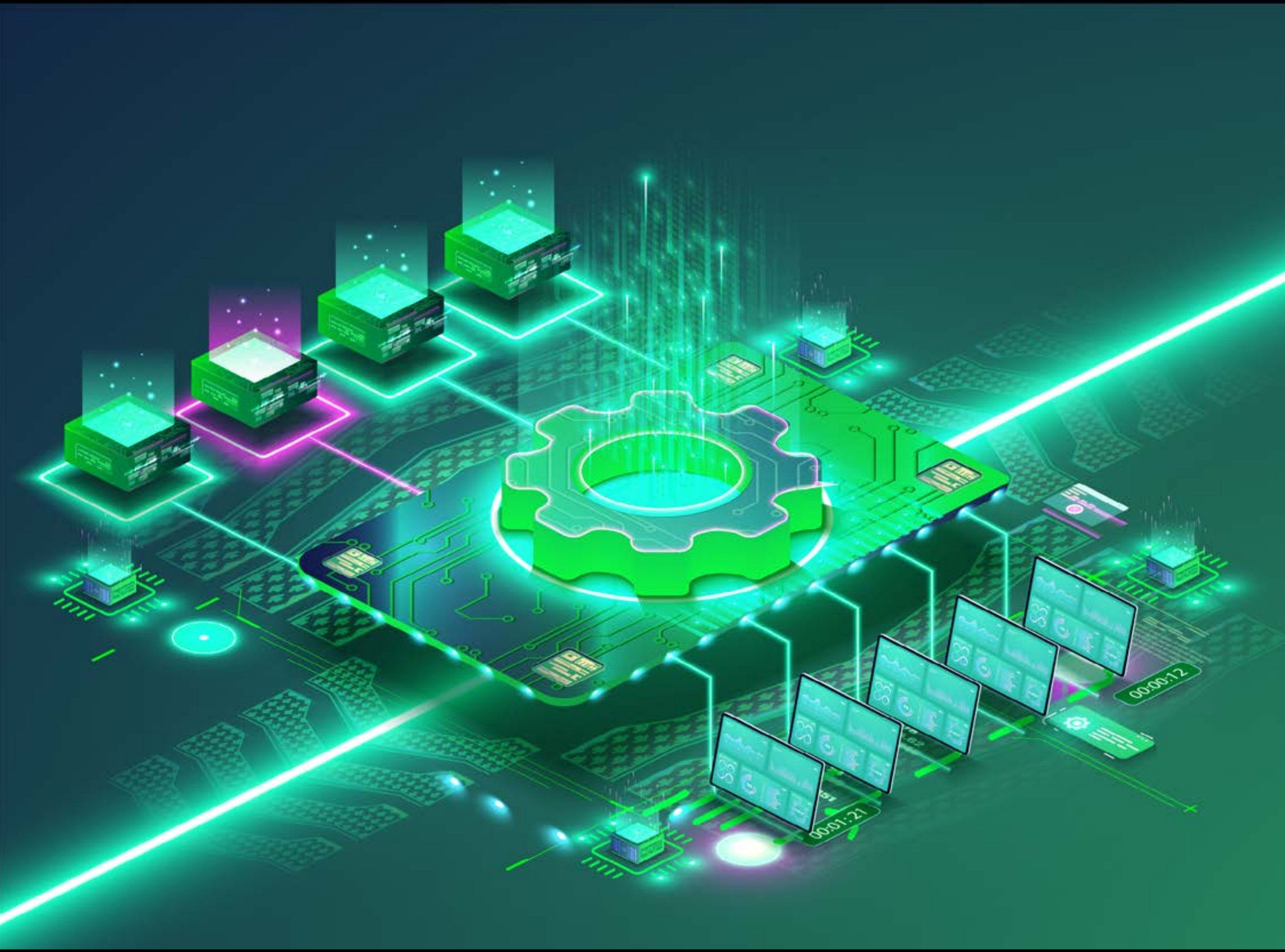
Jetpack Compose 1.6 Essentials



Payload publishing

Jetpack Compose 1.6 Essentials

Jetpack Compose 1.6 Essentials

ISBN-13: 978-1-951442-91-0

© 2024 Neil Smyth / Payload Media, Inc. All Rights Reserved.

This book is provided for personal use only. Unauthorized use, reproduction and/or distribution strictly prohibited. All rights reserved.

The content of this book is provided for informational purposes only. Neither the publisher nor the author offers any warranties or representation, express or implied, with regard to the accuracy of information contained in this book, nor do they accept any liability for any loss or damage arising from any errors or omissions.

This book contains trademarked terms that are used solely for editorial purposes and to the benefit of the respective trademark owner. The terms used within this book are not intended as infringement of any trademarks.

Rev: 1.0

Payload publishing

Find more books at *https://www.payloadbooks.com.*

Table of Contents

Chapter 1

1. Start Here

This book teaches you how to build Android applications using Jetpack Compose 1.6, Android Studio Iguana (2023.2.1), Material Design 3, and the Kotlin programming language.

The book begins with the basics by explaining how to set up an Android Studio development environment.

The book also includes in-depth chapters introducing the Kotlin programming language, including data types, operators, control flow, functions, lambdas, coroutines, and object-oriented programming.

An introduction to the key concepts of Jetpack Compose and Android project architecture is followed by a guided tour of Android Studio in Compose development mode. The book also covers the creation of custom Composables and explains how functions are combined to create user interface layouts, including row, column, box, flow, pager, and list components.

Other topics covered include data handling using state properties and key user interface design concepts such as modifiers, navigation bars, and user interface navigation. Additional chapters explore building your own reusable custom layout components, securing your apps with Biometric authentication, and integrating Google Maps.

The book covers graphics drawing, user interface animation, transitions, Kotlin Flows, and gesture handling.

Chapters also cover view models, SQLite databases, Room database access, the Database Inspector, live data, and custom theme creation. You will also learn to generate extra revenue from your app using in-app billing.

Finally, the book explains how to package up a completed app and upload it to the Google Play Store for publication.

Along the way, the topics covered in the book are put into practice through detailed tutorials, the source code for which is also available for download.

Assuming you already have some rudimentary programming experience, are ready to download Android Studio and the Android SDK, and have access to a Windows, Mac, or Linux system, you are ready to start.

1.1 For Kotlin programmers

This book addresses the needs of existing Kotlin programmers and those new to Kotlin and Jetpack Compose app development. If you are familiar with the Kotlin programming language, you can probably skip the Kotlin-specific chapters.

1.2 For new Kotlin programmers

If you are new to Kotlin programming, the entire book is appropriate for you. Just start at the beginning and keep going.

1.3 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/compose16/

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

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

1.4 Feedback

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

1.5 Errata

While we make every effort to ensure the accuracy of the content of this book, inevitably, 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/compose16_errata

If you find an error not listed in the errata, email our technical support team at *info@payloadbooks.com*.

1.6 Find more books

Visit *https://www.payloadbooks.com* to view our complete book catalog.

1.7 Authors wanted

Payload Publishing is looking for authors.

Are you an aspiring author with a book idea in mind? When you publish with us, you'll receive our full support every step of the way. We offer guidance and technical and editorial assistance to help you bring your book to life. Once your book is completed, we will publish and market it worldwide through our distribution and channel partnerships while paying you higher royalties than traditional publishers.

Find out more at:

https://www.payloadbooks.com/authors-wanted

or email us at:

authors@payloadbooks.com

Chapter 3

3. A Compose Project Overview

Now that we have installed Android Studio, the next step is to create an Android app using Jetpack Compose. Although this project will use several Compose features, it is an intentionally simple example intended to provide an early demonstration of Compose in action and an initial success on which to build as you work through the remainder of the book. The project will also verify that your Android Studio environment is correctly installed and configured.

This chapter will create a new project using the Android Studio Compose project template and explore both the basic structure of a Compose-based Android Studio project and some of the key areas of Android Studio. The next chapter will use this project to create a simple Android app.

Both chapters will briefly explain key features of Compose as they are introduced within the project. If anything is unclear when you have completed the project, rest assured that all the areas covered in the tutorial will be explored in greater detail in later chapters of the book.

3.1 About the project

The completed project will consist of two text components and a slider. When the slider is moved, the current value will be displayed on one of the text components, while the font size of the second text instance will adjust to match the current slider position. Once completed, the user interface for the app will appear as shown in Figure 3-1:

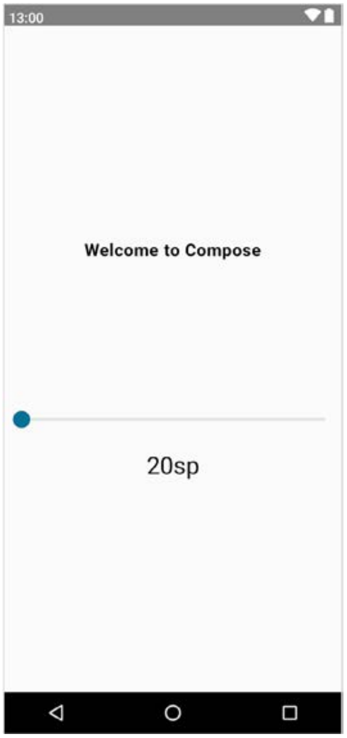


Figure 3-1

3.2 Creating the project

The first step in building an app is to create a new project within Android Studio. Begin, therefore, by launching Android Studio so that the "Welcome to Android Studio" screen appears as illustrated in Figure 3-2:

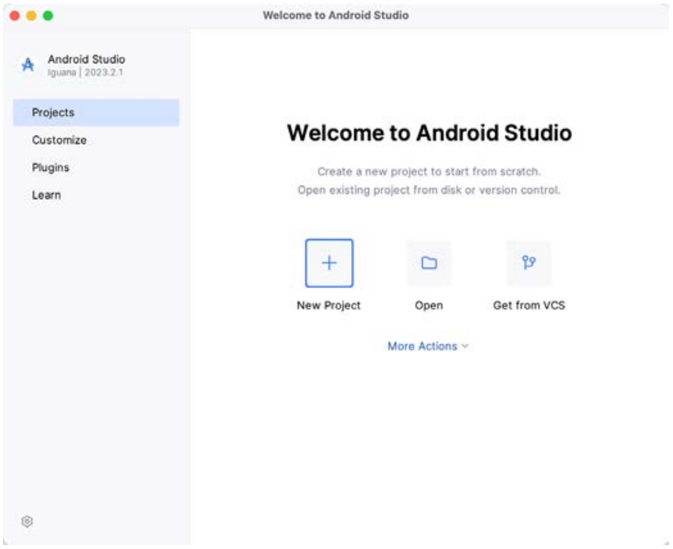

Welcome to Android Studio
Android Studio
Iguana | 2023.2.1
Projects
Customize
Plugins
Learn
Welcome to Android Studio
Create a new project to start from scratch.
Open existing project from disk or version control.
New Project
Open
Get from VCS
More Actions

Figure 3-2

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* button 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 that is to be created for the application. The left-hand panel provides a list of platform categories from which the *Phone and Tablet* option must be selected. Although various activity types are available when developing Android applications, only the *Empty Activity* template provides a pre-configured project ready to work with Compose. Select this option before clicking on the *Next* button:

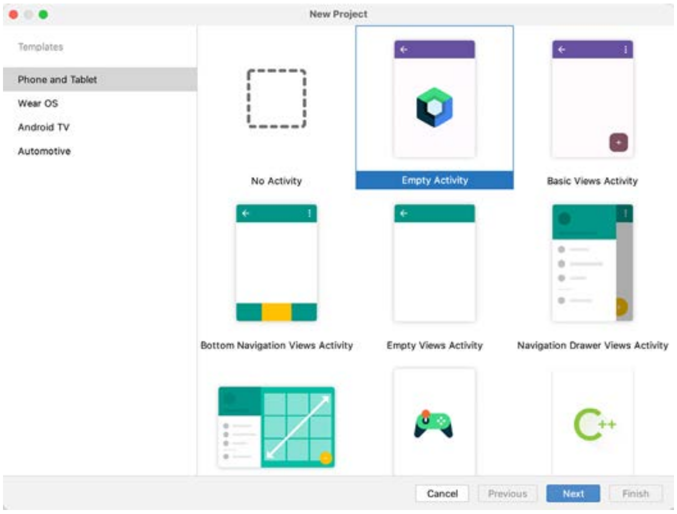
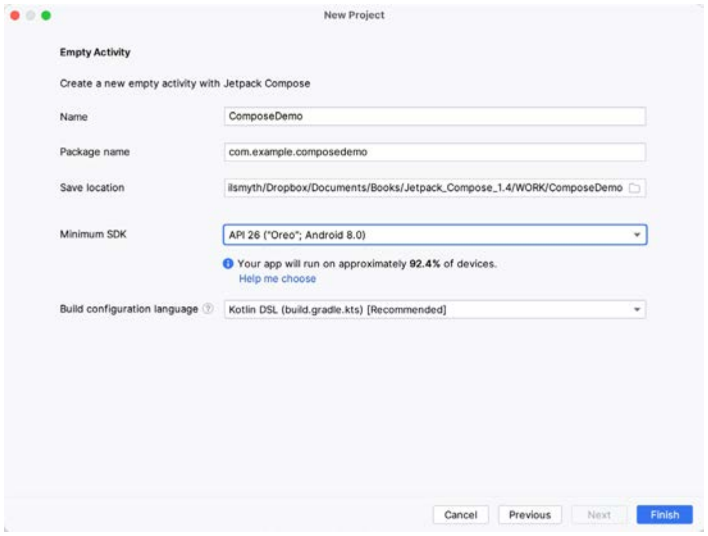


Figure 3-3

3.4 Defining the project and SDK settings

In the project configuration window (Figure 3-4), set the *Name* field to *ComposeDemo*. 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:


New Project

Empty Activity

Create a new empty activity with Jetpack Compose

Name: ComposeDemo

Package name: com.example.composedemo

Save location: ilsmyth/Dropbox/Documents/Books/Jetpack_Compose_1.4/WORK/ComposeDemo

Minimum SDK: API 26 ("Oreo"; Android 8.0)

Your app will run on approximately **92.4%** of devices.
Help me choose

Build configuration language: Kotlin DSL (build.gradle.kts) [Recommended]

Figure 3-4

The *Package name* uniquely identifies the application within the Google Play app store 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 *ComposeDemo*, then the package name might be specified as follows:

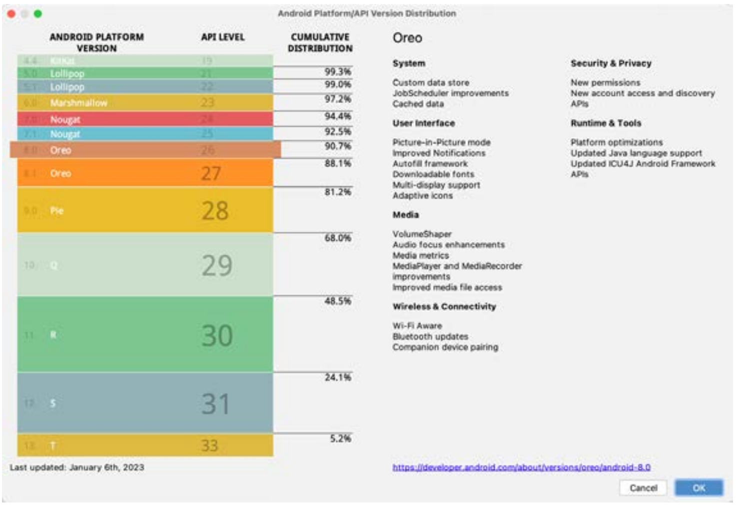
```
com.mycompany.composedemo
```

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.composedemo
```

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: Android 8.0 (Oreo). This is the minimum SDK that 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 also 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* link to see a full breakdown of the various Android versions still in use:


Android Platform/API Version Distribution

ANDROID PLATFORM VERSION		API LEVEL	CUMULATIVE DISTRIBUTION
4.4	KitKat	19	
5.0	Lollipop	21	99.3%
5.1	Lollipop	22	99.0%
6.0	Marshmallow	23	97.2%
7.0	Nougat	24	94.4%
7.1	Nougat	25	92.5%
8.0	Oreo	26	90.7%
8.1	Oreo	27	88.1%
9.0	Pie	28	81.2%
10	Q	29	68.0%
11	R	30	48.5%
12	S	31	24.1%
13	T	33	5.2%

Last updated: January 6th, 2023

Oreo

System

Custom data store
JobScheduler improvements
Cached data

User Interface

Picture-in-Picture mode
Improved Notifications
Autofill framework
Downloadable fonts
Multi-display support
Adaptive icons

Media

VolumeShaper
Audio focus enhancements
Media metrics
MediaPlayer and MediaRecorder improvements
Improved media file access

Wireless & Connectivity

Wi-Fi Aware
Bluetooth updates
Companion device pairing

Security & Privacy

New permissions
New account access and discovery APIs

Runtime & Tools

Platform optimizations
Updated Java language support
Updated ICU4J Android Framework APIs

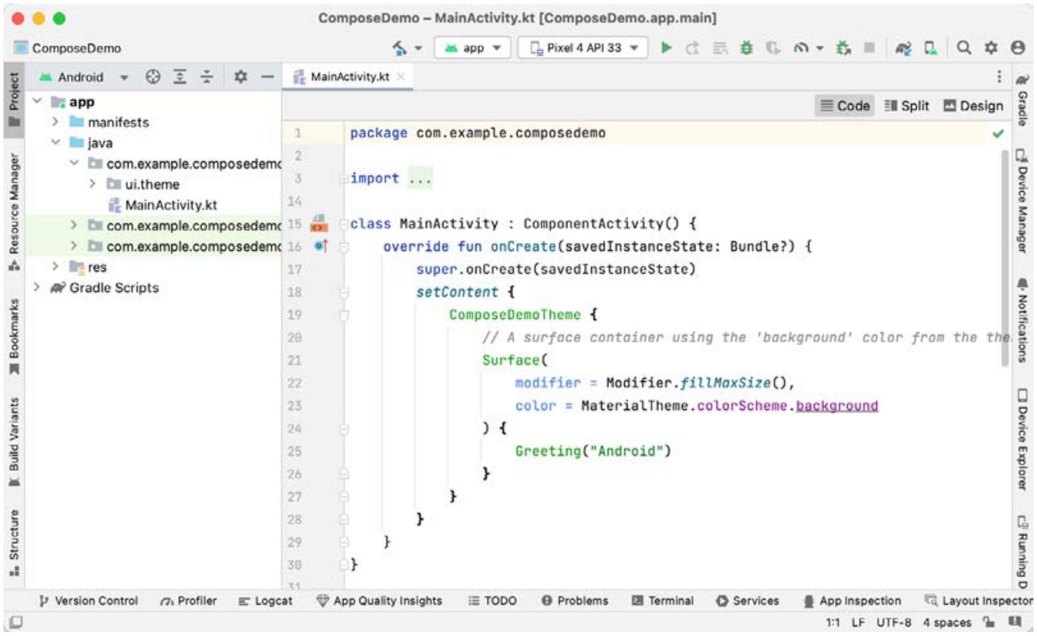
https://developer.android.com/about/versions/oreo/android-8.0

Figure 3-5

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

3.5 Enabling the New Android Studio UI

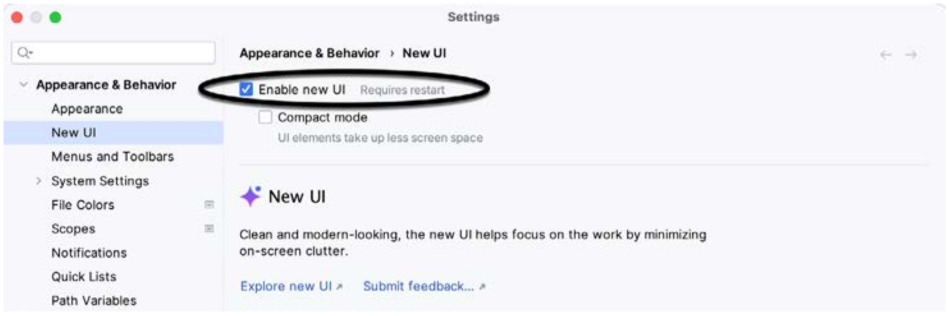
Android Studio is transitioning to a new, modern user interface that is not enabled by default in the Giraffe version. If your installation of Android Studio resembles Figure 3-6 below, then you will need to enable the new UI before proceeding:


ComposeDemo – MainActivity.kt [ComposeDemo.app.main]

```
package com.example.composedemo

import ...

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            ComposeDemoTheme {
                // A surface container using the 'background' color from the
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    Greeting("Android")
                }
            }
        }
    }
}
```

Figure 3-6

Enable the new UI by selecting the *File -> Settings...* menu option (*Android Studio -> Settings...* on macOS) and selecting the New UI option under Appearance and Behavior in the left-hand panel. From the main panel, turn on the *Enable new UI* checkbox before clicking Apply, followed by OK to commit the change:



☑ Enable new UI Requires restart
☐ Compact mode
UI elements take up less screen space

New UI

Clean and modern-looking, the new UI helps focus on the work by minimizing on-screen clutter.

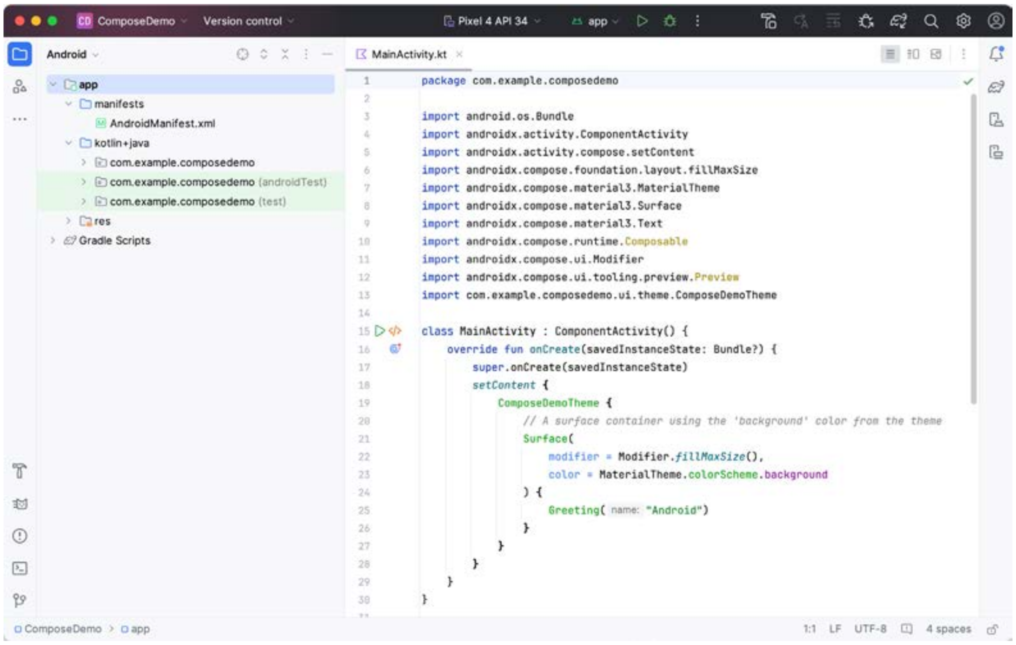
Explore new UI ↗ Submit feedback... ↗

Figure 3-7

When prompted, restart Android Studio to activate the new user interface.

3.6 Previewing the example project

Once Android Studio has restarted, the main window will reappear using the new UI and containing our AndroidSample project as illustrated in Figure 3-8 below:



```
package com.example.composedemo

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Surface
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.tooling.preview.Preview
import com.example.composedemo.ui.theme.ComposeDemoTheme

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            ComposeDemoTheme {
                // A surface container using the 'background' color from the theme
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    Greeting( name: "Android")
                }
            }
        }
    }
}
```

Figure 3-8

The newly created project and references to associated files are listed in the *Project* tool window located on the left-hand 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 3-9. If the panel is not currently in Android mode, use the menu to switch mode:

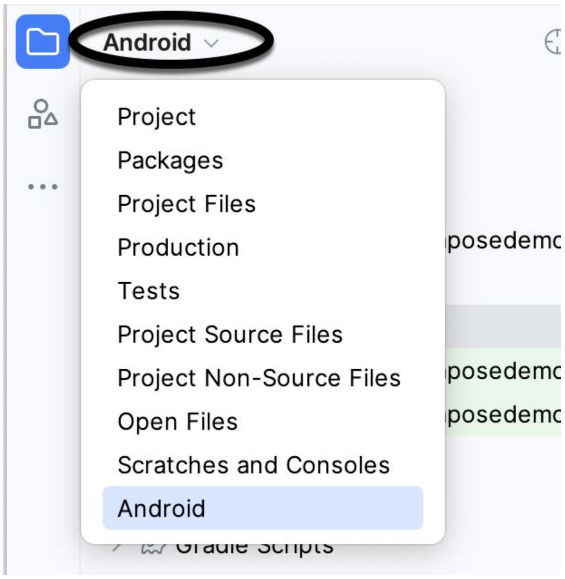


Figure 3-9

The code for the main activity of the project (an activity corresponds to a single user interface screen or module within an Android app) is contained within the *MainActivity.kt* file located under *app -> kotlin+java -> com.example.composedemo* within the Project tool window as indicated in Figure 3-10:

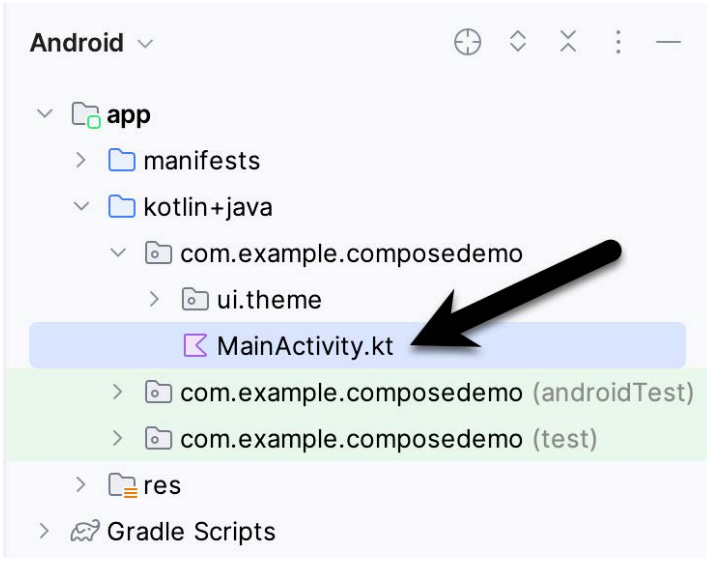


Figure 3-10

Double-click on this file to load it into the main code editor panel. The editor can be used in different view modes. Only the source code of the currently selected file is visible when the editor is in Code mode (as shown in Figure 3-8 above). Code mode is selected by clicking the button A in the figure below. However, the most helpful option when working with Compose is Split mode. To switch to Split mode, click on the button marked B:

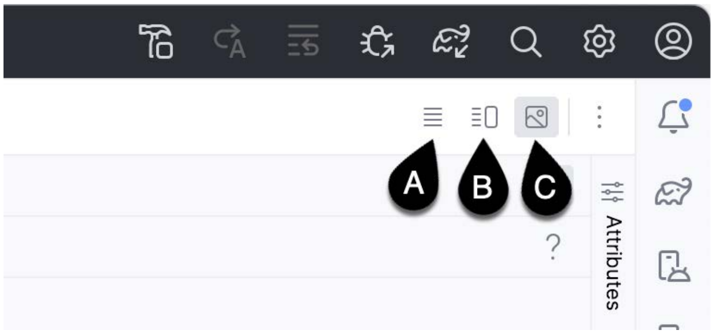
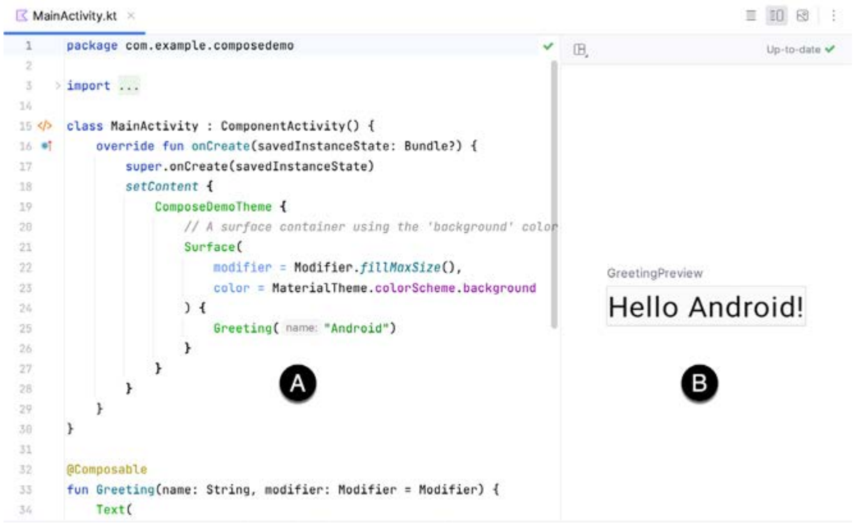


Figure 3-11

Split mode displays the code editor (A) alongside the Preview panel (B) in which the current user interface design will appear:



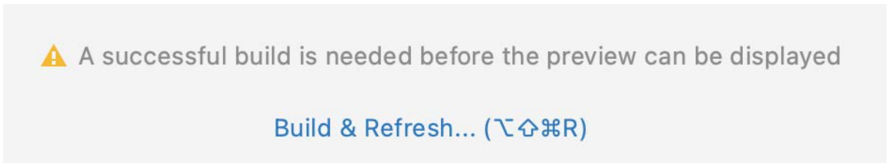
```
package com.example.composedemo

import ...

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            ComposeDemoTheme {
                // A surface container using the 'background' color
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    Greeting( name: "Android")
                }
            }
        }
    }
}

@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
```

Figure 3-12

Only the Preview panel is displayed when the editor is in Design mode (button C).

To get us started, Android Studio has already added some code to the *MainActivity.kt* file to display a Text component configured to display a message which reads “Hello Android”.

If the project has not yet been built, the Preview panel will display the message shown in Figure 3-13:



⚠ A successful build is needed before the preview can be displayed

Build & Refresh... (⌥⇧⌘R)

Figure 3-13

If you see this notification, click on the *Build & Refresh* link to rebuild the project. After the build is complete, the Preview panel should update to display the user interface defined by the code in the *MainActivity.kt* file:

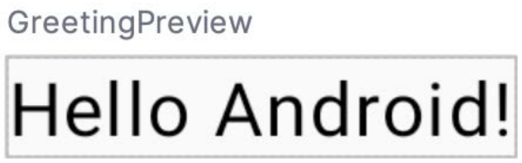


Figure 3-14

3.7 Reviewing the main activity

Android applications are created by combining one or more elements known as *Activities*. An activity is a single, standalone module of application functionality that either correlates directly to a single user interface screen and its corresponding functionality, or acts as a container for a collection of related screens. An appointments application might, for example, contain an activity screen that displays appointments set up for the current day. The application might also utilize a second activity consisting of multiple screens where new appointments may be entered by the user and existing appointments edited.

When we created the ComposeDemo project, Android Studio created a single initial activity for our app, named it MainActivity, and generated some code for it in the *MainActivity.kt* file. This activity contains the first screen that will be displayed when the app is run on a device. Before we modify the code for our requirements in the next chapter, it is worth taking some time to review the code currently contained within the *MainActivity.kt* file.

The file begins with the following line (keep in mind that this may be different if you used your own domain name instead of *com.example*):

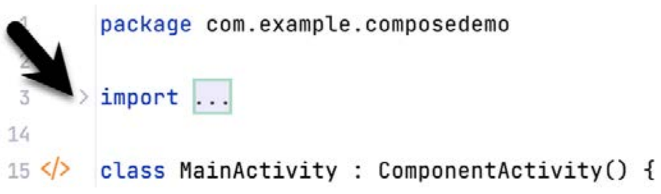
```
package com.example.composedemo
```

This tells the build system that the classes and functions declared in this file belong to the *com.example.composedemo* package which we configured when we created the project.

Next are a series of *import* directives. The Android SDK comprises a vast collection of libraries that provide the foundation for building Android apps. If all of these libraries were included within an app the resulting app bundle would be too large to run efficiently on a mobile device. To avoid this problem an app only imports the libraries that it needs to be able to run:

```
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.material3.MaterialTheme
import androidx.compose.material3.Surface
import androidx.compose.material3.Text
.
.
```

Initially, the list of import directives will most likely be "folded" to save space. To unfold the list, click on the small disclosure button indicated by the arrow in Figure 3-15 below:



```
package com.example.composedemo

import ...

class MainActivity : ComponentActivity() {
```

Figure 3-15

The MainActivity class is then declared as a subclass of the Android ComponentActivity class:

```
class MainActivity : ComponentActivity() {
.
.
}
```

The MainActivity class implements a single method in the form of *onCreate()*. This is the first method that is called when an activity is launched by the Android runtime system and is an artifact of the way apps used to be developed before the introduction of Compose. The *onCreate()* method is used here to provide a bridge between the containing activity and the Compose-based user interfaces that are to appear within it:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContent {
        ComposeDemoTheme {
.
.
        }
    }
}
```

The method declares that the content of the activity's user interface will be provided by a composable function named *ComposeDemoTheme*. This composable function is declared in the *Theme.kt* file located under the *app -> <package name> -> ui.theme* folder in the Project tool window. This, along with the other files in the *ui.theme* folder defines the colors, fonts, and shapes to be used by the activity and provides a central location from which to customize the overall theme of the app's user interface.

