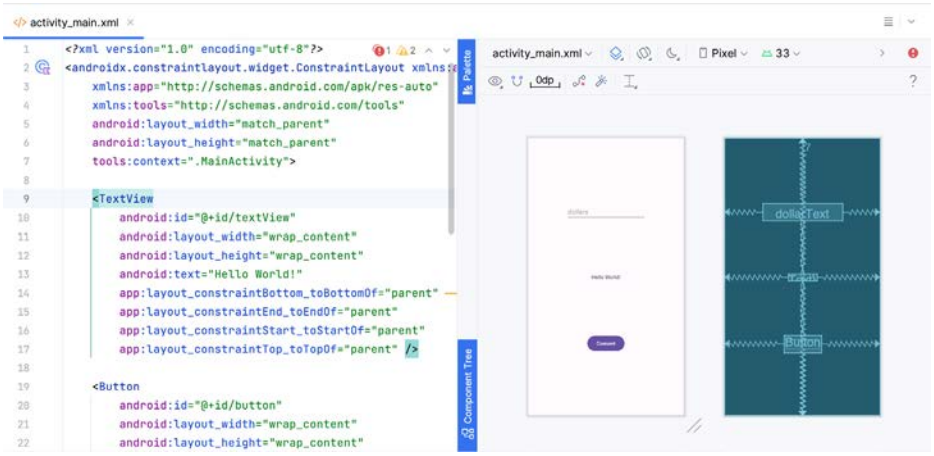

Figure 3-21

The button to the left of the View Modes button (marked B in Figure 3-20 above) is used to toggle between Code and Split modes quickly.

As can be seen from the structure of the XML file, the user interface consists of the ConstraintLayout component, which in turn, is the parent of the TextView, Button, and EditText objects. We can also see, for example, that the *text* property of the Button is set to our *convert_string* resource. Although complexity and content vary, all user interface layouts are structured in this hierarchical, XML-based way.

As changes are made to the XML layout, these will be reflected in the layout canvas. The layout may also be modified visually from within the layout canvas panel, with the changes appearing in the XML listing. To see this in action, switch to Split mode and modify the XML layout to change the background color of the ConstraintLayout to a shade of red as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.
android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity"
    android:background="#ff2438" >
.
.
</androidx.constraintlayout.widget.ConstraintLayout>
```

Note that the layout color changes in real-time to match the new setting in the XML file. Note also that a small red square appears in the XML editor's left margin (also called the *gutter*) next to the line containing the color setting. This is a visual cue to the fact that the color red has been set on a property. Clicking on the red square will display a color chooser allowing a different color to be selected:

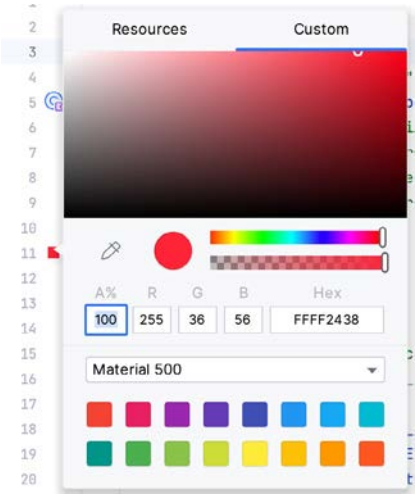

Figure 3-22

Before proceeding, delete the background property from the layout file so that the background returns to the default setting.

Finally, use the Project panel to locate the *app* -> *res* -> *values* -> *strings.xml* file and double-click on it to load it into the editor. Currently, the XML should read as follows:

```
<resources>
    <string name="app_name">AndroidSample</string>
    <string name="convert_string">Convert</string>
    <string name="dollars_hint">dollars</string>
</resources>
```

To demonstrate resources in action, change the string value currently assigned to the *convert_string* resource to "Convert to Euros" and then return to the Layout Editor tool by selecting the tab for the layout file in the editor panel. Note that the layout has picked up the new resource value for the string.

There is also a quick way to access the value of a resource referenced in an XML file. With the Layout Editor tool in Split or Code mode, click on the "@string/convert_string" property setting so that it highlights, and then press Ctrl-B on the keyboard (Cmd-B on macOS). Android Studio will subsequently open the *strings.xml* file and take you to the line in that file where this resource is declared. Use this opportunity to revert the string resource to the original "Convert" text and to add the following additional entry for a string resource that will be referenced later in the app code:

```
<resources>
    <string name="app_name">AndroidSample</string>
    <string name="convert_string">Convert</string>
    <string name="dollars_hint">dollars</string>
    <string name="no_value_string">No Value</string>
</resources>
```

Resource strings may also be edited using the Android Studio Translations Editor by clicking on the *Open editor* link in the top right-hand corner of the editor window. This will display the Translation Editor in the main panel of the Android Studio window:

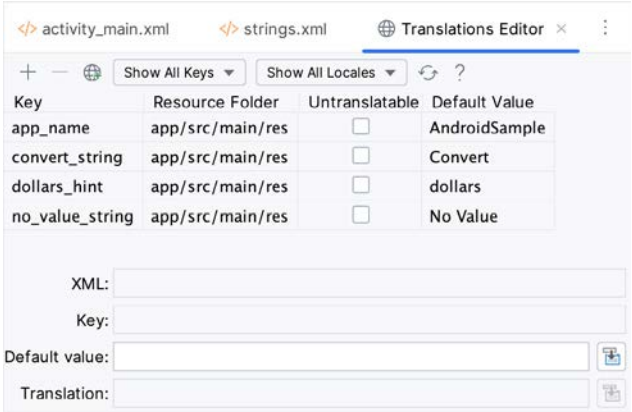

Figure 3-23

This editor allows the strings assigned to resource keys to be edited and for translations for multiple languages to be managed.

3.8 Adding Interaction

The final step in this example project is to make the app interactive so that when the user enters a dollar value into the EditText field and clicks the convert button, the converted euro value appears on the TextView. This involves the implementation of some event handling on the Button widget. Specifically, the Button needs to be configured so that a method in the app code is called when an *onClick* event is triggered. Event handling can be implemented in several ways and is covered in a later chapter entitled *"An Overview and Example of Android Event Handling"*. Return the layout editor to Design mode, select the Button widget in the layout editor, refer to the Attributes tool window, and specify a method named *convertCurrency* as shown below:

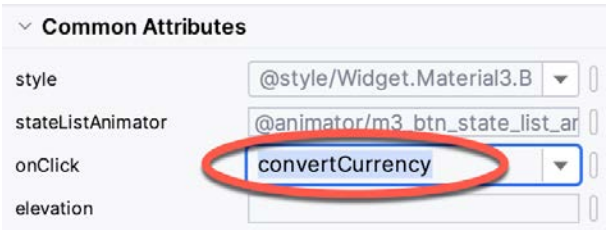

Figure 3-24

Next, double-click on the *MainActivity.kt* file in the Project tool window (*app -> kotlin+java -> <package name> -> MainActivity*) to load it into the code editor and add the code for the *convertCurrency* method to the class file so that it reads as follows, noting that it is also necessary to import some additional Android packages:

```
package com.example.androidsample

import android.os.Bundle
import androidx.activity.enableEdgeToEdge
import androidx.appcompat.app.AppCompatActivity
import androidx.core.view.ViewCompat
import androidx.core.view.WindowInsetsCompat
import android.view.View
import android.widget.EditText
import android.widget.TextView
```

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
.
.
        }
    }

    fun convertCurrency(view: View) {
        val dollarText: EditText = findViewById(R.id.dollarText)
        val textView: TextView = findViewById(R.id.textView)

        if (dollarText.text.isNotEmpty()) {
            val dollarValue = dollarText.text.toString().toFloat()
            val euroValue = dollarValue * 0.85f
            textView.text = euroValue.toString()
        } else {
            textView.text = getString(R.string.no_value_string)
        }
    }
}
```

The method begins by obtaining references to the EditText and TextView objects by making a call to a method named findViewById, passing through the id assigned within the layout file. A check is then made to ensure that the user has entered a dollar value, and if so, that value is extracted, converted from a String to a floating point value, and converted to euros. Finally, the result is displayed on the TextView widget.

If any of this is unclear, rest assured that these concepts will be covered in greater detail in later chapters. In particular, the topic of accessing widgets from within code using findByViewId and an introduction to an alternative technique referred to as *view binding* will be covered in the chapter entitled *"An Overview of Android View Binding"*.

3.9 Summary

While not excessively complex, several steps are involved in setting up an Android development environment. Having performed those steps, it is worth working through an example to ensure the environment is correctly installed and configured. In this chapter, we have created an example application and then used the Android Studio Layout Editor tool to modify the user interface layout. In doing so, we explored the importance of using resources wherever possible, particularly string values, and briefly touched on layouts. Next, we looked at the underlying XML used to store Android application user interface designs.

Finally, an onClick event was added to a Button connected to a method implemented to extract the user input from the EditText component, convert it from dollars to euros and then display the result on the TextView.

With the app ready for testing, the steps necessary to set up an emulator for testing purposes will be covered in detail in the next chapter.

Chapter 14

14. Kotlin Control Flow

Regardless of the programming language used, application development is largely an exercise in applying logic, and much of the art of programming involves writing code that makes decisions based on one or more criteria. Such decisions define which code gets executed, how many times it is executed and, conversely, which code gets by-passed when the program is executing. This is often referred to as *control flow* since it controls the *flow* of program execution. Control flow typically falls into the categories of *looping control* (how often code is executed) and *conditional control flow* (whether or not code is executed). This chapter is intended to provide an introductory overview of both types of control flow in Kotlin.

14.1 Looping Control flow

This chapter will begin by looking at control flow in the form of loops. Loops are essentially sequences of Kotlin statements which are to be executed repeatedly until a specified condition is met. The first looping statement we will explore is the *for* loop.

14.1.1 The Kotlin *for-in* Statement

The for-in loop is used to iterate over a sequence of items contained in a collection or number range.

The syntax of the for-in loop is as follows:

```
for variable name in collection or range {
         // code to be executed
}
```

In this syntax, *variable name* is the name to be used for a variable that will contain the current item from the collection or range through which the loop is iterating. The code in the body of the loop will typically use this name as a reference to the current item in the loop cycle. The *collection* or *range* references the item through which the loop is iterating. This could, for example, be an array of string values, a range operator or even a string of characters.

