Jetpack Compose 1.2 Essentials

Jetpack Compose 1.2 Essentials

ISBN-13: 978-1-951442-49-1

Rev: 1.0

Contents

Table of Contents

Chapter 1

1. Start Here

This book aims to teach you how to build Android applications using Jetpack Compose 1.2, Android Studio, 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, and list components.

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

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. Using in-app billing, you will also learn to generate extra revenue from your app.

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.ebookfrenzy.com/retail/compose12/index.php

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 *feedback@ebookfrenzy.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.ebookfrenzy.com/errata/compose12.html

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

Chapter 4

4. An Example Compose Project

In the previous chapter, we created a new Compose-based Android Studio project named ComposeDemo and took some time to explore both Android Studio and some of the project code that it generated to get us started. With those basic steps covered, this chapter will use the ComposeDemo project as the basis for a new app. This will involve the creation of new composable functions, introduce the concept of state, and make use of the Preview panel in interactive mode. As with the preceding chapter, key concepts explained in basic terms here will be covered in significantly greater detail in later chapters.

4.1 Getting started

Start Android Studio if it is not already running and open the ComposeDemo project created in the previous chapter. Once the project has loaded, double-click on the *MainActivity.kt* file (located in Project tool window under *app -> java -> <package name>*) to open it in the code editor. If necessary, switch the editor into Split mode so that both the editor and Preview panel are visible.

4.2 Removing the template Code

Within the *MainActivity.kt* file, delete some of the template code so that the file reads as follows:

```
package com.example.composedemo
.
.
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            ComposeDemoTheme {
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colors.background
                ) {
                    Greeting("Android")
                }
            }
        }
    }
}

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

@Preview(showSystemUi = true)
```

```
@Composable
fun DefaultPreview() {
    ComposeDemoTheme {
        Greeting("Android")
    }
}
```

4.3 The Composable hierarchy

Before we write the composable functions that will make up our user interface, it helps to visualize the relationships between these components. The ability of one composable to call other composables essentially allows us to build a hierarchy tree of components. The ability of one composable to call other composables essentially allows us to build a hierarchy tree of components. Once completed, the composable hierarchy for our ComposeDemo main activity can be represented as shown in Figure 4-1:

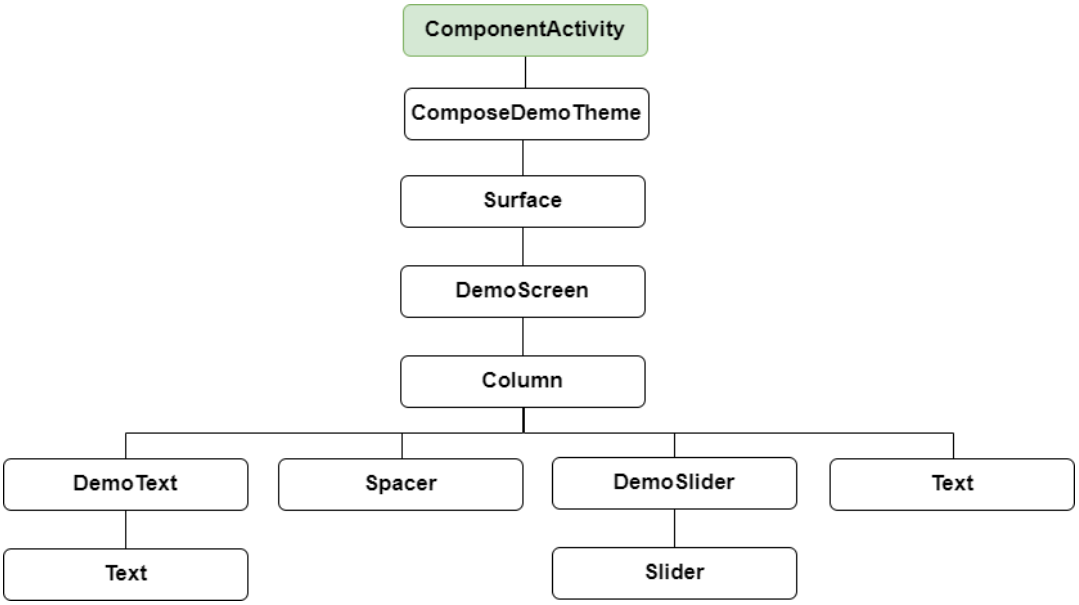

Figure 4-1

All of the elements in the above diagram, except for ComponentActivity, are composable functions. Of those functions, the Surface, Column, Spacer, Text, and Slider functions are built-in composables provided by Compose. The DemoScreen, DemoText, and DemoSlider composables, on the other hand, are functions that we will create to provide both structure to the design and the custom functionality we require for our app. You can find the ComposeDemoTheme composable declaration in the *ui.theme -> Theme.kt* file.

4.4 Adding the DemoText composable

We are now going to add a new composable function to the activity to represent the DemoText item in the hierarchy tree. The purpose of this composable is to display a text string using a font size value that adjusts in real-time as the slider moves. Place the cursor beneath the final closing brace (}) of the MainActivity declaration and add the following function declaration:

```
@Composable
fun DemoText() {
}
```

The @Composable annotation notifies the build system that this is a composable function. When the function is called, the plan is for it to be passed both a text string and the font size at which that text is to be displayed. This means that we need to add some parameters to the function:

```
@Composable
fun DemoText(message: String, fontSize: Float) {
}
```

The next step is to make sure the text is displayed. To achieve this, we will make a call to the built-in Text composable, passing through as parameters the message string, font size and, to make the text more prominent, a bold font weight setting:

```
@Composable
fun DemoText(message: String, fontSize: Float) {
    Text(
        text = message,
        fontSize = fontSize.sp,
        fontWeight = FontWeight.Bold
    )
}
```

Note that after making these changes, the code editor indicates that "sp" and "FontWeight" are undefined. This happens because these are defined and implemented in libraries that have not yet been imported into the *MainActivity.kt* file. One way to resolve this is to click on an undefined declaration so that it highlights as shown below, and then press Alt+Enter (Opt+Enter on macOS) on the keyboard to import the missing library automatically:

Figure 4-2

Alternatively, you may add the missing import statements manually to the list at the top of the file:

```
.
.
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.unit.sp
.
.
```

In the remainder of this book, all code examples will include any required library import statements.

We have now finished writing our first composable function. Notice that, except for the font weight, all the other properties are passed to the function when it is called (a function that calls another function is generally referred to as the *caller*). This increases the flexibility, and therefore re-usability, of the DemoText composable and is a key

goal to keep in mind when writing composable functions.

4.5 Previewing the DemoText composable

At this point, the Preview panel will most likely be displaying a message which reads "No preview found". The reason for this is that our *MainActivity.kt* file does not contain any composable functions prefixed with the @ Preview annotation. Add a preview composable function for DemoText to the *MainActivity.kt* file as follows:

```
@Preview
@Composable
fun DemoTextPreview() {
    DemoText(message = "Welcome to Android", fontSize = 12f)
}
```

After adding the preview composable, the Preview panel should have detected the change and displayed the link to build and refresh the preview rendering. Click the link and wait for the rebuild to complete, at which point the DemoText composable should appear as shown in Figure 4-3:

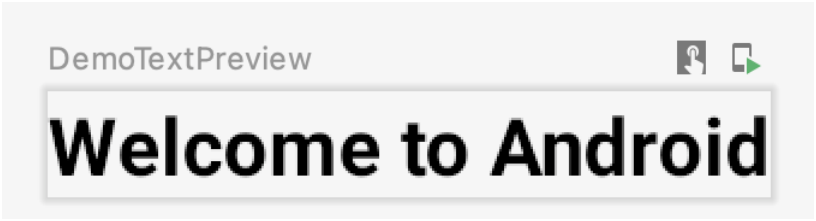

Figure 4-3

Minor changes made to the code in the *MainActivity.kt* file such as changing values will be instantly reflected in the preview without the need to build and refresh. For example, change the "Welcome to Android" text literal to "Welcome to Compose" and note that the text in the Preview panel changes as you type. Similarly, increasing the font size literal will instantly change the size of the text in the preview. This feature is referred to as Live Edit.