The call to the ComposeDemoTheme composable function is configured to contain a Surface composable. Surface is a built-in Compose component designed to provide a background for other composables:

```
ComposeDemoTheme {
    // A surface container using the 'background' color from the theme
    Surface(
        modifier = Modifier.fillMaxSize(),
        color = MaterialTheme.colorScheme.background
.
.
}
```

In this case, the Surface component is configured to fill the entire screen and with the background set to the standard background color defined by the Android Material Design theme. Material Design is a set of design guidelines developed by Google to provide a consistent look and feel across all Android apps. It includes a theme (including fonts and colors), a set of user interface components (such as button, text, and a range of text fields), icons, and generally defines how an Android app should look, behave and respond to user interactions.

Finally, the Surface is configured to contain a composable function named Greeting which is passed a string value that reads "Android":

```
ComposeDemoTheme {
    // A surface container using the 'background' color from the theme
    Surface(
        modifier = Modifier.fillMaxSize(),
        color = MaterialTheme.colorScheme.background
    ) {
        Greeting("Android")
    }
}
```

Outside of the scope of the MainActivity class, we encounter our first composable function declaration within the activity. The function is named Greeting and is, unsurprisingly, marked as being composable by the @*Composable* annotation:

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = "Hello $name!",
        modifier = modifier
    )
}
```

The function accepts a String parameter (labeled *name*) and calls the built-in Text composable, passing through a string value containing the word "Hello" concatenated with the name parameter. The function also accepts an optional modifier parameter (a topic covered in the chapter titled *"Using Modifiers in Compose"*). As will soon become evident as you work through the book, composable functions are the fundamental building blocks for developing Android apps using Compose.

The second composable function declared in the *MainActivity.kt* file reads as follows:

```
@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    ComposeDemoTheme {
        Greeting("Android")
    }
}
```

Earlier in the chapter, we looked at how the Preview panel allows us to see how the user interface will appear without having to compile and run the app. At first glance, it would be easy to assume that the preview rendering is generated by the code in the *onCreate()* method. In fact, that method only gets called when the app runs on a device or emulator. Previews are generated by preview composable functions. The *@Preview* annotation associated with the function tells Android Studio that this is a preview function and that the content emitted by the function is to be displayed in the Preview panel. As we will see later in the book, a single activity can contain multiple preview composable functions configured to preview specific sections of a user interface using different data values.

In addition, each preview may be configured by passing parameters to the *@Preview* annotation. For example, to view the preview with the rest of the standard Android screen decorations, modify the preview annotation so that it reads as follows:

```
@Preview(showSystemUi = true)
```

Once the preview has been updated, it should now be rendered as shown in Figure 3-16:

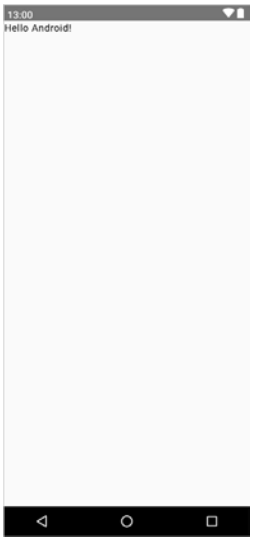


Figure 3-16

3.8 Preview updates

One final point worth noting is that the Preview panel is live and will automatically reflect minor changes made to the composable functions that make up a preview. To see this in action, edit the call to the Greeting function in the *GreetingPreview()* preview composable function to change the name from "Android" to "Compose". Note that as you make the change in the code editor, it is reflected in the preview.

More significant changes will require a build and refresh before being reflected in the preview. When this is required, Android Studio will display the following "Out of date" notice at the top of the Preview panel and a *Build & Refresh* button (indicated by the arrow in Figure 3-17):

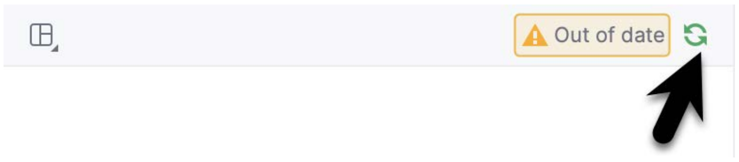


Figure 3-17

Simply click on the button to update the preview for the latest changes. Occasionally, Android Studio will fail to update the preview after code changes. If you believe that the preview no longer matches your code, hover the mouse pointer over the Up-to-date status text and select Build & Refresh from the resulting menu, as illustrated in Figure 3-18:

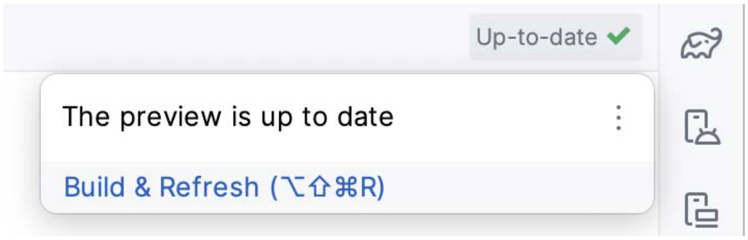


Figure 3-18

The Preview panel also includes an interactive mode that allows you to trigger events on the user interface components (for example, clicking buttons, moving sliders, scrolling through lists, etc.). Since ComposeDemo contains only an inanimate Text component at this stage, it makes more sense to introduce interactive mode in the next chapter.

3.9 Bill of Materials and the Compose version

Although Jetpack Compose and Android Studio appear to be tightly integrated, they are two separate products developed by different teams at Google. As a result, there is no guarantee that the most recent Android Studio version will default to using the latest version of Jetpack Compose. It can, therefore, be helpful to know which version of Jetpack Compose is being used by Android Studio. This is declared in a *Bill of Materials* (BOM) setting within the build configuration files of your Android Studio projects.

To identify the BOM for a project, locate the *Gradle Scripts -> libs.versions.toml* file (highlighted in the figure below) and double-click on it to load it into the editor:

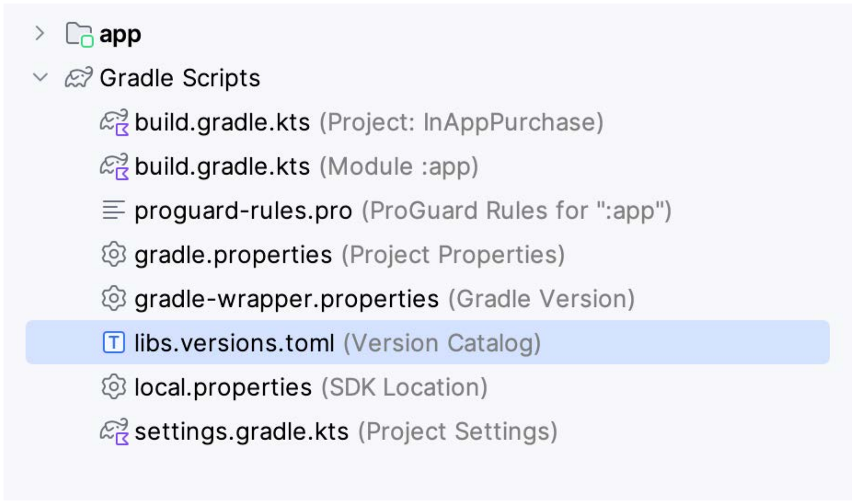

app
Gradle Scripts
build.gradle.kts (Project: InAppPurchase)
build.gradle.kts (Module :app)
proguard-rules.pro (ProGuard Rules for ":app")
gradle.properties (Project Properties)
gradle-wrapper.properties (Gradle Version)
libs.versions.toml (Version Catalog)
local.properties (SDK Location)
settings.gradle.kts (Project Settings)

Figure 3-19

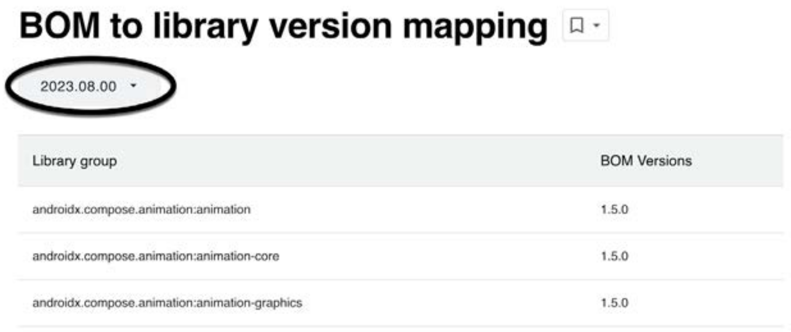
With the file loaded into the editor, locate the *composeBom* entry in the [versions] section:

```
[versions]
.
.
composeBom = "2023.08.00"
.
.
```

In the above example, we can see that the project is using BOM 2023.08.00. With this information, we can use the *BOM to library version mapping* web page at the following URL to identify the library versions being used to build our app:

https://developer.android.com/jetpack/compose/bom/bom-mapping

Once the web page has loaded, select the BOM version from the menu highlighted in Figure 3-20 below. For example, the figure shows that BOM 2023.08.00 uses version 1.5.0 of the Compose libraries:


BOM to library version mapping

2023.08.00

Library group	BOM Versions
androidx.compose.animation:animation	1.5.0
androidx.compose.animation:animation-core	1.5.0
androidx.compose.animation:animation-graphics	1.5.0

Figure 3-20

At the time of writing, Android Studio Iguana defaults to BOM 2023.08.00, while the latest stable BOM version is 2024.03.00, which maps to Jetpack Compose 1.6.4. Therefore, when working with the projects in this book, you should edit the *composeBom* entry in the *Gradle Scripts* -> *libs.versions.toml* and upgrade the BOM version to at least 2024.03.00.

Library versions and dependencies will be covered in greater detail in the *"A Guide to Gradle Version Catalogs"* chapter.

3.10 Summary

In this chapter, we have created a new project using Android Studio's *Empty Activity* template and explored some of the code automatically generated for the project. We have also introduced several features of Android Studio designed to make app development with Compose easier. The most useful features, and the places where you will spend most of your time while developing Android apps, are the code editor and Preview panel.

While the default code in the *MainActivity.kt* file provides an interesting example of a basic user interface, it bears no resemblance to the app we want to create. In the next chapter, we will modify and extend the app by removing some of the template code and writing our own composable functions.

Chapter 13

13. Kotlin Operators and Expressions

So far, we have looked at using variables and constants in Kotlin and also described the different data types. Being able to create variables is only part of the story, however. The next step is to learn how to use these variables in Kotlin code. The primary method for working with data is in the form of *expressions*.

13.1 Expression syntax in Kotlin

The most basic expression consists of an *operator*, two *operands,* and an *assignment*. The following is an example of an expression:

```
val myresult = 1 + 2
```

In the above example, the (+) operator is used to add two operands (1 and 2) together. The *assignment operator* (=) subsequently assigns the result of the addition to a variable named *myresult*. The operands could have easily been variables (or a mixture of values and variables) instead of the actual numerical values used in the example.

In the remainder of this chapter, we will look at the basic types of operators available in Kotlin.

13.2 The Basic assignment operator

We have already looked at the most basic of assignment operators, the = operator. This assignment operator simply assigns the result of an expression to a variable. In essence, the = assignment operator takes two operands. The left-hand operand is the variable to which a value is to be assigned and the right-hand operand is the value to be assigned. The right-hand operand is, more often than not, an expression that performs some type of arithmetic or logical evaluation or a call to a function, the result of which will be assigned to the variable. The following examples are all valid uses of the assignment operator:

```
var x: Int // Declare a mutable Int variable
val y = 10 // Declare and initialize an immutable Int variable

x = 10 // Assign a value to x
x = x + y // Assign the result of x + y to x
x = y // Assign the value of y to x
```

13.3 Kotlin arithmetic operators

Kotlin provides a range of operators for creating mathematical expressions. These operators primarily fall into the category of *binary operators* in that they take two operands. The exception is the *unary negative operator* (-) which serves to indicate that a value is negative rather than positive. This contrasts with the *subtraction operator* (-) which takes two operands (i.e. one value to be subtracted from another). For example:

```
var x = -10 // Unary - operator used to assign -10 to variable x
x = x - 5 // Subtraction operator. Subtracts 5 from x
```

The following table lists the primary Kotlin arithmetic operators:

Operator	**Description**
-(unary)	Negates the value of a variable or expression
*	Multiplication

/	Division
+	Addition
-	Subtraction
%	Remainder/Modulo

Table 13-1

Note that multiple operators may be used in a single expression.

For example:

```
x = y * 10 + z - 5 / 4
```

13.4 Augmented assignment operators

In an earlier section, we looked at the basic assignment operator (=). Kotlin provides several operators designed to combine an assignment with a mathematical or logical operation. These are primarily of use when performing an evaluation where the result is to be stored in one of the operands. For example, one might write an expression as follows:

```
x = x + y
```

The above expression adds the value contained in variable x to the value contained in variable y and stores the result in variable x. This can be simplified using the addition augmented assignment operator:

```
x += y
```

The above expression performs the same task as *x = x + y* but saves the programmer some typing.

Numerous augmented assignment operators are available in Kotlin. The most frequently used of which are outlined in the following table:

Operator	**Description**
x += y	Add x to y and place result in x
x -= y	Subtract y from x and place result in x
x *= y	Multiply x by y and place result in x
x /= y	Divide x by y and place result in x
x %= y	Perform Modulo on x and y and place result in x

Table 13-2

13.5 Increment and decrement operators

Another useful shortcut can be achieved using the Kotlin increment and decrement operators (also referred to as unary operators because they operate on a single operand). Consider the code fragment below:

```
x = x + 1 // Increase value of variable x by 1
x = x - 1 // Decrease value of variable x by 1
```

These expressions increment and decrement the value of x by 1. Instead of using this approach, however, it is quicker to use the ++ and -- operators. The following examples perform the same tasks as the examples above:

```
x++ // Increment x by 1
x-- // Decrement x by 1
```

These operators can be placed either before or after the variable name. If the operator is placed before the variable name, the increment or decrement operation is performed before any other operations are performed on the variable. For example, in the following code, x is incremented before it is assigned to y, leaving y with a value of 10:

```
var x = 9
val y = ++x
```

In the next example, however, the value of x (9) is assigned to variable y before the decrement is performed. After the expression is evaluated the value of y will be 9 and the value of x will be 8.

```
var x = 9
val y = x--
```

13.6 Equality operators

Kotlin also includes a set of logical operators useful for performing comparisons. These operators all return a Boolean result depending on the result of the comparison. These operators are *binary operators* in that they work with two operands.

Equality operators are most frequently used in constructing program control flow logic. For example, an *if* statement may be constructed based on whether one value matches another:

```
if (x == y) {
      // Perform task
}
```

The result of a comparison may also be stored in a Boolean variable. For example, the following code will result in a *true* value being stored in the variable result:

```
var result: Boolean
val x = 10
val y = 20

result = x < y
```

Clearly 10 is less than 20, resulting in a *true* evaluation of the *x < y* expression. The following table lists the full set of Kotlin comparison operators:

Operator	**Description**
x == y	Returns true if x is equal to y
x > y	Returns true if x is greater than y
x >= y	Returns true if x is greater than or equal to y
x < y	Returns true if x is less than y
x <= y	Returns true if x is less than or equal to y
x != y	Returns true if x is not equal to y

Table 13-3

13.7 Boolean logical operators

Kotlin also provides a set of so-called logical operators designed to return Boolean *true* or *false* values. These operators both return Boolean results and take Boolean values as operands. The key operators are NOT (!), AND (&&), and OR (||).

The NOT (!) operator simply inverts the current value of a Boolean variable or the result of an expression. For example, if a variable named *flag* is currently true, prefixing the variable with a '!' character will invert the value to false:

```
val flag = true // variable is true
val secondFlag = !flag // secondFlag set to false
```

The OR (||) operator returns true if one of its two operands evaluates to true, otherwise, it returns false. For example, the following code evaluates to true because at least one of the expressions on either side of the OR operator is true:

```
if ((10 < 20) || (20 < 10)) {
        print("Expression is true")
}
```

The AND (&&) operator returns true only if both operands are evaluated to be true. The following example will return false because only one of the two operand expressions evaluates to true:

```
if ((10 < 20) && (20 < 10)) {
      print("Expression is true")
}
```

13.8 Range operator

Kotlin includes a useful operator that allows a range of values to be declared. As will be seen in later chapters, this operator is invaluable when working with looping in program logic.

The syntax for the range operator is as follows:

```
x..y
```

This operator represents the range of numbers starting at x and ending at y where both x and y are included within the range (referred to as a closed range). The range operator 5..8, for example, specifies the numbers 5, 6, 7, and 8.

13.9 Bitwise operators

As previously discussed, computer processors work in binary. These are essentially streams of ones and zeros, each one referred to as a bit. Bits are formed into groups of 8 to form bytes. As such, it is not surprising that we, as programmers, will occasionally end up working at this level in our code. To facilitate this requirement, Kotlin provides a range of *bit operators*.

Those familiar with bitwise operators in other languages such as C, C++, C#, Objective-C, and Java will find nothing new in this area of the Kotlin language syntax. For those unfamiliar with binary numbers, now may be a good time to seek out reference materials on the subject to understand how ones and zeros are formed into bytes to form numbers. Other authors have done a much better job of describing the subject than we can do within the scope of this book.

For this exercise, we will be working with the binary representation of two numbers. First, the decimal number 171 is represented in binary as:

```
10101011
```

Second, the number 3 is represented by the following binary sequence:

```
00000011
```

Now that we have two binary numbers with which to work, we can begin to look at the Kotlin bitwise operators:

13.9.1 Bitwise inversion

The Bitwise inversion (also referred to as NOT) is performed using the *inv()* operation and has the effect of inverting all of the bits in a number. In other words, all the zeros become ones and all the ones become zeros. Taking our example 3 number, a Bitwise NOT operation has the following result:

```
00000011 NOT
```

```
========
11111100
```

The following Kotlin code, therefore, results in a value of -4:

```
val y = 3
val z = y.inv()

print("Result is $z")
```

13.9.2 Bitwise AND

The Bitwise AND is performed using the *and()* operation. It makes a bit-by-bit comparison of two numbers. Any corresponding position in the binary sequence of each number where both bits are 1 results in a 1 appearing in the same position of the resulting number. If either bit position contains a 0 then a zero appears in the result. Taking our two example numbers, this would appear as follows:

```
10101011 AND
00000011
========
00000011
```

As we can see, the only locations where both numbers have 1s are the last two positions. If we perform this in Kotlin code, therefore, we should find that the result is 3 (00000011):

```
val x = 171
val y = 3
val z = x.and(y)

print("Result is $z")
```

13.9.3 Bitwise OR

The bitwise OR also performs a bit-by-bit comparison of two binary sequences. Unlike the AND operation, the OR places a 1 in the result if there is a 1 in the first or second operand. Using our example numbers, the result will be as follows:

```
10101011 OR
00000011
========
10101011
```

If we perform this operation in Kotlin using the *or()* operation the result will be 171:

```
val x = 171
val y = 3
val z = x.or(y)

print("Result is $z")
```

13.9.4 Bitwise XOR

The bitwise XOR (commonly referred to as *exclusive OR* and performed using the *xor()* operation) performs a similar task to the OR operation except that a 1 is placed in the result if one or other corresponding bit positions in the two numbers is 1. If both positions are a 1 or a 0 then the corresponding bit in the result is set to a 0. For example:

```
10101011 XOR
```

```
00000011
========
10101000
```

The result, in this case, is 10101000 which converts to 168 in decimal. To verify this we can, once again, try some Kotlin code:

```
val x = 171
val y = 3
val z = x.xor(y)

print("Result is $z")
```

When executed, we get the following output from print:

```
Result is 168
```

13.9.5 Bitwise left shift

The bitwise left shift moves each bit in a binary number a specified number of positions to the left. Shifting an integer one position to the left has the effect of doubling the value.

As the bits are shifted to the left, zeros are placed in the vacated rightmost (low-order) positions. Note also that once the leftmost (high-order) bits are shifted beyond the size of the variable containing the value, those high -order bits are discarded:

```
10101011 Left Shift one bit
========
101010110
```

In Kotlin the bitwise left shift operator is performed using the *shl()* operation, passing through the number of bit positions to be shifted. For example, to shift left by 1 bit:

```
val x = 171
val z = x.shl(1)

print("Result is $z")
```

When compiled and executed, the above code will display a message stating that the result is 342 which, when converted to binary, equates to 101010110.

13.9.6 Bitwise right shift

A bitwise right shift is, as you might expect, the same as a left except that the shift takes place in the opposite direction. Shifting an integer one position to the right has the effect of halving the value.

Note that since we are shifting to the right, there is no opportunity to retain the lowermost bits regardless of the data type used to contain the result. As a result, the low-order bits are discarded. Whether or not the vacated high-order bit positions are replaced with zeros or ones depends on whether the *sign bit* used to indicate positive and negative numbers is set or not.

```
10101011 Right Shift one bit
========
01010101
```

The bitwise right shift is performed using the *shr()* operation passing through the shift count:

```
val x = 171
val z = x.shr(1)

print("Result is $z")
```

When executed, the above code will report the result of the shift as being 85, which equates to binary 01010101.

13.10 Summary

Operators and expressions provide the underlying mechanism by which variables and constants are manipulated and evaluated within Kotlin code. This can take the simplest of forms whereby two numbers are added using the addition operator in an expression and the result stored in a variable using the assignment operator. Operators fall into a range of categories, details of which have been covered in this chapter.

Chapter 29

29. An Introduction to FlowRow and FlowColumn

The chapter entitled *"Composing Layouts with Row and Column"* used the Row and Column composables to present content elements uniformly within a user interface. One limitation of Row and Column-based layouts is that they are not well suited to organizing dynamic elements in terms of the quantity and sizes of the content. These composables are also less effective when designing layouts that are responsive to device screen orientation and size changes.

In this chapter, we will learn about the Flow layout composables and explore how they provide a more flexible way to organize content in rows and columns.

29.1 FlowColumn and FlowRow

The Row and Column composables work best when you know the number of items to be displayed and their respective sizes. This results in a spreadsheet-like layout with rows of aligned columns. The Flow layouts, however, are designed to flow content onto the next row or column when space runs out. These composables also discard the spreadsheet approach to organization, providing a more flexible approach to displaying items of varying sizes. Figure 29-1, for example, shows a typical FlowRow layout:

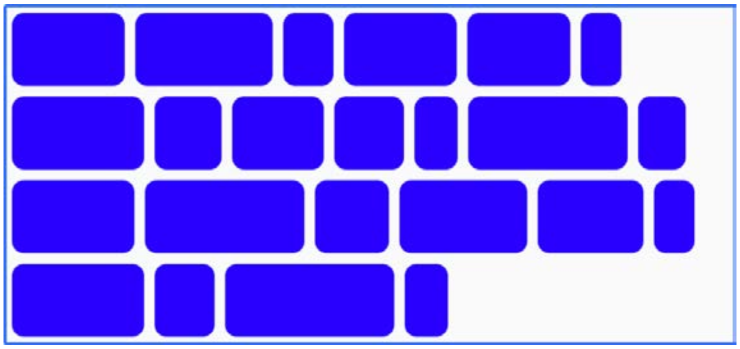


Figure 29-1

As we will explore later in this chapter, Flow layouts provide extensive options for configuring the layout and arrangement of child items, including weight, spacing, alignment, and the maximum number of items per row or column.

The FlowRow composable uses the following syntax:

```
FlowRow(
    modifier: Modifier = Modifier,
    horizontalArrangement: Arrangement.Horizontal,
    verticalArrangement: Arrangement.Vertical,
    maxItemsInEachRow: Int
) {
   // Content here
}
```

Figure 29-2 shows an example FlowColumn layout:

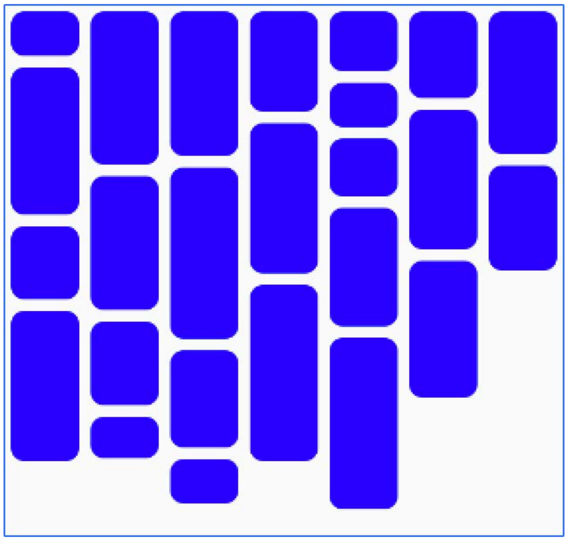


Figure 29-2

The FlowColumn composable uses the following syntax:

```
FlowColumn(
    modifier: Modifier,
    verticalArrangement: Arrangement.Vertical,
    horizontalArrangement: Arrangement.Horizontal,
    maxItemsInEachColumn: Int,
) {
    // Content here
}
```

29.2 Maximum number of items

Without restrictions, the Flow layouts will fit as many items into a row or column as possible before flowing to the next one. The maximum number of items can be restricted using the *maxItemsInEachColumn* and *maxItemsInEachRow* properties of the FlowColumn and FlowRow. For example:

```
FlowRow(maxItemsInEachRow = 10) {
    // Flow items here
}

FlowColumn(maxItemsInEachColumn = 5) {
    // Flow items here
}
```

29.3 Working with main axis arrangement

Main axis arrangement defines how the flow items are positioned along the main axis of the parent Flow layout. For example, the *horizontalArrangement* property controls the arrangement of flow items along the horizontal axis of the FlowRow composable. Table 29-1 shows the effects of the various horizontalArrangement options when applied to a FlowRow instance:

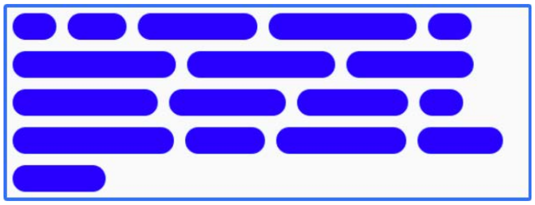
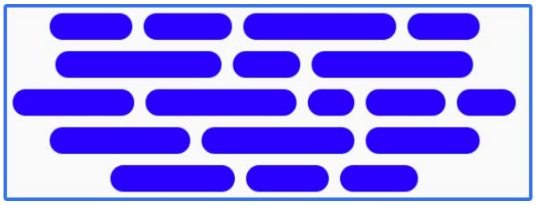
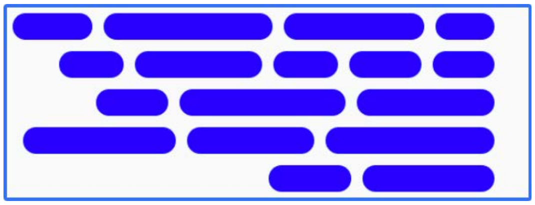
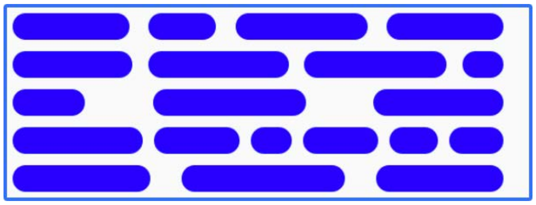
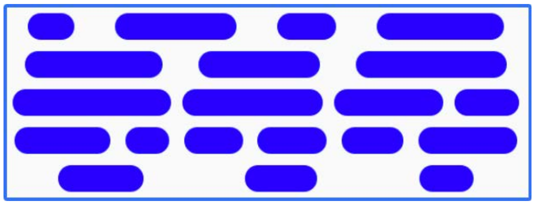
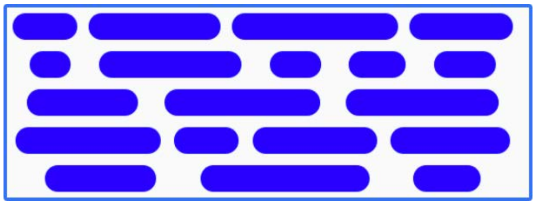
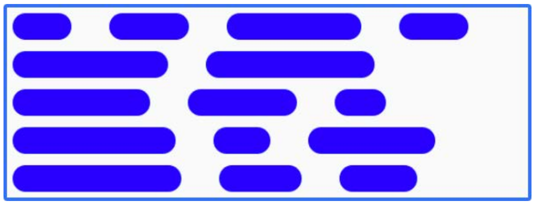
	
Arrangement.Start	Arrangement.Center
	
Arrangement.End	Arrangement.SpaceBetween
	
Arrangement.SpaceAround	Arrangement.SpaceEvenly
	
Arrangement.spacedBy(10.dp)	

Table 29-1

Similarly, the *verticalArrangement* property controls the positioning of flow items along the vertical access of the FlowColumn. The same arrangement options are available as those listed above, except that *Arrangement.Start* and *Arrangement.End* are replaced by *Arrangement.Top* and *Arrangement.Bottom*.

29.4 Understanding cross-axis arrangement

Cross-axis arrangement controls the arrangement of a flow layout on the opposite axis to the main flow. In other words, the *verticalArrangement* property controls the vertical positioning of FlowRow items, while *horizontalArrangement* does the same along the horizontal axis of FlowColumn items. Table 29-2 demonstrates the three *horizontalArrangement* options applied to a FlowColumn instance:

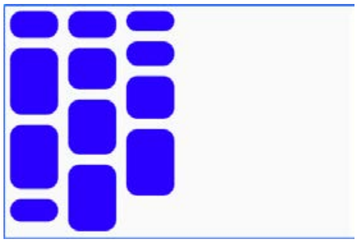
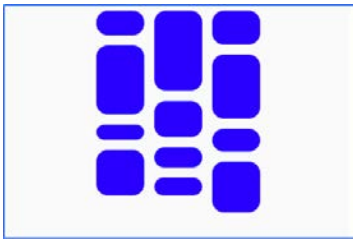
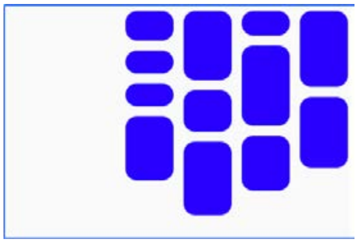
		
Arrangement.Start	Arrangement.Center	Arrangement.End

Table 29-2

29.5 Item alignment

The alignment of items within individual rows or columns can be controlled by passing an alignment value to the *align()* modifier of the child items of a Flow layout. This is useful when the Flow items vary in height (FlowRow) or width (FlowColumn). The following code, for example, specifies bottom alignment for a FlowRow item:

```
FlowRow {
    repeat(6) {
        MyFlowItem(modifier = Modifier.align(Alignment.Bottom))
    }
}
```

The following table illustrates the effect of applying *Alignment.Top*, *Alignment.CenterVertically*, and *Alignment.Bottom* to FlowRow items of varying height:

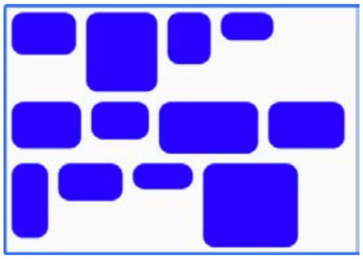
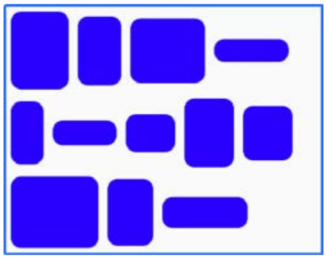
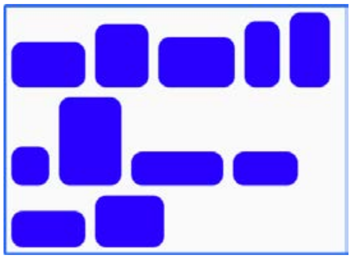
		
Alignment.Top	Alignment.CenterVertically	Alignment.Bottom

Table 29-3

Equivalent alignment effects can be achieved for FlowColumn items using *Alignment.Start*, *Alignment.CenterHorizontally*, and *Alignment.End*

29.6 Controlling item size

Weight factors can be applied to individual Flow items to specify the size relative to the overall space available and the weights of other items in the same row or column. Weights are expressed as Float values and applied to individual Flow items using the *weight()* modifier. Consider, for example, a FlowRow containing a single item with a weight of 1f:

```
FlowRow {
    MyFlowItem(
        Modifier
            .weight(1f)
            )
}
```

When the layout is rendered, the item will occupy all the available space because it is the only item in the row:

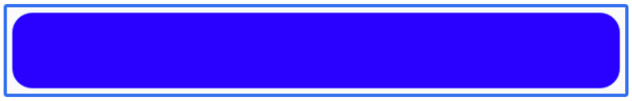


Figure 29-3

If we add a second item, also with a weight of 1f, the two items will share the row equally:

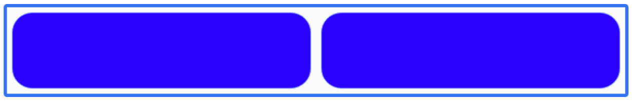


Figure 29-4

If we add a third item with a weight of 1f, each item would occupy a third of the space. However, suppose that the third item has a weight of 2f, giving us a weight combination of 1f, 1f, and 2f. In this case, the first two items occupy half of the available space, while the third occupies the other half:

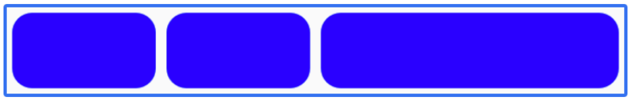


Figure 29-5

To calculate an item's when using weights, the Flow composables divide the amount of space remaining in the row or column by the total item weights, multiplied by the weight of the current item.

Another way to control the size of the items in a Flow layout is to use fractional sizing. Fractional sizing involves specifying the percentage of the overall space in a row or column that an item is to occupy. The fraction is declared as a Float value and applied to FlowRow and FlowColumn items using the *fillMaxWidth()* and *fillMaxHeight()* modifiers, respectively. For example:

```
FlowRow {
     MyFlowItem(Modifier.width(50.dp))
     MyFlowItem(Modifier.fillMaxWidth(0.7f))
     MyFlowItem(Modifier.width(50.dp))
}
```

Regardless of the sizes of the other items, the fractional item in the above code example will always occupy 70% of the row:

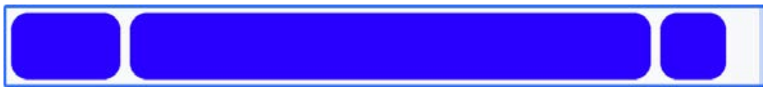


Figure 29-6

If there is insufficient room for the fractional item, items will flow onto the next row to make room:

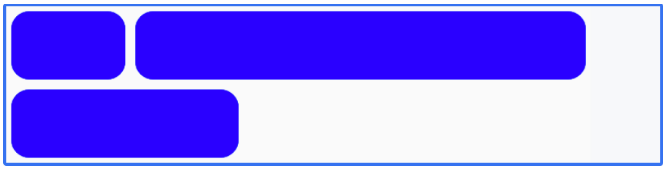


Figure 29-7

29.7 Summary

The FlowRow and FlowColumn composables are ideal for arranging groups of items of varying sizes and quantities into flexible rows and columns. When a Flow layout runs out of space to display items, the remaining content flows to the next row or column. Combined with an extensive collection of alignment, spacing, and arrangement options, these composables provide a flexible and easy layout solution for presenting content within apps.

Chapter 46

46. Working with ViewModels in Compose

Until a few years ago, Google did not recommend a specific approach to building Android apps other than to provide tools and development kits while letting developers decide what worked best for a particular project or individual programming style. That changed in 2017 with the introduction of the Android Architecture Components which became part of Android Jetpack when it was released in 2018. Jetpack has of course, since been expanded with the addition of Compose.

This chapter will provide an overview of the concepts of Jetpack, Android app architecture recommendations, and the ViewModel component.

46.1 What is Android Jetpack?

Android Jetpack consists of Android Studio, the Android Architecture Components, Android Support Library, and the Compose framework together with a set of guidelines that recommend how an Android App should be structured. The Android Architecture Components were designed to make it quicker and easier both to perform common tasks when developing Android apps while also conforming to the key principle of the architectural guidelines. While many of these components have been superseded by features built into Compose, the ViewModel architecture component remains relevant today. Before exploring the ViewModel component, it first helps to understand both the old and new approaches to Android app architecture.

46.2 The "old" architecture

In the chapter entitled *"An Example Compose Project"*, an Android project was created consisting of a single activity that contained all of the code for presenting and managing the user interface together with the back-end logic of the app. Up until the introduction of Jetpack, the most common architecture followed this paradigm with apps consisting of multiple activities (one for each screen within the app) with each activity class to some degree mixing user interface and back-end code.

This approach led to a range of problems related to the lifecycle of an app (for example an activity is destroyed and recreated each time the user rotates the device leading to the loss of any app data that had not been saved to some form of persistent storage) as well as issues such as inefficient navigation involving launching a new activity for each app screen accessed by the user.

46.3 Modern Android architecture

At the most basic level, Google now advocates single activity apps where different screens are loaded as content within the same activity.

Modern architecture guidelines also recommend separating different areas of responsibility within an app into entirely separate modules (a concept called "separation of concerns"). One of the keys to this approach is the ViewModel component.

46.4 The ViewModel component

The purpose of ViewModel is to separate the user interface-related data model and logic of an app from the code responsible for displaying and managing the user interface and interacting with the operating system.

When designed in this way, an app will consist of one or more *UI Controllers,* such as an activity, together with ViewModel instances responsible for handling the data needed by those controllers.

A ViewModel is implemented as a separate class and contains *state* values containing the model data and functions that can be called to manage that data. The activity containing the user interface *observes* the model state values such that any value changes trigger a recomposition. User interface events relating to the model data such as a button click are configured to call the appropriate function within the ViewModel. This is, in fact, a direct implementation of the *unidirectional data flow* concept described in the chapter entitled *"An Overview of Compose State and Recomposition"*. The diagram in Figure 46-1 illustrates this concept as it relates to activities and ViewModels:

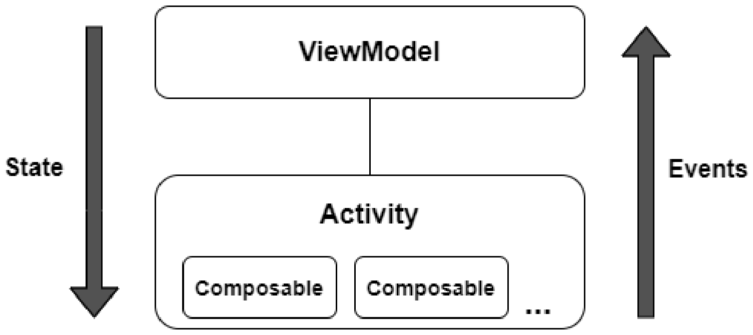


Figure 46-1

This separation of responsibility addresses the issues relating to the lifecycle of activities. Regardless of how many times an activity is recreated during the lifecycle of an app, the ViewModel instances remain in memory thereby maintaining data consistency. A ViewModel used by an activity, for example, will remain in memory until the activity finishes which, in the single activity app, is not until the app exits.

In addition to using ViewModels, the code responsible for gathering data from data sources such as web services or databases should be built into a separate *repository* module instead of being bundled with the view model. This topic will be covered in detail beginning with the chapter entitled *"Room Databases and Compose"*.

46.5 ViewModel implementation using state

The main purpose of a ViewModel is to store data that can be observed by the user interface of an activity. This allows the user interface to react when changes occur to the ViewModel data. There are two ways to declare the data within a ViewModel so that it is observable. One option is to use the Compose state mechanism which has been used extensively throughout this book. An alternative approach is to use the Jetpack LiveData component, a topic that will be covered later in this chapter.

Much like the state declared within composables, ViewModel state is declared using the *mutableStateOf* group of functions. The following ViewModel declaration, for example, declares a state containing an integer count value with an initial value of 0:

```
class MyViewModel : ViewModel() {

    var customerCount by mutableStateOf(0)

}
```

With some data encapsulated in the model, the next step is to add a function that can be called from within the UI to change the counter value:

```
class MyViewModel : ViewModel() {

    var customerCount by mutableStateOf(0)

    fun increaseCount() {
        customerCount++
    }
}
```

Even complex models are nothing more than a continuation of these two basic state and function building blocks.

46.6 Connecting a ViewModel state to an activity

A ViewModel is of little use unless it can be used within the composables that make up the app user interface. All this requires is to pass an instance of the ViewModel as a parameter to a composable from which the state values and functions can be accessed. Programming convention recommends that these steps be performed in a composable dedicated solely for this task and located at the top of the screen's composable hierarchy. The model state and event handler functions can then be passed to child composables as necessary. The following code shows an example of how a ViewModel might be accessed from within an activity:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            ViewModelWorkTheme {
                Surface(color = MaterialTheme.colorScheme.background) {
                    TopLevel()
                }
            }
        }
    }
}

@Composable
fun TopLevel(model: MyViewModel = viewModel()) {
    MainScreen(model.customerCount) { model.increaseCount() }
}

@Composable
fun MainScreen(count: Int, addCount: () -> Unit = {}) {
    Column(horizontalAlignment = Alignment.CenterHorizontally,
        modifier = Modifier.fillMaxWidth()) {
        Text("Total customers = $count",
        Modifier.padding(10.dp))
        Button(
            onClick = addCount,
        ) {
            Text(text = "Add a Customer")
        }
    }
}
```

In the above example, the first function call is made by the *onCreate()* method to the TopLevel composable which is declared with a default ViewModel parameter initialized via a call to the *viewModel()* function:

```
@Composable
fun TopLevel(model: MyViewModel = viewModel()) {
.
.
```

The *viewModel()* function is provided by the Compose view model lifecycle library which needs to be added to the project's build dependencies when working with view models. This requires the following additions to the *Gradle Scripts -> libs.version.tomi* file:

```
[versions]
activityCompose = "1.8.2"
.
.
[libraries]
androidx-lifecycle-viewmodel-compose = { module = "androidx.lifecycle:lifecycle-
viewmodel-compose", version.ref = "lifecycleRuntimeKtx" }
.
.
```

Once the library has been added to the version catalog, it must be added to the *dependencies* section of the *Gradle Scripts -> build.gradle.kts (Module :app)* file:

```
dependencies {
.
.
    implementation(libs.androidx.lifecycle.viewmodel.compose)
.
.
```

If an instance of the view model has already been created within the current scope, the *viewModel()* function will return a reference to that instance. Otherwise, a new view model instance will be created and returned.

With access to the ViewModel instance, the TopLevel function is then able to obtain references to the view model *customerCount* state variable and *increaseCount()* function which it passes to the MainScreen composable:

```
MainScreen(model.customerCount) { model.increaseCount() }
```

As implemented, Button clicks will result in calls to the view model *increaseCount()* function which, in turn, increments the *customerCount* state. This change in state triggers a recomposition of the user interface, resulting in the new customer count value appearing in the Text composable.

The use of state and view models will be demonstrated in the chapter entitled *"A Compose ViewModel Tutorial"*.

46.7 ViewModel implementation using LiveData

The Jetpack LiveData component predates the introduction of Compose and can be used as a wrapper around data values within a view model. Once contained in a LiveData instance, those variables become observable to composables within an activity. LiveData instances can be declared as being mutable using the MutableLiveData class, allowing the ViewModel functions to make changes to the underlying data value. An example view model designed to store a customer name could, for example, be implemented as follows using MutableLiveData instead of state:

```
class MyViewModel : ViewModel() {

    var customerName: MutableLiveData<String> = MutableLiveData("")

    fun setName(name: String) {
        customerName.value = name
    }
}
```

Note that new values must be assigned to the live data variable via the *value* property.

46.8 Observing ViewModel LiveData within an activity

As with state, the first step when working with LiveData is to obtain an instance of the view model within an initialization composable:

```
@Composable
fun TopLevel(model: MyViewModel = viewModel()) {

}
```

Once we have access to a view model instance, the next step is to make the live data observable. This is achieved by calling the *observeAsState()* method on the live data object:

```
@Composable
fun TopLevel(model: MyViewModel = viewModel()) {
    var customerName: String by model.customerName.observeAsState("")
}
```

In the above code, the *observeAsState()* call converts the live data value into a state instance and assigns it to the customerName variable. Once converted, the state will behave in the same way as any other state object, including triggering recompositions whenever the underlying value changes.

The use of LiveData and view models will be demonstrated in the chapter entitled *"A Compose Room Database and Repository Tutorial".*

46.9 Summary

Until recently, Google has tended not to recommend any particular approach to structuring an Android app. That changed with the introduction of Android Jetpack which consists of a set of tools, components, libraries, and architecture guidelines. These architectural guidelines recommend that an app project be divided into separate modules, each being responsible for a particular area of functionality, otherwise known as "separation of concerns". In particular, the guidelines recommend separating the view data model of an app from the code responsible for handling the user interface. This is achieved using the ViewModel component. In this chapter, we have covered ViewModel-based architecture and demonstrated how this is implemented when developing with Compose. We have also explored how to observe and access view model data from within an activity using both state and LiveData.