Consider, for example, the following for-in loop construct:

```
for (index in 1..5) {
  println("Value of index is $index")
}
```

The loop begins by stating that the current item is to be assigned to a constant named *index*. The statement then declares a closed range operator to indicate that the for loop is to iterate through a range of numbers, starting at 1 and ending at 5. The body of the loop prints out a message to the console indicating the current value assigned to the *index* constant, resulting in the following output:

```
Value of index is 1
Value of index is 2
Value of index is 3
Value of index is 4
Value of index is 5
```

The for-in loop is of particular benefit when working with collections such as arrays. In fact, the for-in loop can be used to iterate through any object that contains more than one item. The following loop, for example, outputs

each of the characters in the specified string:

```
for (index in "Hello") {
  println("Value of index is $index")
}
```

The operation of a for-in loop may be configured using the *downTo* and *until* functions. The downTo function causes the for loop to work backwards through the specified collection until the specified number is reached. The following for loop counts backwards from 100 until the number 90 is reached:

```
for (index in 100 downTo 90) {
  print("$index.. ")
}
```

When executed, the above loop will generate the following output:

```
100.. 99.. 98.. 97.. 96.. 95.. 94.. 93.. 92.. 91.. 90..
```

The until function operates in much the same way with the exception that counting starts from the bottom of the collection range and works up until (but not including) the specified end point (a concept referred to as a half closed range):

```
for (index in 1 until 10) {
  print("$index.. ")
}
```

The output from the above code will range from the start value of 1 through to 9:

```
1.. 2.. 3.. 4.. 5.. 6.. 7.. 8.. 9..
```

The increment used on each iteration through the loop may also be defined using the step function as follows:

```
for (index in 0 until 100 step 10) {
  print("$index.. ")
}
```

The above code will result in the following console output:

```
0.. 10.. 20.. 30.. 40.. 50.. 60.. 70.. 80.. 90..
```

14.1.2 The *while* Loop

The Kotlin *for* loop described previously works well when it is known in advance how many times a particular task needs to be repeated in a program. There will, however, be instances where code needs to be repeated until a certain condition is met, with no way of knowing in advance how many repetitions are going to be needed to meet that criteria. To address this need, Kotlin includes the *while* loop.

Essentially, the while loop repeats a set of tasks while a specified condition is met. The *while* loop syntax is defined as follows:

```
while condition {
      // Kotlin statements go here
}
```

In the above syntax, *condition* is an expression that will return either *true* or *false* and the *// Kotlin statements go here* comment represents the code to be executed while the condition expression is true. For example:

```
var myCount = 0

while  (myCount < 100) {
```

```
    myCount++
    println(myCount)
}
```

In the above example, the *while* expression will evaluate whether the *myCount* variable is less than 100. If it is already greater than 100, the code in the braces is skipped and the loop exits without performing any tasks.

If, on the other hand, *myCount* is not greater than 100 the code in the braces is executed and the loop returns to the while statement and repeats the evaluation of *myCount*. This process repeats until the value of *myCount* is greater than 100, at which point the loop exits.

14.1.3 The *do ... while* loop

It is often helpful to think of the *do ... while* loop as an inverted while loop. The *while* loop evaluates an expression before executing the code contained in the body of the loop. If the expression evaluates to *false* on the first check then the code is not executed. The *do ... while* loop, on the other hand, is provided for situations where you know that the code contained in the body of the loop will *always* need to be executed at least once. For example, you may want to keep stepping through the items in an array until a specific item is found. You know that you have to at least check the first item in the array to have any hope of finding the entry you need. The syntax for the *do ... while* loop is as follows:

```
do {
       // Kotlin statements here
} while conditional expression
```

In the *do ... while* example below the loop will continue until the value of a variable named i equals 0:

```
var i = 10

do {
   i--
   println(i)
} while (i > 0)
```

14.1.4 Breaking from Loops

Having created a loop, it is possible that under certain conditions you might want to break out of the loop before the completion criteria have been met (particularly if you have created an infinite loop). One such example might involve continually checking for activity on a network socket. Once activity has been detected it will most likely be necessary to break out of the monitoring loop and perform some other task.

For the purpose of breaking out of a loop, Kotlin provides the *break* statement which breaks out of the current loop and resumes execution at the code directly after the loop. For example:

```
var j = 10

for (i in 0..100)
{
    j += j

    if (j > 100) {
        break
    }
```

```
    println("j = $j")
}
```

In the above example the loop will continue to execute until the value of j exceeds 100 at which point the loop will exit and execution will continue with the next line of code after the loop.

14.1.5 The *continue* Statement

The *continue* statement causes all remaining code statements in a loop to be skipped, and execution to be returned to the top of the loop. In the following example, the *println* function is only called when the value of variable *i* is an even number:

```
var i = 1

while (i < 20)
{
        i += 1

        if (i % 2 != 0) {
            continue
        }

        println("i = $i")
}
```

The *continue* statement in the above example will cause the *println* call to be skipped unless the value of *i* can be divided by 2 with no remainder. If the *continue* statement is triggered, execution will skip to the top of the while loop and the statements in the body of the loop will be repeated (until the value of *i* exceeds 19).

14.1.6 Break and Continue Labels

Kotlin expressions may be assigned a label by preceding the expression with a label name followed by the @ sign. This label may then be referenced when using break and continue statements to designate where execution is to resume. This is particularly useful when breaking out of nested loops. The following code contains a for loop nested within another for loop. The inner loop contains a break statement which is executed when the value of j reaches 10:

```
for (i in 1..100) {

    println("Outer loop i = $i")

    for (j in 1..100) {
        println("Inner loop j = $j")
        if (j == 10) break
    }
}
```

As currently implemented, the break statement will exit the inner for loop but execution will resume at the top of the outer for loop. Suppose, however, that the break statement is required to also exit the outer loop. This can be achieved by assigning a label to the outer loop and referencing that label with the break statement as follows:

```
outerloop@ for (i in 1..100) {
```

```
    println("Outer loop i = $i")

    for (j in 1..100) {

        println("Inner loop j = $j")

        if (j == 10) break@outerloop
    }
}
```

Now when the value assigned to variable j reaches 10 the break statement will break out of both loops and resume execution at the line of code immediately following the outer loop.

14.2 Conditional Control Flow

In the previous chapter we looked at how to use logical expressions in Kotlin to determine whether something is *true* or *false*. Since programming is largely an exercise in applying logic, much of the art of programming involves writing code that makes decisions based on one or more criteria. Such decisions define which code gets executed and, conversely, which code gets by-passed when the program is executing.

14.2.1 Using the *if* Expressions

The *if* expression is perhaps the most basic of control flow options available to the Kotlin programmer. Programmers who are familiar with C, Swift, C++ or Java will immediately be comfortable using Kotlin if statements, although there are some subtle differences.

The basic syntax of the Kotlin *if* expression is as follows:

```
if (boolean expression) {
    // Kotlin code to be performed when expression evaluates to true
}
```

Unlike some other programming languages, it is important to note that the braces are optional in Kotlin if only one line of code is associated with the *if* expression. In fact, in this scenario, the statement is often placed on the same line as the if expression.

Essentially if the *Boolean expression* evaluates to *true* then the code in the body of the statement is executed. If, on the other hand, the expression evaluates to *false* the code in the body of the statement is skipped.

For example, if a decision needs to be made depending on whether one value is greater than another, we would write code similar to the following:

```
val x = 10


if (x > 9) println("x is greater than 9!")
```

Clearly, x is indeed greater than 9 causing the message to appear in the console panel.

At this point it is important to notice that we have been referring to the if expression instead of the if statement. The reason for this is that unlike the if statement in other programming languages, the Kotlin if returns a result. This allows if constructs to be used within expressions. As an example, a typical if expression to identify the largest of two numbers and assign the result to a variable might read as follows:

```
if (x > y)
    largest = x
else
```

```
    largest = y
```

The same result can be achieved using the *if* statement within an expression using the following syntax:

```
variable = if (condition) return_val_1 else return_val_2
```

The original example can, therefore be re-written as follows:

```
val largest = if (x > y) x else y
```

The technique is not limited to returning the values contained within the condition. The following example is also a valid use of if in an expression, in this case assigning a string value to the variable:

```
val largest = if (x > y) "x is greatest" else "y is greatest"
println(largest)
```

For those familiar with programming languages such as Java, this feature allows code constructs similar to ternary statements to be implemented in Kotlin.

14.2.2 Using *if ... else …* Expressions

The next variation of the *if* expression allows us to also specify some code to perform if the expression in the if expression evaluates to *false*. The syntax for this construct is as follows:

```
if (boolean expression) {
    // Code to be executed if expression is true
} else {
    // Code to be executed if expression is false
}
```

The braces are, once again, optional if only one line of code is to be executed.

Using the above syntax, we can now extend our previous example to display a different message if the comparison expression evaluates to be *false*:

```
val x = 10

if (x > 9) println("x is greater than 9!")
    else println("x is less than 9!")
```

In this case, the second println statement will execute if the value of x was less than 9.

14.2.3 Using *if ... else if ...* Expressions

So far we have looked at *if* statements which make decisions based on the result of a single logical expression. Sometimes it becomes necessary to make decisions based on a number of different criteria. For this purpose, we can use the *if ... else if ...* construct, an example of which is as follows:

```
var x = 9

if (x == 10) println("x is 10")
        else if (x == 9) println("x is 9")
            else if (x == 8) println("x is 8")
                 else println("x is less than 8")
}
```

14.2.4 Using the *when* Statement

The Kotlin *when* statement provides a cleaner alternative to the *if ... else if ...* construct and uses the following syntax:

```
when (value) {
       match1 -> // code to be executed on match
       match2 -> // code to be executed on match
       .
       .
       else -> // default code to executed if no match
}
```

Using this syntax, the previous *if ... else if ...* construct can be rewritten to use the *when* statement:

```
when (x) {
    10 -> println ("x is 10")
    9 -> println("x is 9")
    8 -> println("x is 8")
    else ->  println("x is less than 8")
}
```

The *when* statement is similar to the *switch* statement found in many other programming languages.

14.3 Summary

The term *control flow* is used to describe the logic that dictates the execution path that is taken through the source code of an application as it runs. This chapter has looked at the two types of control flow provided by Kotlin (looping and conditional) and explored the various Kotlin constructs that are available to implement both forms of control flow logic.

Chapter 62

62. An Introduction to Kotlin Coroutines

When an Android application is first started, the runtime system creates a single thread in which all components will run by default. This thread is generally referred to as the *main thread*. The primary role of the main thread is to handle the user interface in terms of event handling and interaction with views in the user interface. Any additional components started within the application will, by default, also run on the main thread.