4.6 Adding the DemoSlider composable

The DemoSlider composable is a little more complicated than DemoText. It will need to be passed a variable containing the current slider position and an event handler function or lambda to call when the slider is moved by the user so that the new position can be stored and passed to the two Text composables. With these requirements in mind, add the function as follows:

```
.
.
import androidx.compose.foundation.layout.*
import androidx.compose.material.Slider
import androidx.compose.ui.unit.dp
.
.
@Composable
fun DemoSlider(sliderPosition: Float, onPositionChange: (Float) -> Unit ) {
    Slider(
        modifier = Modifier.padding(10.dp),
        valueRange = 20f..40f,
        value = sliderPosition,
        onValueChange = { onPositionChange(it) }
```

```
    )
}
```

The DemoSlider declaration contains a single Slider composable which is, in turn, passed four parameters. The first is a Modifier instance configured to add padding space around the slider. Modifier is a Kotlin class built into Compose which allows a wide range of properties to be set on a composable within a single object. Modifiers can also be created and customized in one composable before being passed to other composables where they can be further modified before being applied.

The second value passed to the Slider is a range allowed for the slider value (in this case the slider is limited to values between 20 and 40).

The next parameter sets the value of the slider to the position passed through by the caller. This ensures that each time DemoSlider is recomposed it retains the last position value.

Finally, we set the *onValueChange* parameter of the Slider to call the function or lambda we will be passing to the DemoSlider composable when we call it later. Each time the slider position changes, the call will be made and passed the current value which we can access via the Kotlin *it* keyword. We can further simplify this by assigning just the event handler parameter name (*onPositionChange*) and leaving the compiler to handle the passing of the current value for us:

```
onValueChange = onPositionChange
```

4.7 Adding the DemoScreen composable

The next step in our project is to add the DemoScreen composable. This will contain a variable named *sliderPosition* in which to store the current slider position and the implementation of the *handlePositionChange* event handler to be passed to the DemoSlider. This lambda will be responsible for storing the current position in the *sliderPosition* variable each time it is called with an updated value. Finally, DemoScreen will contain a Column composable configured to display the DemoText, Spacer, DemoSlider and the second, as yet to be added, Text composable in a vertical arrangement.

Start by adding the DemoScreen function as follows:

```
.
.
import androidx.compose.runtime.*
.
.
@Composable
fun DemoScreen() {

    var sliderPosition by remember { mutableStateOf(20f) }

    val handlePositionChange = { position : Float ->
        sliderPosition = position
    }
}
```

The *sliderPosition* variable declaration requires some explanation. As we will learn later, the Compose system repeatedly and rapidly *recomposes* user interface layouts in response to data changes. The change of slider position will, therefore, cause DemoScreen to be recomposed along with all of the composables it calls. Consider if we had declared and initialized our *sliderPosition* variable as follows:

```
var sliderPosition = 20f
```

Suppose the user slides the slider to position 21. The *handlePositionChange* event handler is called and stores the new value in the *sliderPosition* variable as follows:

```
val handlePositionChange = { position : Float ->
    sliderPosition = position
}
```

The Compose runtime system detects this data change and recomposes the user interface, including a call to the DemoScreen function. This will, in turn, reinitialize the *sliderposition* variable to 20 causing the previous value of 21 to be lost. Declaring the *sliderPosition* variable in this way informs Compose that the current value needs to be remembered during recompositions:

```
var sliderPosition by remember { mutableStateOf(20f) }
```

The only remaining work within the DemoScreen implementation is to add a Column containing the required composable functions:

```
.
.
import androidx.compose.ui.Alignment
.
.
@Composable
fun DemoScreen() {

    var sliderPosition by remember { mutableStateOf(20f) }

    val handlePositionChange = { position : Float ->
        sliderPosition = position
    }

    Column(
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center,
        modifier = Modifier.fillMaxSize()
    ) {

        DemoText(message = "Welcome to Compose", fontSize = sliderPosition)

        Spacer(modifier = Modifier.height(150.dp))

        DemoSlider(
            sliderPosition = sliderPosition,
            onPositionChange = handlePositionChange
        )

        Text(
            style = MaterialTheme.typography.h2,
```

```
                text = sliderPosition.toInt().toString() + "sp"
            )
        }
}
```