Chapter 61

61. An Overview of Android In-App Billing

In the early days of mobile applications for operating systems such as Android and iOS, the most common method for earning revenue was to charge an upfront fee to download and install the application. However, Google soon introduced another revenue opportunity by embedding advertising within applications. Perhaps the most common and lucrative option is now to charge the user for purchasing items from within the application after it has been installed. This typically takes the form of access to a higher level in a game, acquiring virtual goods or currency, or subscribing to premium content in the digital edition of a magazine or newspaper.

Google supports integrating in-app purchasing through the Google Play In-App Billing API and the Play Console. This chapter will provide an overview of in-app billing and outline how to integrate in-app billing into your Android projects. Once these topics have been explored, the next chapter will walk you through creating an example app that includes in-app purchasing features.

61.1 Preparing a project for In-App purchasing

Building in-app purchasing into an app will require a Google Play Developer Console account, which was covered previously in the *"Creating, Testing and Uploading an Android App Bundle"* chapter. In addition, you must also register a Google merchant account and configure your payment settings. You can find these settings by navigating to *Setup -> Payments profile* in the Play Console. Note that merchant registration is not available in all countries. For details, refer to the following page:

https://support.google.com/googleplay/android-developer/answer/9306917

The app will then need to be uploaded to the console and enabled for in-app purchasing. The console will not activate in-app purchasing support for an app, however, unless the Google Play Billing Library has been added to the module-level *build.gradle.kts* file. When working with Kotlin, the Google Play Kotlin Extensions Library is also recommended:

```
dependencies {
.
.
    implementation(libs.billing)
    implementation(libs.billing.ktx)
.
.
}
```

The corresponding entries in the *libs.versions.toml* file for the above libraries will read as follows:

```
[versions]
billing = "<latest version>"
.
.
[libraries]
```

```
billing = { module = "com.android.billingclient:billing", version.ref = "billing"
}
billing-ktx = { module = "com.android.billingclient:billing-kStx", version.ref =
"billing" }
.
.
```

Once the build files have been modified and the app bundle uploaded to the console, the next step is to add in-app products or subscriptions for the user to purchase.

61.2 Creating In-App products and subscriptions

Products and subscriptions are created and managed using the options listed beneath the Monetize section of the Play Console navigation panel as highlighted in Figure 61-1 below:

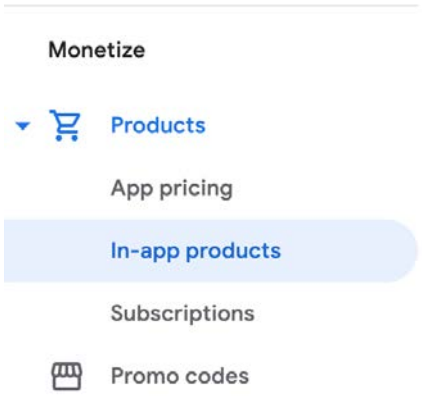


Figure 61-1

Each product or subscription needs an ID, title, description, and pricing information. Purchases fall into the categories of *consumable* (the item must be purchased each time it is required by the user such as virtual currency in a game), *non-consumable* (only needs to be purchased once by the user such as content access), and *subscription*-based. Consumable and non-consumable products are collectively referred to as *managed products*.

Subscriptions are useful for selling an item that needs to be renewed on a regular schedule such as access to news content or the premium features of an app. When creating a subscription, a *base plan* is defined specifying the price, renewal period (monthly, annually, etc.), and whether the subscription auto-renews. Users can also be provided with discount *offers* and given the option of pre-purchasing a subscription.

61.3 Billing client initialization

A BillingClient instance handles communication between your app and the Google Play Billing Library. In addition, BillingClient includes a set of methods that can be called to perform both synchronous and asynchronous billing-related activities. When the billing client is initialized, it will need to be provided with a reference to a PurchasesUpdatedListener callback handler. The client will call this handler to notify your app of the results of any purchasing activity. To avoid duplicate notifications, it is recommended to have only one BillingClient instance per app.

A BillingClient instance can be created using the *newBuilder()* method, passing through the current activity or fragment context. The purchase update handler is then assigned to the client via the *setListener()* method:

```
private val purchasesUpdatedListener =
    PurchasesUpdatedListener { billingResult, purchases ->
```

```
        if (billingResult.responseCode ==
            BillingClient.BillingResponseCode.OK
            && purchases != null
        ) {
            for (purchase in purchases) {
                // Process the purchases
            }
        } else if (billingResult.responseCode ==
            BillingClient.BillingResponseCode.USER_CANCELED
        ) {
            // Purchase cancelled by user
        } else {
            // Handle errors here
        }
    }

billingClient = BillingClient.newBuilder(this)
    .setListener(purchasesUpdatedListener)
    .enablePendingPurchases()
    .build()
```

61.4 Connecting to the Google Play Billing library

After successfully creating the Billing Client, the next step is initializing a connection to the Google Play Billing Library. To establish this connection, a call needs to be made to the *startConnection()* method of the billing client instance. Since the connection is performed asynchronously, a BillingClientStateListener handler needs to be implemented to receive a callback indicating whether the connection was successful. Code should also be added to override the *onBillingServiceDisconnected()* method. This is called if the connection to the Billing Library is lost and can be used to report the problem to the user and retry the connection.

Once the setup and connection tasks are complete, the BillingClient instance will make a call to the *onBillingSetupFinished()* method which can be used to check that the client is ready:

```
billingClient.startConnection(object : BillingClientStateListener {
    override fun onBillingSetupFinished(
        billingResult: BillingResult
    ) {
        if (billingResult.responseCode ==
            BillingClient.BillingResponseCode.OK
        ) {
            // Connection successful
        } else {
            // Connection failed
        }
    }

    override fun onBillingServiceDisconnected() {
        // Connection to billing service lost
    }
})
```

61.5 Querying available products

Once the billing environment is initialized and ready to go, the next step is to request the details of the products or subscriptions available for purchase. This is achieved by making a call to the *queryProductDetailsAsync()* method of the BillingClient and passing through an appropriately configured QueryProductDetailsParams instance containing the product ID and type (ProductType.SUBS for a subscription or ProductType.INAPP for a managed product):

```
val queryProductDetailsParams = QueryProductDetailsParams.newBuilder()
    .setProductList(
        ImmutableList.of(
            QueryProductDetailsParams.Product.newBuilder()
                .setProductId(productId)
                .setProductType(
                    BillingClient.ProductType.INAPP
                )
                .build()
        )
    )
    .build()

billingClient.queryProductDetailsAsync(
    queryProductDetailsParams
) { billingResult, productDetailsList ->
    if (!productDetailsList.isEmpty()) {
        // Process list of matching products
    } else {
        // No product matches found
    }
}
```

The *queryProductDetailsAsync()* method is passed a ProductDetailsResponseListener handler (in this case in the form of a lambda code block) which, in turn, is called and passed a list of ProductDetail objects containing information about the matching products. For example, we can call methods on these objects to get information such as the product name, title, description, price, and offer details.

61.6 Starting the purchase process

Once a product or subscription has been queried and selected for purchase by the user, the purchase process is ready to be launched. We do this by calling the *launchBillingFlow()* method of the BillingClient, passing through as arguments the current activity and a BillingFlowParams instance configured with the ProductDetail object for the item being purchased.

```
val billingFlowParams = BillingFlowParams.newBuilder()
    .setProductDetailsParamsList(
        ImmutableList.of(
            BillingFlowParams.ProductDetailsParams.newBuilder()
                .setProductDetails(productDetails)
                .build()
        )
    )
    .build()

billingClient.launchBillingFlow(this, billingFlowParams)
```

The success or otherwise of the purchase operation will be reported via a call to the PurchasesUpdatedListener callback handler outlined earlier in the chapter.

61.7 Completing the purchase

When purchases are successful, the PurchasesUpdatedListener handler will be passed a list containing a Purchase object for each item. You can verify that the item has been purchased by calling the *getPurchaseState()* method of the Purchase instance as follows:

```
if (purchase.getPurchaseState() == Purchase.PurchaseState.PURCHASED) {
    // Purchase completed.
} else if (purchase.getPurchaseState() == Purchase.PurchaseState.PENDING) {
    // Payment is still pending
}
```

Note that your app will only support pending purchases if a call is made to the *enablePendingPurchases()* method during initialization. A pending purchase will remain so until the user completes the payment process.

When the purchase of a non-consumable item is complete, it will need to be acknowledged to prevent a refund from being issued to the user. This requires the *purchase token* for the item which is obtained via a call to the *getPurchaseToken()* method of the Purchase object. This token is used to create an AcknowledgePurchaseParams instance together with an AcknowledgePurchaseResponseListener handler. Managed product purchases and subscriptions are acknowledged by calling the BillingClient's *acknowledgePurchase()* method as follows:

```
billingClient.acknowledgePurchase(acknowledgePurchaseParams,
                             acknowledgePurchaseResponseListener);
val acknowledgePurchaseParams = AcknowledgePurchaseParams.newBuilder()
    .setPurchaseToken(purchase.purchaseToken)
    .build()

val acknowledgePurchaseResponseListener = AcknowledgePurchaseResponseListener {
    // Check acknowledgement result
}

billingClient.acknowledgePurchase(
    acknowledgePurchaseParams,
    acknowledgePurchaseResponseListener
)
```

For consumable purchases, you will need to notify Google Play when the item has been consumed so that it is available to be repurchased by the user. This requires a configured ConsumeParams instance containing a purchase token and a call to the billing client's *consumePurchase()* method:

```
val consumeParams = ConsumeParams.newBuilder()
    .setPurchaseToken(purchase.purchaseToken)
    .build()
```

```
coroutineScope.launch {
    val result = billingClient.consumePurchase(consumeParams)

    if (result.billingResult.responseCode ==
                    BillingClient.BillingResponseCode.OK) {
        // Purchase successfully consumed
    }
}
```

61.8 Querying previous purchases

When working with in-app billing it is a common requirement to check whether a user has already purchased a product or subscription. A list of all the user's previous purchases of a specific type can be generated by calling the *queryPurchasesAsync()* method of the BillingClient instance and implementing a PurchaseResponseListener. The following code, for example, obtains a list of all previously purchased items that have not yet been consumed:

```
val queryPurchasesParams = QueryPurchasesParams.newBuilder()
    .setProductType(BillingClient.ProductType.INAPP)
    .build()

billingClient.queryPurchasesAsync(
    queryPurchasesParams,
    purchasesListener
)
.
.
private val purchasesListener =
    PurchasesResponseListener { billingResult, purchases ->

        if (!purchases.isEmpty()) {
            // Access existing active purchases
        } else {
            // No
        }
    }
```

To obtain a list of active subscriptions, change the ProductType value from INAPP to SUBS.

Alternatively, to obtain a list of the most recent purchases for each product, make a call to the BillingClient *queryPurchaseHistoryAsync()* method:

```
val queryPurchaseHistoryParams = QueryPurchaseHistoryParams.newBuilder()
    .setProductType(BillingClient.ProductType.INAPP)
    .build()

billingClient.queryPurchaseHistoryAsync(queryPurchaseHistoryParams) {
billingResult, historyList ->
    // Process purchase history list
}
```

61.9 Summary

In-app purchases provide a way to generate revenue from within Android apps by selling virtual products and subscriptions to users. In this chapter, we have explored managed products and subscriptions and explained the difference between consumable and non-consumable products. In-app purchasing support is added to an app using the Google Play In-app Billing Library and involves creating and initializing a billing client on which methods are called to perform tasks such as making purchases, listing available products, and consuming existing purchases. The next chapter contains a tutorial demonstrating the addition of in-app purchases to an Android Studio project.

Index