Any code within an application that performs a time-consuming task using the main thread will cause the entire application to appear to lock up until the task is completed. This typically results in the operating system displaying an "Application is not responding" warning to the user. This is far from the desired behavior for any application. Fortunately, Kotlin provides a lightweight alternative in the form of Coroutines. This chapter will introduce Coroutines, including terminology such as dispatchers, coroutine scope, suspend functions, coroutine builders, and structured concurrency. The chapter will also explore channel-based communication between coroutines.

62.1 What are Coroutines?

Coroutines are blocks of code that execute asynchronously without blocking the thread from which they are launched. Coroutines can be implemented without worrying about building complex AsyncTask implementations or directly managing multiple threads. Because of the way they are implemented, coroutines are much more efficient and less resource intensive than using traditional multi-threading options. Coroutines also make for code that is much easier to write, understand and maintain since it allows code to be written sequentially without having to write callbacks to handle thread-related events and results.

Although a relatively recent addition to Kotlin, there is nothing new or innovative about coroutines. Coroutines, in one form or another, have existed in programming languages since the 1960s and are based on a model known as Communicating Sequential Processes (CSP). Though it does so efficiently, Kotlin still uses multi-threading behind the scenes.

62.2 Threads vs. Coroutines

A problem with threads is that they are a finite resource and expensive in terms of CPU capabilities and system overhead. In the background, much work is involved in creating, scheduling, and destroying a thread. Although modern CPUs can run large numbers of threads, the actual number of threads that can be run in parallel at any one time is limited by the number of CPU cores (though newer CPUs have 8 cores, most Android devices contain CPUs with 4 cores). When more threads are required than there are CPU cores, the system has to perform thread scheduling to decide how the execution of these threads is to be shared between the available cores.

To avoid these overheads, instead of starting a new thread for each coroutine and destroying it when the coroutine exits, Kotlin maintains a pool of active threads and manages how coroutines are assigned to those threads. When an active coroutine is suspended, the Kotlin runtime saves it, and another coroutine resumes to take its place. When the coroutine is resumed, it is restored to an existing unoccupied thread within the pool to continue executing until it either completes or is suspended. Using this approach, a limited number of threads are used efficiently to execute asynchronous tasks with the potential to perform large numbers of concurrent

tasks without the inherent performance degeneration that would occur using standard multi-threading.

62.3 Coroutine Scope

All coroutines must run within a specific scope, allowing them to be managed as groups instead of as individual ones. This is particularly important when canceling and cleaning up coroutines, for example, when a Fragment or Activity is destroyed, and ensuring that coroutines do not "leak" (in other words, continue running in the background when the app no longer needs them). By assigning coroutines to a scope, they can, for example, all be canceled in bulk when they are no longer needed.

Kotlin and Android provide built-in scopes and the option to create custom scopes using the CoroutineScope class. The built-in scopes can be summarized as follows:

- **GlobalScope** – GlobalScope is used to launch top-level coroutines tied to the entire application lifecycle. Since this has the potential for coroutines in this scope to continue running when not needed (for example, when an Activity exits), use of this scope is not recommended for Android applications. Coroutines running in GlobalScope are considered to be using *unstructured concurrency*.
- **ViewModelScope** – Provided specifically for ViewModel instances when using the Jetpack architecture ViewModel component. Coroutines launched in this scope from within a ViewModel instance are automatically canceled by the Kotlin runtime system when the corresponding ViewModel instance is destroyed.
- **LifecycleScope** - Every lifecycle owner has associated with it a LifecycleScope. This scope is canceled when the corresponding lifecycle owner is destroyed, making it particularly useful for launching coroutines from within activities and fragments.

For all other requirements, a custom scope will likely be used. The following code, for example, creates a custom scope named *myCoroutineScope*:

```
private val myCoroutineScope = CoroutineScope(Dispatchers.Main)
```

The coroutineScope declares the dispatcher that will be used to run coroutines (though this can be overridden) and must be referenced each time a coroutine is started if it is to be included within the scope. All of the running coroutines in a scope can be canceled via a call to the *cancel()* method of the scope instance:

```
myCoroutineScope.cancel()
```

62.4 Suspend Functions

A suspend function is a special type of Kotlin function that contains the code of a coroutine. It is declared using the Kotlin *suspend* keyword, which indicates to Kotlin that the function can be paused and resumed later, allowing long-running computations to execute without blocking the main thread.

The following is an example suspend function:

```
suspend fun mySlowTask() {
    // Perform long-running tasks here
}
```

62.5 Coroutine Dispatchers

Kotlin maintains threads for different types of asynchronous activity, and when launching a coroutine, it will be necessary to select the appropriate dispatcher from the following options:

- **Dispatchers.Main** – Runs the coroutine on the main thread and is suitable for coroutines that need to make changes to the UI and as a general-purpose option for performing lightweight tasks.
- **Dispatchers.IO** – Recommended for coroutines that perform network, disk, or database operations.

- **Dispatchers.Default** – Intended for CPU-intensive tasks such as sorting data or performing complex calculations.

The dispatcher is responsible for assigning coroutines to appropriate threads and suspending and resuming the coroutine during its lifecycle. In addition to the predefined dispatchers, it is also possible to create dispatchers for your own custom thread pools.

62.6 Coroutine Builders

The coroutine builders bring together all of the components covered so far and launch the coroutines so that they start executing. For this purpose, Kotlin provides the following six builders:

- **launch** – Starts a coroutine without blocking the current thread and does not return a result to the caller. Use this builder when calling a suspend function from within a traditional function and when the results of the coroutine do not need to be handled (sometimes referred to as "fire and forget" coroutines).
- **async** – Starts a coroutine and allows the caller to wait for a result using the await() function without blocking the current thread. Use async when you have multiple coroutines that need to run in parallel. The async builder can only be used from within another suspend function.
- **withContext** – Allows a coroutine to be launched in a different context from that used by the parent coroutine. Using this builder, a coroutine running using the Main context could launch a child coroutine in the Default context. The withContext builder also provides a useful alternative to async when returning results from a coroutine.
- **coroutineScope** – The coroutineScope builder is ideal for situations where a suspend function launches multiple coroutines that will run in parallel and where some action must occur only when all the coroutines reach completion. If those coroutines are launched using the coroutineScope builder, the calling function will not return until all child coroutines have completed. When using coroutineScope, a failure in any coroutine will cancel all other coroutines.
- **supervisorScope** – Similar to the coroutineScope outlined above, except that a failure in one child does not result in the cancellation of the other coroutines.
- **runBlocking** - Starts a coroutine and blocks the current thread until the coroutine reaches completion. This is typically the exact opposite of what is wanted from coroutines but is useful for testing code and when integrating legacy code and libraries. Otherwise to be avoided.

62.7 Jobs

Each call to a coroutine builder, such as launch or async, returns a Job instance which can, in turn, be used to track and manage the lifecycle of the corresponding coroutine. Subsequent builder calls from within the coroutine create new Job instances, which will become children of the immediate parent Job, forming a parent-child relationship tree where canceling a parent Job will recursively cancel all its children. Canceling a child does not, however, cancel the parent, though an uncaught exception within a child created using the launch builder may result in the cancellation of the parent (this is not the case for children created using the async builder, which encapsulates the exception in the result returned to the parent).

The status of a coroutine can be identified by accessing the isActive, isCompleted, and isCancelled properties of the associated Job object. In addition to these properties, several methods are also available on a Job instance. For example, a Job and all of its children may be canceled by calling the cancel() method of the Job object, while a call to the *cancelChildren()* method will cancel all child coroutines.

The *join()* method can be called to suspend the coroutine associated with the job until all of its child jobs have completed. To perform this task and cancel the Job once all child jobs have completed, call the *cancelAndJoin()*

method.

This hierarchical Job structure, together with coroutine scopes, form the foundation of structured concurrency, which aims to ensure that coroutines do not run longer than required without manually keeping references to each coroutine.

62.8 Coroutines – Suspending and Resuming

It helps to see some coroutine examples in action to understand coroutine suspension better. To start with, let's assume a simple Android app containing a button that, when clicked, calls a function named *startTask()*. This function calls a suspend function named *performSlowTask()* using the Main coroutine dispatcher. The code for this might read as follows:

```
private val myCoroutineScope = CoroutineScope(Dispatchers.Main)

fun startTask(view: View) {
    myCoroutineScope.launch(Dispatchers.Main) {
        performSlowTask()
    }
}
```

In the above code, a custom scope is declared and referenced in the call to the launch builder, which, in turn, calls the *performSlowTask()* suspend function. Since *startTask()* is not a suspend function, the coroutine must be started using the launch builder instead of the async builder.

Next, we can declare the *performSlowTask()* suspend function as follows:

```
suspend fun performSlowTask() {
    Log.i(TAG, "performSlowTask before")
    delay(5_000) // simulates long-running task
    Log.i(TAG, "performSlowTask after")
}
```

As implemented, all the function does is output diagnostic messages before and after performing a 5-second delay, simulating a long-running task. While the 5-second delay is in effect, the user interface will continue to be responsive because the main thread is not being blocked. To understand why it helps to explore what is happening behind the scenes.

First, the *startTask()* function is executed and launches the *performSlowTask()* suspend function as a coroutine. This function then calls the Kotlin *delay()* function passing through a time value. The built-in Kotlin *delay()* function is implemented as a suspend function, so it is also launched as a coroutine by the Kotlin runtime environment. The code execution has now reached what is referred to as a suspend point which will cause the *performSlowTask()* coroutine to be suspended while the delay coroutine is running. This frees up the thread on which *performSlowTask()* was running and returns control to the main thread so that the UI is unaffected.

Once the *delay()* function reaches completion, the suspended coroutine will be resumed and restored to a thread from the pool where it can display the Log message and return to the *startTask()* function.

When working with coroutines in Android Studio suspend points within the code editor are marked as shown in the figure below:

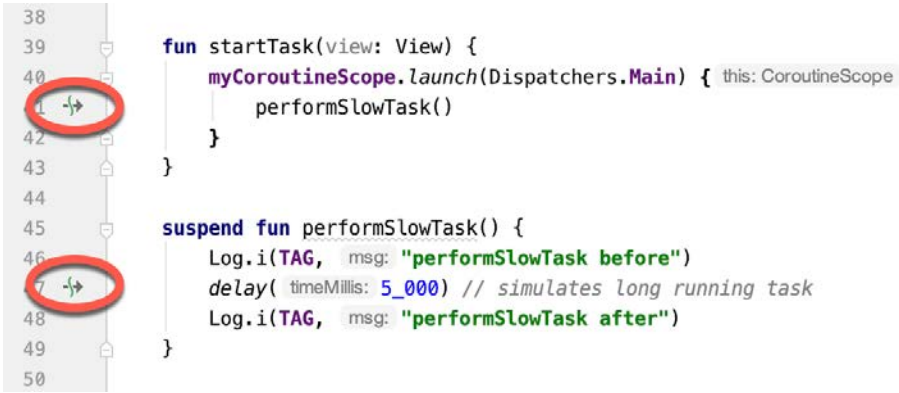

Figure 62-1

62.9 Returning Results from a Coroutine

The above example ran a suspend function as a coroutine but did not demonstrate how to return results. However, suppose the *performSlowTask()* function is required to return a string value to be displayed to the user via a TextView object.