Points to note regarding these changes may be summarized as follows:

- When DemoSlider is called, it is passed a reference to our handlePositionChange event handler as the onPositionChange parameter.
- The Column composable accepts parameters that customize layout behavior. In this case, we have configured the column to center its children both horizontally and vertically.
- A Modifier has been passed to the Spacer to place a 150dp vertical space between the DemoText and DemoSlider components.
- The second Text composable is configured to use the h2 (Heading 2) style of the Material theme. In addition, the *sliderPosition* value is converted from a Float to an integer so that only whole numbers are displayed and then converted to a string value before being displayed to the user.

4.8 Previewing the DemoScreen composable

To confirm that the DemoScreen layout meets our expectations, we need to modify the DemoTextPreview composable:

```
.
.
@Preview(showSystemUi = true)
@Composable
fun Preview() {
    ComposeDemoTheme {
        DemoScreen()
    }
}
```

Note that we have enabled the *showSystemUi* property of the preview so that we will experience how the app will look when running on an Android device.

After performing a preview rebuild and refresh, the user interface should appear as originally shown in Figure 3-1.

4.9 Testing in interactive mode

At this stage, we know that the user interface layout for our activity looks how we want it to, but we don't know if it will behave as intended. One option is to run the app on an emulator or physical device (topics which are covered in later chapters). A quicker option, however, is to switch the preview panel into interactive mode. This is achieved by clicking on the button indicated in Figure 4-4 below:

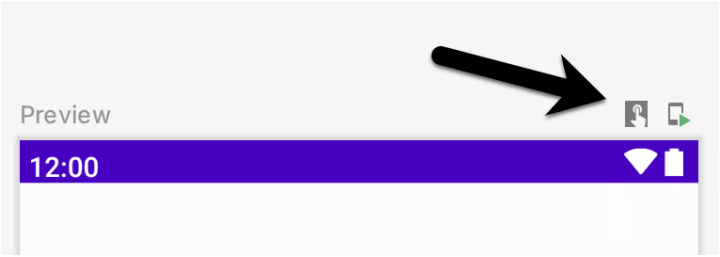

Figure 4-4

When clicked, there will be a short delay when interactive mode starts, after which it should be possible to move the slider and watch the two Text components update:

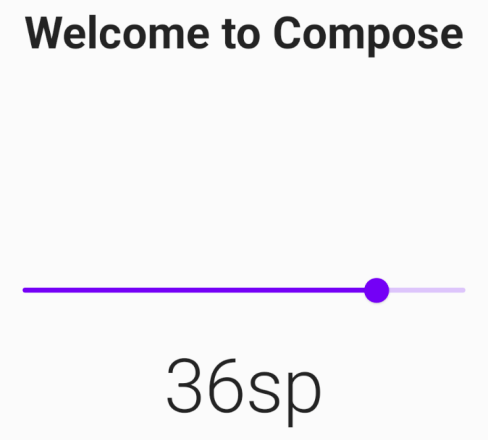

Figure 4-5

Click the stop button (marked A in Figure 4-6 below) to exit interactive mode. If it appears that the preview needs to be refreshed, click on the *Build Refresh* button (B):

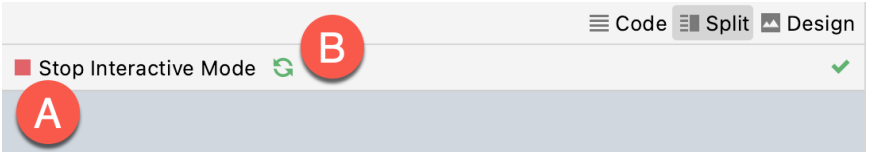

Figure 4-6

4.10 Completing the project

The final step is to make sure that the DemoScreen composable is called from within the Surface function located in the *onCreate()* method of the MainActivity class. Locate this method and modify it as follows:

```
.
.
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            ComposeDemoTheme {
                Surface(
```

```
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colors.background
                ) {
                    DemoScreen()
                }
            }
        }
    }
}
```

This will ensure that, in addition to appearing in the preview panel, our user interface will also be displayed when the app runs on a device or emulator (a topic that will be covered in later chapters).