To do this, we must rewrite the suspend function to return a Deferred object. A Deferred object is a commitment to provide a value at some point in the future. By calling the *await()* function on the Deferred object, the Kotlin runtime will deliver the value when the coroutine returns it. The code in our *startTask()* function might, therefore, be rewritten as follows:

```
fun startTask(view: View) {

    coroutineScope.launch(Dispatchers.Main) {
        statusText.text = performSlowTask().await()
    }
}
```

The problem now is that we are having to use the launch builder to start the coroutine since *startTask()* is not a suspend function. As outlined earlier in this chapter, it is only possible to return results when using the async builder. To get around this, we have to adapt the suspend function to use the async builder to start another coroutine that returns a Deferred result:

```
suspend fun performSlowTask(): Deferred<String> =
    coroutineScope.async(Dispatchers.Default) {
        Log.i(TAG, "performSlowTask before")
        delay(5_000)
        Log.i(TAG, "performSlowTask after")
    return@async "Finished"
}
```

When the app runs, the “Finished” result string will be displayed on the TextView object when the *performSlowTask()* coroutine completes. Once again, the wait for the result will occur in the background without blocking the main thread.

62.10 Using withContext

As we have seen, coroutines are launched within a specified scope and using a specific dispatcher. By default, any child coroutines will inherit the same dispatcher as that used by the parent. Consider the following code

designed to call multiple functions from within a suspend function:

```
fun startTask(view: View) {

    coroutineScope.launch(Dispatchers.Main) {
        performTasks()
    }
}

suspend fun performTasks() {
    performTask1()
    performTask2()
    performTask3()
}

suspend fun performTask1() {
    Log.i(TAG, "Task 1 ${Thread.currentThread().name}")
}

suspend fun performTask2() {
    Log.i(TAG, "Task 2 ${Thread.currentThread().name}")
}

suspend fun performTask3 () {
    Log.i(TAG, "Task 3 ${Thread.currentThread().name}")
}
```

Since the *performTasks()* function was launched using the Main dispatcher, all three functions will default to the main thread. To prove this, the functions have been written to output the name of the thread in which they are running. On execution, the Logcat panel will contain the following output:

```
Task 1 main
Task 2 main
Task 3 main
```

However, imagine that the *performTask2()* function performs network-intensive operations more suited to the IO dispatcher. This can easily be achieved using the withContext launcher, which allows the context of a coroutine to be changed while still staying in the same coroutine scope. The following change switches the *performTask2()* coroutine to an IO thread:

```
suspend fun performTasks() {
    performTask1()
    withContext(Dispatchers.IO) { performTask2() }
    performTask3()
}
```

When executed, the output will read as follows, indicating that the Task 2 coroutine is no longer on the main thread:

```
Task 1 main
Task 2 DefaultDispatcher-worker-1
```

```
Task 3 main
```

The withContext builder also provides an interesting alternative to using the async builder and the Deferred object *await()* call when returning a result. Using withContext, the code from the previous section can be rewritten as follows:

```
fun startTask(view: View) {

    coroutineScope.launch(Dispatchers.Main) {
          statusText.text = performSlowTask()
    }
}

suspend fun performSlowTask(): String =
    withContext(Dispatchers.Main) {
        Log.i(TAG, "performSlowTask before")
        delay(5_000)
        Log.i(TAG, "performSlowTask after")

        return@withContext "Finished"
    }
}
```

62.11 Coroutine Channel Communication

Channels provide a simple way to implement communication between coroutines, including streams of data. In the simplest form, this involves the creation of a Channel instance and calling the *send()* method to send the data. Once sent, transmitted data can be received in another coroutine via a call to the *receive()* method of the same Channel instance.

The following code, for example, passes six integers from one coroutine to another:

```
.
.
import kotlinx.coroutines.channels.*
.
.
val channel = Channel<Int>()

suspend fun channelDemo() {
    coroutineScope.launch(Dispatchers.Main) { performTask1() }
    coroutineScope.launch(Dispatchers.Main) { performTask2() }
}

suspend fun performTask1() {
    (1..6).forEach {
        channel.send(it)
    }
}
```

```
suspend fun performTask2() {
    repeat(6) {
        Log.d(TAG, "Received: ${channel.receive()}")
    }
}
```

When executed, the following logcat output will be generated:

```
Received: 1
Received: 2
Received: 3
Received: 4
Received: 5
Received: 6
```

62.12 Summary

Kotlin coroutines provide a simpler and more efficient approach to performing asynchronous tasks than traditional multi-threading. Coroutines allow asynchronous tasks to be implemented in a structured way without implementing the callbacks associated with typical thread-based tasks. This chapter has introduced the basic concepts of coroutines, including jobs, scope, builders, suspend functions, structured concurrency, and channel-based communication.

Chapter 71

71. Understanding Android Content Providers

The previous chapter worked on creating an example application designed to store data using a SQLite database. When implemented this way, the data is private to the application and, as such, inaccessible to other applications running on the same device. While this may be the desired behavior for many application types, situations will inevitably arise whereby the data stored on behalf of an application could benefit other applications. A prime example is the data stored by the built-in Contacts application on an Android device. While the Contacts application is primarily responsible for managing the user's address book details, this data is also made accessible to any other applications needing access. This data sharing between Android applications is achieved through implementing *content providers*.

71.1 What is a Content Provider?

A content provider provides access to structured data between different Android applications. This data is exposed to applications either as tables of data (in much the same way as a SQLite database) or as a handle to a file. This essentially involves the implementation of a client/server arrangement whereby the application seeking access to the data is the client and the content provider is the server, performing actions and returning results on behalf of the client.

A successful content provider implementation involves several elements, each of which will be covered in detail in the remainder of this chapter.

71.2 The Content Provider

A content provider is created as a subclass of the *android.content.ContentProvider* class. Typically, the application responsible for managing the data to be shared will implement a content provider to facilitate sharing of that data with other applications.

Creating a content provider involves implementing methods to manage the data on behalf of other client applications. These methods are as follows:

71.2.1 onCreate()

This method is called when the content provider is first created and should be used to perform any initialization tasks required by the content provider.

71.2.2 query()

This method will be called when a client requests that data be retrieved from the content provider. This method identifies the data to be retrieved (single or multiple rows), performs the data extraction, and returns the results wrapped in a Cursor object.

71.2.3 insert()

This method is called when a new row needs to be inserted into the provider database. This method must identify the destination for the data, perform the insertion and return the full URI of the newly added row.

71.2.4 update()

The method called when existing rows need to be updated on behalf of the client. The method uses the arguments passed through to update the appropriate table rows and return the number of rows updated as a result of the operation.

71.2.5 delete()

Called when rows are to be deleted from a table. This method deletes the designated rows and returns a count of the number of rows deleted.

71.2.6 getType()

Returns the MIME type of the data stored by the content provider.

It is important when implementing these methods in a content provider to keep in mind that, with the exception of the *onCreate()* method, they can be called from many processes simultaneously and must, therefore, be thread safe.

Once a content provider has been implemented, the issue that then arises is how the provider is identified within the Android system. This is where the *content URI* comes into play.

71.3 The Content URI

An Android device will potentially contain several content providers. The system must, therefore, provide some way of identifying one provider from another. Similarly, a single content provider may provide access to multiple forms of content (typically in the form of database tables). Client applications, therefore, need a way to specify the underlying data for which access is required. This is achieved using content URIs.

The content URI is used to identify specific data within a specific content provider. The Authority section of the URI identifies the content provider and usually takes the form of the package name of the content provider. For example:

```
com.example.mydbapp.myprovider
```

A specific database table within the provider data structure may be referenced by appending the table name to the authority. For example, the following URI references a table named *products* within the content provider:

```
com.example.mydbapp.myprovider/products
```

Similarly, a specific row within the specified table may be referenced by appending the row ID to the URI. The following URI, for example, references the row in the products table in which the value stored in the ID column equals 3:

```
com.example.mydbapp.myprovider/products/3
```

When implementing the insert, query, update and delete methods in the content provider, it will be the responsibility of these methods to identify whether the incoming URI is targeting a specific row in a table, or references multiple rows, and act accordingly. This can potentially be a complex task given that a URI can extend to multiple levels. This process can, however, be eased significantly using the *UriMatcher* class, as will be outlined in the next chapter.

71.4 The Content Resolver

Access to a content provider is achieved via a ContentResolver object. An application can obtain a reference to its content resolver by calling the *getContentResolver()* method of the application context.

The content resolver object contains a set of methods that mirror those of the content provider (insert, query, delete etc.). The application simply makes calls to the methods, specifying the URI of the content on which the

operation is to be performed. The content resolver and content provider objects then communicate to perform the requested task on behalf of the application.

71.5 The <provider> Manifest Element

For a content provider to be visible within an Android system, it must be declared within the Android manifest file for the application in which it resides. This is achieved using the <provider> element, which must contain the following items:

- **android:authority** – The full authority URI of the content provider. For example com.example.mydbapp.mydbapp.myprovider.
- **android:name** – The name of the class that implements the content provider. In most cases, this will use the same value as the authority.

Similarly, the <provider> element may be used to define the permissions that must be held by client applications in order to qualify for access to the underlying data. If no permissions are declared, the default behavior is for permission to be allowed for all applications.

Permissions can be set to cover the entire content provider, or limited to specific tables and records.

71.6 Summary

The data belonging to an application is typically private to the application and inaccessible to other applications. Setting up a content provider is necessary when the data needs to be shared. This chapter has covered the basic elements that combine to enable data sharing between applications and outlined the concepts of the content provider, content URI, and content resolver.

In the next chapter, the SQLDemo project created previously will be extended to make the underlying customer data available via a content provider.

Chapter 97

97. Working with Material Design 3 Theming

The appearance of an Android app is intended to conform to a set of guidelines defined by Material Design. Google developed Material Design to provide a level of design consistency between different apps while also allowing app developers to include their own branding in terms of color, typography, and shape choices (a concept referred to as Material theming). In addition to design guidelines, Material Design also includes a set of UI components for use when designing user interface layouts, many of which we have used throughout this book.

This chapter will provide an overview of how theming works within an Android Studio project and explore how the default design configurations provided for newly created projects can be modified to meet your branding requirements.

97.1 Material Design 2 vs. Material Design 3

Before beginning, it is important to note that Google is transitioning from Material Design 2 to Material Design 3 and that Android Studio Jellyfish projects default to Material Design 3. Material Design 3 provides the basis for Material You, a feature introduced in Android 12 that allows an app to automatically adjust theme elements to complement preferences configured by the user on the device. For example, dynamic color support provided by Material Design 3 allows the colors used in apps to adapt automatically to match the user's wallpaper selection.

97.2 Understanding Material Design Theming

We know that Android app user interfaces are created by assembling components such as layouts, text fields, and buttons. These components appear using default colors unless we specifically override a color attribute in the XML layout resource file or by writing code. The project's theme defines these default colors. The theme consists of a set of color slots (declared in *themes.xml* files) which are assigned color values (declared in the *colors.xml* file). Each UI component is programmed internally to use theme color slots as the default color for specific attributes (such as the foreground and background colors of the Text widget). It follows, therefore, that we can change the application-wide theme of an app by changing the colors assigned to specific theme slots. When the app runs, the new default colors will be used for all widgets when the user interface is rendered.

97.3 Material Design 3 Theming

Before exploring Material Design 3, we must consider how it is used in an Android Studio project. The theme used by an application project is declared as a property of the *application* element within the *AndroidManifest.xml* file, for example:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <application
.
.
```

```
        android:supportsRtl="true"
        android:theme="@style/Theme.MyDemoApp"
        tools:targetApi="31">
        <activity
.
.
```

As previously discussed, all of the files associated with the project theme are contained within the *colors.xml* and *themes.xml* files located in the *res* -> *values* folder, as shown in Figure 97-1:

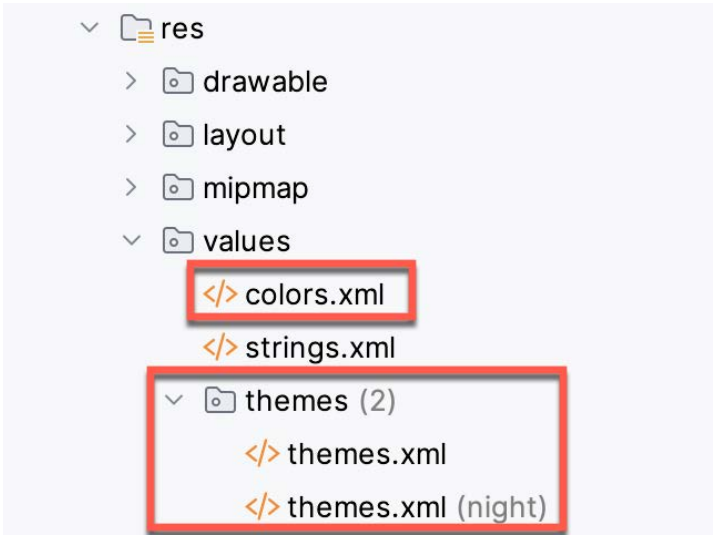

Figure 97-1

The theme itself is declared in the two *themes.xml* files located in the *themes* folder. These resource files declare different color palettes containing Material Theme color slots for use when the device is in light or dark (night) mode. Note that the style name property in each file must match that referenced in the *AndroidManifest.xml* file, for example:

```
<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- Base application theme. -->
    <style name="Base.Theme.MyDemoApp" parent="Theme.Material3.DayNight.
NoActionBar">
        <!-- Customize your light theme here. -->
        <!-- <item name="colorPrimary">@color/my_light_primary</item> -->
    </style>

    <style name="Theme.MyDemoApp" parent="Base.Theme.MyDemoApp" />
</resources>
```

These color slots (also referred to as *color attributes*) are used by the Material components to set colors when they are rendered on the screen. For example, the *colorPrimary* color slot is used as the background color for the Material Button component.

Color slots in MD3 are grouped as Primary, Secondary, Tertiary, Error, Background, and Surface. These slots are further divided into pairs consisting of a *base color* and an *"on" base color*. This generally translates to the background and foreground colors of a Material component.

The particular group used for coloring will differ between widgets. A Material Button widget, for example, will use the *colorPrimary* base color for the background color and *colorOnPrimary* for its content (i.e., the text or icon it displays). The FloatingActionButton component, on the other hand, uses *colorPrimaryContainer* as the background color and *colorOnPrimaryContainer* for the foreground. The correct group for a specific widget type can usually be identified quickly by changing color settings in the theme files and reviewing the rendering in the layout editor.

Suppose that we need to change *colorPrimary* to red. We achieve this by adding a new entry to the *colors.xml* file for the red color and then assigning it to the *colorPrimary* slot in the *themes.xml* file. The *colorPrimary* slot in an MD3 theme night, therefore, read as follows:

```
<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- Base application theme. -->
    <style name="Base.Theme.MyDemoApp" parent="Theme.Material3.DayNight.
NoActionBar">
        <item name="colorPrimary">@color/my_bright_primary</item>
    </style>

    <style name="Theme.MyDemoApp" parent="Base.Theme.MyDemoApp" />
</resources>
```

This color is then declared in the *colors.xml* file:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
.
.
    <color name="my_bright_primary">#FC0505</color>
</resources>
```

97.4 Building a Custom Theme

As we have seen, the coding work in implementing a theme is relatively simple. The difficult part, however, is often choosing complementary colors to make up the theme. Fortunately, Google has developed a tool that makes it easy to design custom color themes for your apps. This tool is called the Material Theme Builder and is available at:

https://m3.material.io/theme-builder#/custom

On the custom screen (Figure 97-2), make a color selection for the primary color key (A) by clicking on the color circle to display the color selection dialog. Once a color has been selected, the preview (B) will change to reflect the recommended colors for all MD3 color slots, along with example app interfaces and widgets. In addition, you can override the generated colors for the Secondary, Tertiary, and Neutral slots by clicking on the corresponding color circles to display the color selection dialog.

The area marked B displays example app interfaces, light and dark color scheme charts, and widgets that update to preview your color selections. Since the panel is longer than the typical browser window, you must scroll down to see all the information.

To incorporate the theme into your design, click the Export button (C) and select the Android View (XML) option. Once downloaded, the *colors.xml* and *themes.xml* files can be used to replace the existing files in your project. Note that the theme name in the two exported *themes.xml* files must be changed to match your project.

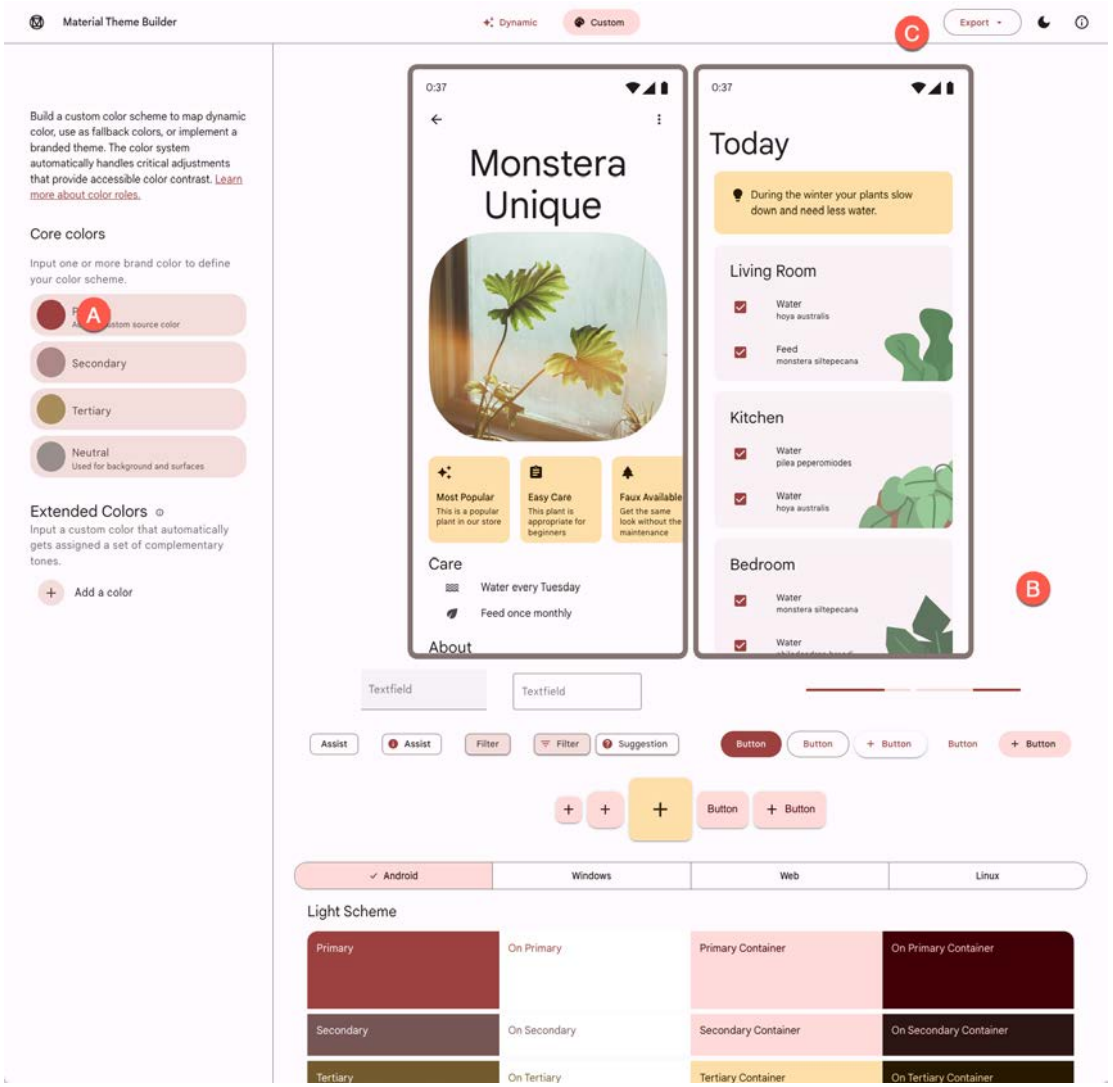

Figure 97-2

97.5 Summary

Material Design provides guidelines and components defining how Android apps appear. Individual branding can be applied to an app by designing themes that specify the colors, fonts, and shapes used when displaying the app. Google recently introduced Material Design 3, which replaces Material Design 2 and supports the new features of Material You, including dynamic colors. Google also provides the Material Theme Builder for designing your own themes, which eases the task of choosing complementary theme colors. Once this tool has been used to design a theme, the corresponding files can be exported and used within an Android Studio project.