4.11 Summary

In this chapter, we have extended our ComposeDemo project to include some additional user interface elements in the form of two Text composables, a Spacer, and a Slider. These components were arranged vertically using a Column composable. We also introduced the concept of mutable state variables and explained how they are used to ensure that the app remembers state when the Compose runtime performs recompositions. The example also demonstrated how to use event handlers to respond to user interaction (in this case, the user moving a slider). Finally, we made use of the Preview panel in interactive mode to test the app without the need to compile and run it on an emulator or physical device.

Chapter 9

9. The Basics of the Android Studio Code Editor

Developing applications for Android involves a considerable amount of programming work which, by definition, involves typing, reviewing, and modifying lines of code. It should come as no surprise that the majority of a developer's time spent using Android Studio will typically involve editing code within the editor window.

The modern code editor needs to go far beyond the original basics of typing, deleting, cutting, and pasting. Today the usefulness of a code editor is generally gauged by factors such as the amount by which it reduces the typing required by the programmer, ease of navigation through large source code files, and the editor's ability to detect and highlight programming syntax errors in real-time as the code is being written. As will become evident in this chapter, these are just a few of the areas in which the Android Studio editor excels.

While not an exhaustive overview of the features of the Android Studio editor, this chapter aims to provide a guide to the key features of the tool. Experienced programmers will find that some of these features are common to most code editors available today, while a number are unique to this particular editing environment.

9.1 The Android Studio editor

The Android Studio editor appears in the center of the main window when a Java, Kotlin, XML, or other text-based file is selected for editing. Figure 9-1, for example, shows a typical editor session with a Kotlin source code file loaded:

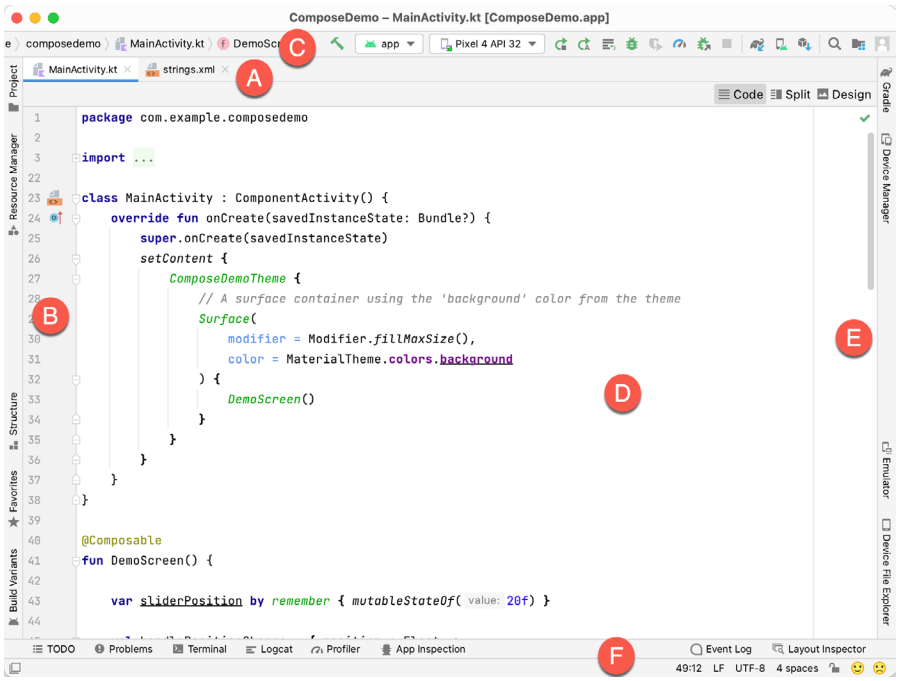

Figure 9-1

The elements that comprise the editor window can be summarized as follows:

A – Document Tabs – Android Studio is capable of holding multiple files open for editing at any one time. As each file is opened, it is assigned a document tab displaying the file name in the tab bar located along the top edge of the editor window. A small drop-down menu will appear in the far right-hand corner of the tab bar when there is insufficient room to display all of the tabs. Clicking on this menu will drop down a list of additional open files. A wavy red line underneath a file name in a tab indicates that the code in the file contains one or more errors that need to be addressed before the project can be compiled and run.