Chapter 98

98. A Material Design 3 Theming and Dynamic Color Tutorial

This chapter will show you how to create a new Material Design 3 theme using the Material Theme Builder tool and integrate it into an Android Studio project. The tutorial will also demonstrate how to add support for and test dynamic theme colors to an app.

98.1 Creating the ThemeDemo Project

Select the *New Project* option from the welcome screen and, within the resulting new project dialog, choose the Empty Views Activity template before clicking on the Next button.

Enter *ThemeDemo* into the Name field and specify *com.ebookfrenzy.themedemo* as the package name. Before clicking on the Finish button, change the Minimum API level setting to API 26: Android 8.0 (Oreo) and the Language menu to Kotlin.

98.2 Designing the User Interface

The main activity will consist of a simple layout containing some user interface components that will enable us to see the effects of the theming work performed later in the chapter. For information on MD3 components, refer to the following web page:

https://material.io/blog/migrating-material-3

The layout will be designed within the *activity_main.xml* file, which currently contains a single Text view. Open this file in the layout editor, delete the Text view, turn off Autoconnect mode (marked A in Figure 98-1), and click on the button to clear all constraints from the layout (B).

Figure 98-1

From the Buttons section of the Palette, drag Chip, CheckBox, Switch, and Button views onto the layout canvas. Next, drag a FloatingActionButton onto the layout canvas to position it beneath the Button component. When prompted to choose an icon to appear on the FloatingActionButton, select the *ic_lock_power_off* icon from within the resource tool window, as illustrated in Figure 98-2:

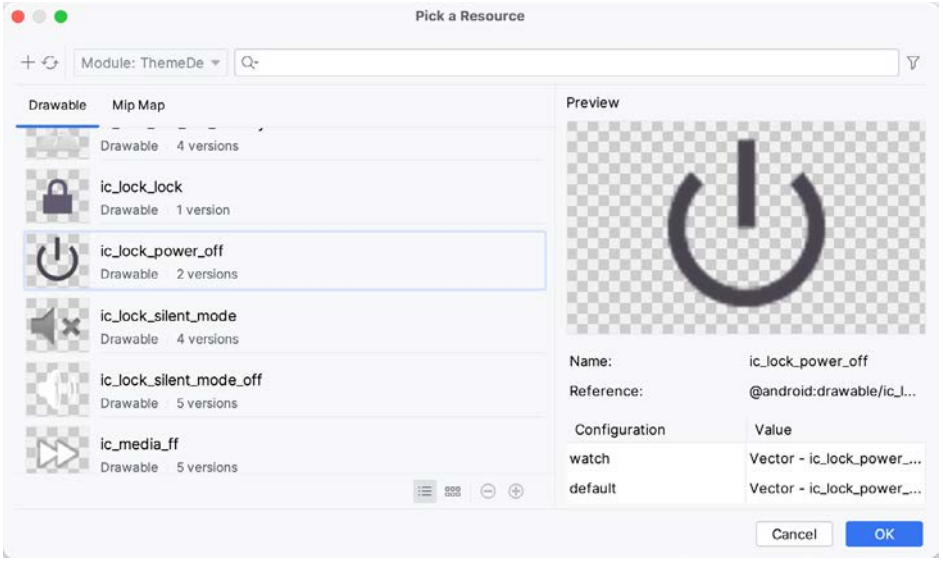

Figure 98-2

Change the text attribute for the Chip widget to "This is my chip" and set the chipIcon attribute to *@android:drawable/ic_btn_speak_now* so that the layout resembles that shown to the left in Figure 98-3:

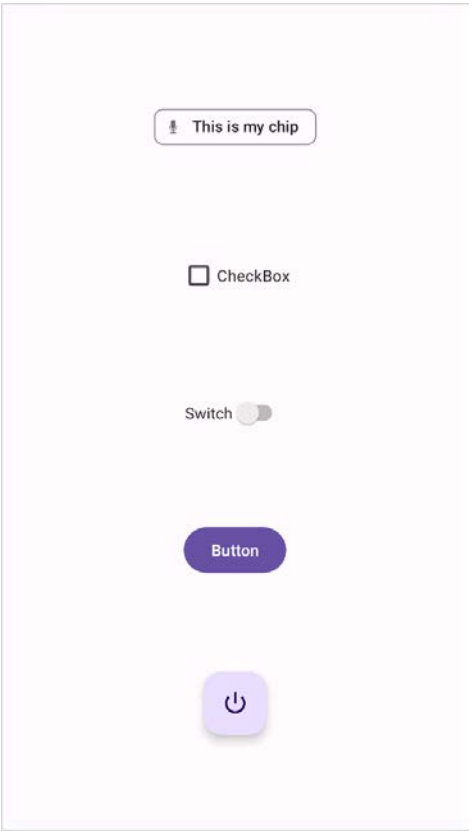

Figure 98-3

To set up the constraints, select all the components, right-click on the Chip view, and select *Chains -> Create Vertical Chain* from the resulting menu. Repeat this step, this time selecting the *Center -> Horizontally in Parent* menu option.

Compile and run the app on a device or emulator and verify that the user interface matches that shown in Figure 98-3 above. The next step is to create a custom theme and apply it to the project.

98.3 Building a New Theme

Begin by opening a browser window and navigating to the following URL to access the builder tool:

https://m3.material.io/theme-builder#/custom

Once you have loaded the builder, select a wallpaper followed by the Custom button at the top of the screen, and then click on the Primary color circle in the Core Colors section to display the color selector. From the color selector, choose any color as the basis for your theme:

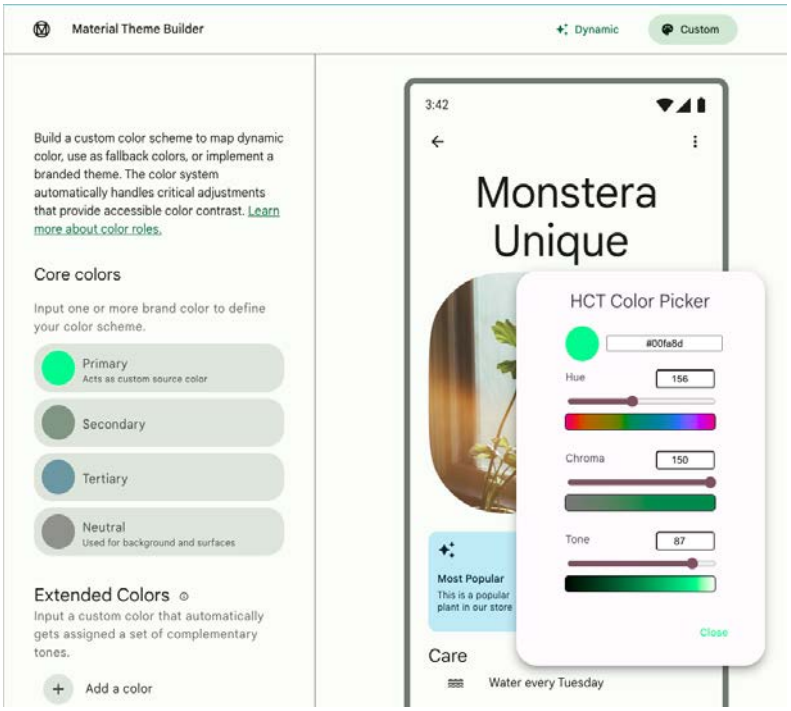

Figure 98-4

Review the color scheme in the Your Theme panel and make any necessary color adjustments using the Core colors settings until you are happy with the color slots. Once the theme is ready, click the Export button in the top right-hand corner and select the *Android Views (XML)* option. When prompted, save the file to a suitable location on your computer filesystem. The theme will be saved as a compressed file named *material-theme.zip*.

Using the appropriate tool for your operating system, unpack the theme file, which should contain the following folders and files in a folder named *material-theme*:

- **values/colors.xml** - The color definitions.
- **values/themes.xml** - The theme for the light mode.
- **values-night/themes.xml** - The theme for dark mode.

Now that the theme files have been generated, they need to be integrated into the Android Studio project.

98.4 Adding the Theme to the Project

Before adding the new theme to the project, we first need to remove the old theme files. This is easier if the Project tool window is in Project Files mode. To switch mode, use the menu at the top of the tool Project tool window as shown below and select the Project Files option:

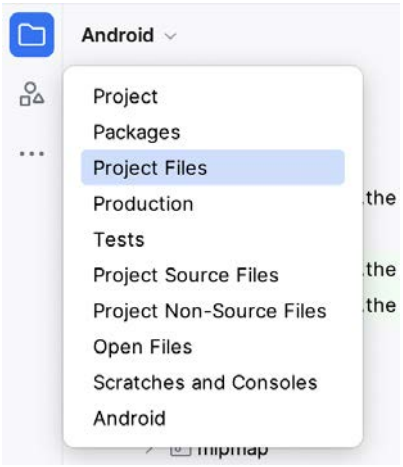

Figure 98-5

With Project Files mode selected, navigate to the *app* -> *src* -> *main* -> *res* -> *values* folder and select and delete the *colors.xml* and *themes.xml* files. Also, delete the *themes.xml* file located in the *values-night* folder.

Open the filesystem navigation tool for your operating system, locate the *colors.xml* and *themes.xml* files in the *values* folder of the new material theme, and copy and paste them into the *values* folder within the Project tool window. Repeat this step to copy the *themes.xml* file in the *values-night* folder, this time pasting it into the *values-night* folder.

Switch the Project tool window back to Android mode, at which point the value resource files section should match Figure 98-6:

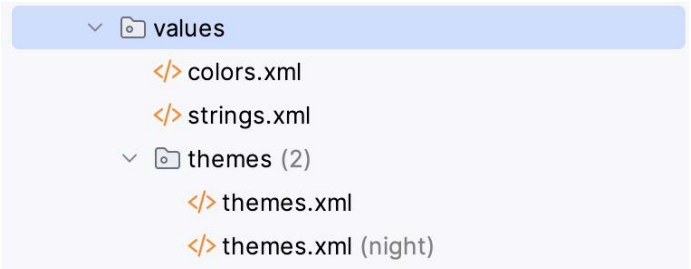

Figure 98-6

Next, modify the light *themes.xml* file to match the current project as follows:

```
<resources>
    <style name="Base.Theme.ThemeDemo" parent="Theme.Material3.Light.
NoActionBar">
        <item name="colorPrimary">@color/md_theme_light_primary</item>
        <item name="colorOnPrimary">@color/md_theme_light_onPrimary</item>
```

```
.
.
    </style>

    <style name="Theme.ThemeDemo" parent="Base.Theme.ThemeDemo" />

</resources>
```

Repeat these steps to make the same modifications to the *themes.xml (night)* file.

Return to the *activity_main.xml* file or rerun the app to confirm that the user interface is rendered using the custom theme colors.

98.5 Enabling Dynamic Color Support

The app will need to be run on a device or emulator running Android 12 or later with the correct Wallpaper settings to test dynamic colors. On the device or emulator, launch the Settings app and select *Wallpaper & style* from the list of options. On the wallpaper settings screen, click the option to change the wallpaper (marked A in Figure 98-7) and select a wallpaper image containing colors that differ significantly from the colors in your theme. Once selected, assign the wallpaper to the Home screen.

Return to the Wallpaper & styles screen and make sure that the *Wallpaper colors* option is selected (B) before choosing an option from the color scheme buttons (C). As each option is clicked, the wallpaper example will change to reflect the selection:

Figure 98-7

To enable dynamic colors, we need to call the *applyToActivitiesIfAvailable()* method of the DynamicColors class. To enable dynamic color support for the entire app, this needs to be called from within the *onCreate()* method of a custom Application instance. Begin by adding a new Kotlin class file to the project under *app -> kotlin+java -> com.ebookfrenzy.themedemo* named *ThemeDemoApplication.kt* and modifying it so that it reads as follows:

```
package com.ebookfrenzy.themedemo
```

```
import android.app.Application
import com.google.android.material.color.DynamicColors

class ThemeDemoApplication: Application() {
    override fun onCreate() {
        super.onCreate()
        DynamicColors.applyToActivitiesIfAvailable(this)
    }
}
```

With the custom Application class created, we must configure the project to use this class instead of the default Application instance. To do this, edit the *AndroidManifest.xml* file and add an *android:name* element referencing the new class:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.ebookfrenzy.themedemo">

    <application
        android:name=".ThemeDemoApplication"
        android:allowBackup="true"
.
.
```

Build and run the app and note that the layout uses a theme matching the wallpaper color. Place the ThemeDemo app into the background, return to the *Wallpaper & styles* settings screen, and choose a different wallpaper. Bring the ThemeDemo app to the foreground again. At this point, it will have dynamically adapted to match the new wallpaper.

98.6 Previewing Dynamic Colors

Dynamic color behavior can also be previewed within the Android Studio layout editor. To try this, open the *activity_main.xml* file, click on the theme menu, and select the More Themes option, as shown in Figure 98-8:

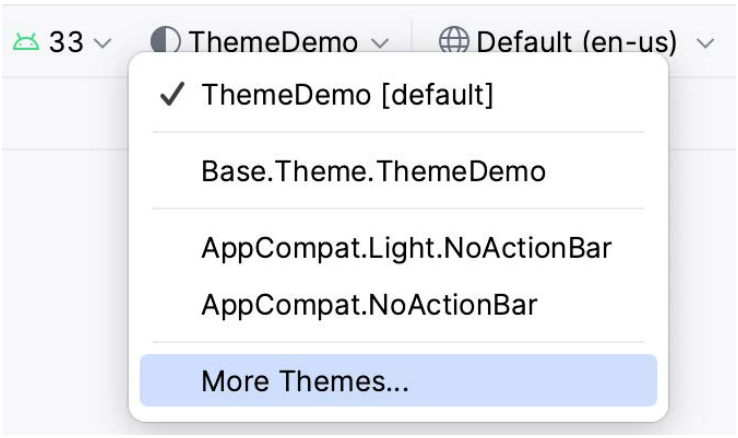

Figure 98-8

Next, use the search field in the theme selection dialog to list dynamic themes:

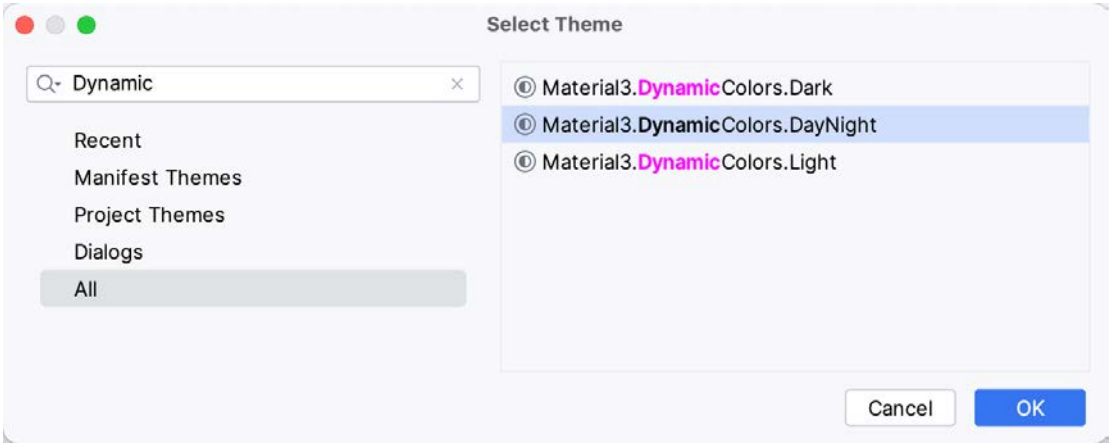

Figure 98-9

Select the *Material3.DynamicColors.DayNight* theme before clicking on the OK button. On returning to the layout editor, select the System UI Mode menu and choose one of the wallpaper options as highlighted in Figure 98-10:

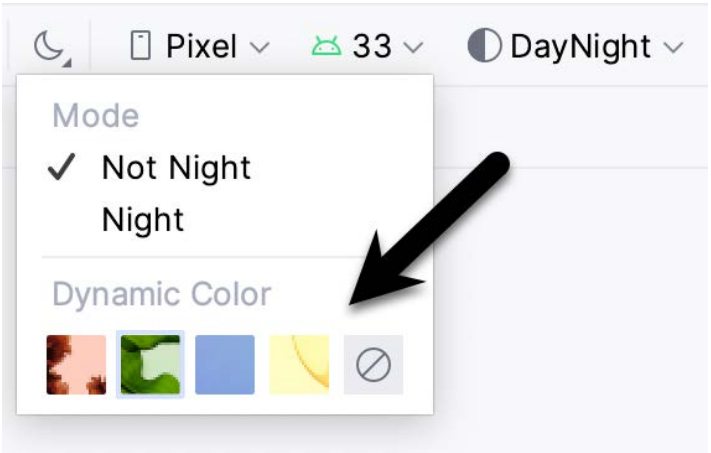

Figure 98-10

Once a wallpaper has been selected, the colors of the components in the layout will change accordingly.

98.7 Summary

In this chapter, we have used the Material Theme Builder to design a new theme and explained the steps to integrate the generated theme files into an Android Studio project. Finally, the chapter demonstrated how to implement and use the Material You dynamic colors feature.

Chapter 99

99. An Overview of Gradle in Android Studio

In the *"A Guide to Gradle Version Catalogs"* chapter, we introduced the library version catalog and explained how the Gradle build system relies on it to ensure that projects are built using the correct libraries and versions. Aside from some modifications to the version catalog and library decencies in the intervening chapters, it has been taken for granted that Android Studio will take the necessary steps to compile and run the application projects that have been created. Android Studio has been achieving this in the background using a system known as *Gradle*.

It is time to look at how Gradle is used to compile and package an application project's various elements and begin exploring how to configure this system when more advanced requirements are needed for building projects in Android Studio.

99.1 An Overview of Gradle

Gradle is an automated build toolkit that allows how projects are built to be configured and managed through a set of build configuration files. This includes defining how a project will be built, what dependencies need to be fulfilled to build successfully, and what the build process's end result (or results) should be.

The strength of Gradle lies in the flexibility that it provides to the developer. The Gradle system is a self-contained, command-line-based environment that can be integrated into other environments using plugins. In the case of Android Studio, Gradle integration is provided through the appropriately named Android Studio Plugin.

Although the Android Studio Plug-in allows Gradle tasks to be initiated and managed from within Android Studio, the Gradle command-line wrapper can still be used to build Android Studio-based projects, including on systems on which Android Studio is not installed.

The configuration rules to build a project are declared in Gradle build files and scripts based on the Groovy programming language.

99.2 Gradle and Android Studio

Gradle brings many powerful features to building Android application projects. Some of the key features are as follows:

99.2.1 Sensible Defaults

Gradle implements a concept referred to as *convention over configuration*. This means that Gradle has a predefined set of sensible default configuration settings that will be used unless settings in the build files override them. This means that builds can be performed with the minimum configuration required by the developer. Changes to the build files are only needed when the default configuration does not meet your build needs.

99.2.2 Dependencies

Another key area of Gradle functionality is that of dependencies. Consider, for example, a module within an Android Studio project which triggers an intent to load another module in the project. The first module has, in effect, a dependency on the second module since the application will fail to build if the second module cannot be located and launched at runtime. This dependency can be declared in the Gradle build file for the first module

so that the second module is included in the application build, or an error flagged if the second module cannot be found or built. Other examples of dependencies are libraries and JAR files on which the project depends to compile and run.

Gradle dependencies can be categorized as *local* or *remote.* A local dependency references an item that is present on the local file system of the computer system on which the build is being performed. A remote dependency refers to an item that is present on a remote server (typically referred to as a *repository*).

Remote dependencies are handled for Android Studio projects using another project management tool named *Maven*. If a remote dependency is declared in a Gradle build file using Maven syntax, then the dependency will be downloaded automatically from the designated repository and included in the build process. The following dependency declaration, for example, causes the AppCompat library to be added to the project from the Google repository:

```
implementation(libs.androidx.appcompat)
```

99.2.3 Build Variants

In addition to dependencies, Gradle also provides *build variant* support for Android Studio projects. This allows multiple variations of an application to be built from a single project. Android runs on many different devices encompassing a range of processor types and screen sizes. To target as wide a range of device types and sizes as possible, it will often be necessary to build several variants of an application (for example, one with a user interface for phones and another for tablet-sized screens). Through the use of Gradle, this is now possible in Android Studio.

99.2.4 Manifest Entries

Each Android Studio project has associated with it an *AndroidManifest.xml* file containing configuration details about the application. Several manifest entries can be specified in Gradle build files which are then auto-generated into the manifest file when the project is built. This capability complements the build variants feature, allowing elements such as the application version number, application ID, and SDK version information to be configured differently for each build variant.

99.2.5 APK Signing

The chapter *"Creating, Testing, and Uploading an Android App Bundle"* covered creating a signed release APK file using the Android Studio environment. It is also possible to include the signing information entered through the Android Studio user interface within a Gradle build file to generate signed APK files from the command line.

99.2.6 ProGuard Support

ProGuard is a tool included with Android Studio that optimizes, shrinks, and obfuscates Java byte code to make it more efficient and harder to reverse engineer (the method by which others can identify the logic of an application through analysis of the compiled Java byte code). The Gradle build files allow you to control whether or not ProGuard is run on your application when it is built.

99.3 The Property and Settings Gradle Build File

The gradle build configuration consists of configuration, property, and settings files. The *gradle.properties* file, for example, contains mostly esoteric settings relating to the command-line flags used by the Java Virtual Machine (JVM), whether or not the project uses the AndroidX libraries and Kotlin coding style support. As a typical user, it is unlikely that you will need to change any of these settings in this file.

The *settings.gradle.kts* file, on the other hand, defines which online repositories are to be searched when the build system needs to download and install any additional libraries and plugins required to build the project and the project name. A typical *settings.gradle.kts* file will read as follows:

```
pluginManagement {
```

```
    repositories {
        google()
        mavenCentral()
        gradlePluginPortal()
    }
}
dependencyResolutionManagement {
    repositoriesMode.set(RepositoriesMode.FAIL_ON_PROJECT_REPOS)
    repositories {
        google()
        mavenCentral()
    }
}

rootProject.name = "ThemeDemo"
include(":app")
```

As with the *gradle.properties* file, it is unlikely that changes will need to be made to this file.

99.4 The Top-level Gradle Build File

A completed Android Studio project contains everything needed to build an Android application and consists of modules, libraries, manifest files, and Gradle build files.

Each project contains one top-level Gradle build file. This file is listed as *build.gradle.kts (Project: <project name>)* and can be found in the project tool window as highlighted in Figure 99-1:

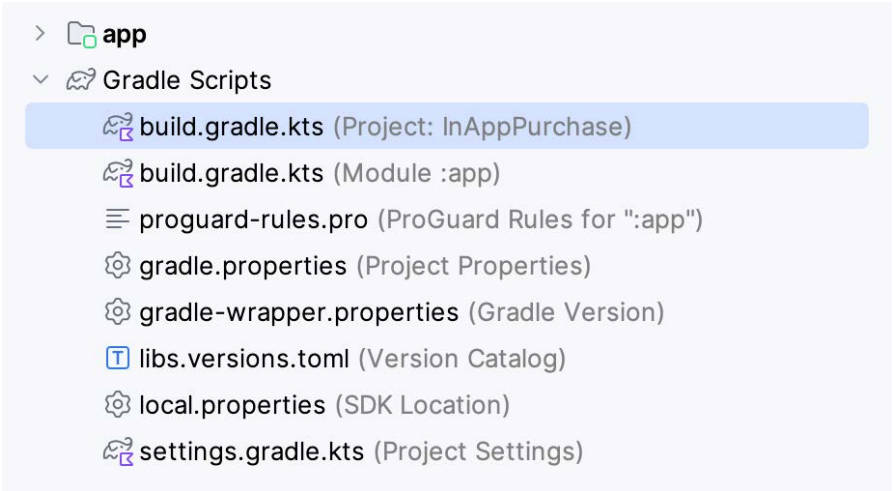

Figure 99-1

By default, the contents of the top-level Gradle build file reads as follows:

```
plugins {
    alias(libs.plugins.androidApplication) apply false
    alias(libs.plugins.jetbrainsKotlinAndroid) apply false
}
```

In most situations, making any changes to this build file is unnecessary.

99.5 Module Level Gradle Build Files

An Android Studio application project is made up of one or more modules. Take, for example, a hypothetical application project named GradleDemo which contains modules named Module1 and Module2, respectively. In this scenario, each module will require its own Gradle build file. In terms of the project structure, these would be located as follows:

- Module1/build.gradle.kts
- Module2/build.gradle.kts

By default, the Module1 *build.gradle.kts* file would resemble that of the following listing:

```
plugins {
    alias(libs.plugins.androidApplication)
    alias(libs.plugins.jetbrainsKotlinAndroid)
}

android {
    namespace = "com.example.gradlesample"
    compileSdk = 34

    defaultConfig {
        applicationId = "com.example.gradlesample"
        minSdk = 26
        targetSdk = 34
        versionCode = 1
        versionName = "1.0"

        testInstrumentationRunner = "androidx.test.runner.AndroidJUnitRunner"
    }

    buildTypes {
        release {
            isMinifyEnabled = false
            proguardFiles(
                getDefaultProguardFile("proguard-android-optimize.txt"),
                "proguard-rules.pro"
            )
        }
    }
    compileOptions {
        sourceCompatibility = JavaVersion.VERSION_1_8
        targetCompatibility = JavaVersion.VERSION_1_8
    }
    kotlinOptions {
        jvmTarget = "1.8"
```

```
    }
}

dependencies {

    implementation(libs.androidx.core.ktx)
    implementation(libs.androidx.appcompat)
    implementation(libs.material)
    implementation(libs.androidx.activity)
    implementation(libs.androidx.constraintlayout)
    testImplementation(libs.junit)
    androidTestImplementation(libs.androidx.junit)
    androidTestImplementation(libs.androidx.espresso.core)
}
```

As is evident from the file content, the build file begins by declaring the use of the Gradle Android application and Kotlin plug-ins:

```
plugins {
    alias(libs.plugins.androidApplication)
    alias(libs.plugins.jetbrainsKotlinAndroid)
}
```

The *android* section of the file declares the project namespace and then states the version of the SDK to be used when building Module1.

```
android {
    namespace = "com.example.gradlesample"
    compileSdk = 34
```

The items declared in the defaultConfig section define elements to be generated into the module's *AndroidManifest.xml* file during the build. These settings, which may be modified in the build file, are taken from the settings entered within Android Studio when the module was first created:

```
defaultConfig {
    applicationId = "com.example.gradlesample"
    minSdk = 26
    targetSdk = 34
    versionCode = 1
    versionName = "1.0"

    testInstrumentationRunner = "androidx.test.runner.AndroidJUnitRunner"
}
```

The buildTypes section contains instructions on whether and how to run ProGuard on the APK file when a release version of the application is built:

```
buildTypes {
    release {
        isMinifyEnabled = false
        proguardFiles(
```

```
            getDefaultProguardFile("proguard-android-optimize.txt"),
            "proguard-rules.pro"
        )
    }
}
```

As currently configured, ProGuard will not be run when Module1 is built. To enable ProGuard, the *minifyEnabled* entry must be changed from *false* to *true*. The *proguard-rules.pro* file can be found in the module directory of the project. Changes made to this file override the default settings in the *proguard-android.txt* file, which is located in the Android SDK installation directory under *sdk/tools/proguard*.

Since no debug buildType is declared in this file, the defaults will be used (built without ProGuard, signed with a debug key, and debug symbols enabled).

An additional section, entitled *productFlavors*, may also be included in the module build file to enable multiple build variants to be created.

Next, directives are included to specify the version of the Java compiler to be used when building the project:

```
compileOptions {
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
}
kotlinOptions {
    jvmTarget = "1.8"
}
```

Finally, the dependencies section lists any local and remote dependencies on which the module depends. The dependency lines in the above example file designate the Android libraries that need to be included from the Android Repository:

```
dependencies {

    implementation(libs.androidx.core.ktx)
    implementation(libs.androidx.appcompat)
    implementation(libs.material)
.
.
}
```

Note that the dependency declarations include version numbers to indicate which library version should be included.

99.6 Configuring Signing Settings in the Build File

The *"Creating, Testing, and Uploading an Android App Bundle"* chapter of this book covered the steps involved in setting up keys and generating a signed release APK file using the Android Studio user interface. These settings may also be declared within a *signingConfigs* section of the *build.gradle.kts* file. For example:

```
.
.
    defaultConfig {
.
```

```
.
    }
    signingConfigs {
        release {
            storeFile file("keystore.release")
            storePassword "your keystore password here"
            keyAlias "your key alias here"
            keyPassword "your key password here"
        }
    }
    buildTypes {
.
.
}
```

The above example embeds the key password information directly into the build file. An alternative to this approach is to extract these values from system environment variables:

```
signingConfigs {
    release {
        storeFile file("keystore.release")
        storePassword System.getenv("KEYSTOREPASSWD")
        keyAlias "your key alias here"
        keyPassword System.getenv("KEYPASSWD")
    }
}
```

Yet another approach is to configure the build file so that Gradle prompts for the passwords to be entered during the build process:

```
signingConfigs {
    release {
        storeFile file("keystore.release")
        storePassword System.console().readLine
                 ("\nEnter Keystore password: ")
        keyAlias "your key alias here"
        keyPassword System.console().readLIne("\nEnter Key password: ")
    }
}
```

99.7 Running Gradle Tasks from the Command Line

Each Android Studio project contains a Gradle wrapper tool to invoke Gradle tasks from the command line. This tool is located in the root directory of each project folder. While this wrapper is executable on Windows systems, it may need to have execute permission enabled on Linux and macOS before it can be used. To enable execute permission, open a terminal window, change directory to the project folder for which the wrapper is needed, and execute the following command:

```
chmod +x gradlew
```

Once the file has execute permissions, the location of the file will either need to be added to your $PATH environment variable or the name prefixed by ./ to run. For example:

```
./gradlew tasks
```

Gradle views project building in terms of several different tasks. A full listing of tasks that are available for the current project can be obtained by running the following command from within the project directory (remembering to prefix the command with a ./ if running on macOS or Linux):

```
gradlew tasks
```

To build a debug release of the project suitable for device or emulator testing, use the assembleDebug option:

```
gradlew assembleDebug
```

Alternatively, to build a release version of the application:

```
gradlew assembleRelease
```

99.8 Summary

For the most part, Android Studio performs application builds in the background without any intervention from the developer. This build process is handled using the Gradle system, an automated build toolkit designed to allow how projects are built to be configured and managed through a set of build configuration files. While the default behavior of Gradle is adequate for many basic project build requirements, the need to configure the build process is inevitable with more complex projects. This chapter has provided an overview of the Gradle build system and configuration files within the context of an Android Studio project.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z