Switching between files is simply a matter of clicking on the corresponding tab or using the Alt-Left and Alt-Right keyboard shortcuts. Navigation between files may also be performed using the Switcher mechanism (accessible via the Ctrl-Tab keyboard shortcut).

To detach an editor panel from the Android Studio main window so that it appears in a separate window, click on the tab and drag it to an area on the desktop outside of the main window. To return the editor to the main window, click on the file tab in the separated editor window and drag and drop it onto the original editor tab bar in the main window.

B – The Editor Gutter Area - The gutter area is used by the editor to display informational icons and controls. Some typical items, among others, which appear in this gutter area are debugging breakpoint markers, controls to fold and unfold blocks of code, bookmarks, change markers, and line numbers. Line numbers are switched on by default but may be disabled by right-clicking in the gutter and selecting the Show Line Numbers menu option.

C – Code Structure Location - This bar at the bottom of the editor displays the current position of the cursor as it relates to the overall structure of the code. In the following figure, for example, the bar indicates that the *onCreate()* method is currently being edited and that this method is contained within the MainActivity class.

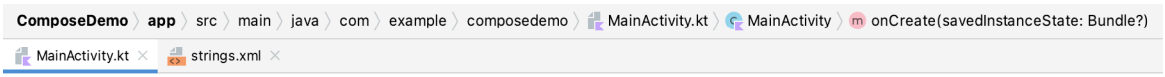

Figure 9-2

Double-clicking an element within the bar will move the cursor to the corresponding location within the code file. For example, double-clicking on the *onCreate()* entry will move the cursor to the top of that method within the source code. Similarly clicking on the *MainActivity.kt* entry will drop down a list of available code navigation points for selection:

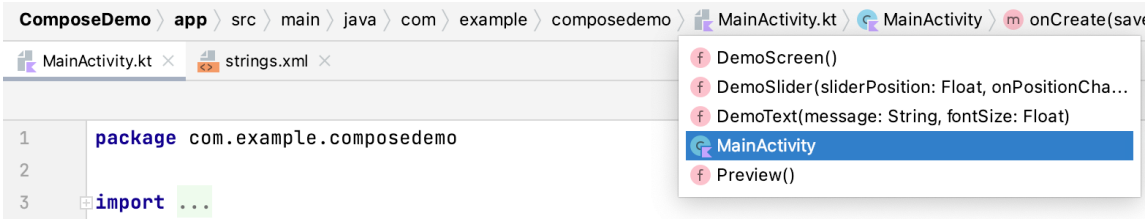

Figure 9-3

D – The Editor Area – This is the main area where the code is displayed, entered, and edited by the user. Later sections of this chapter will cover the key features of the editing area in detail.

E – The Validation and Marker Sidebar – Android Studio incorporates a feature referred to as "on-the-fly code analysis". What this essentially means is that as you are typing code, the editor is analyzing the code to check for warnings and syntax errors. The indicators at the top of the validation sidebar will update in real-time to indicate the number of errors and warnings found as code is added. Clicking on this indicator will display a popup

containing a summary of the issues found with the code in the editor as illustrated in Figure 9-4:

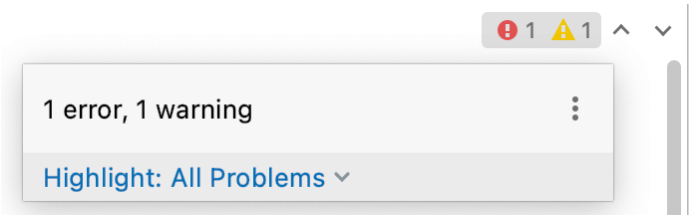

Figure 9-4

The up and down arrows may be used to move between the error locations within the code. A green checkmark indicates that no warnings or errors have been detected.

The sidebar also displays markers at the locations where issues have been detected using the same color-coding. Hovering the mouse pointer over a marker when the line of code is visible in the editor area will display a popup containing a description of the issue:

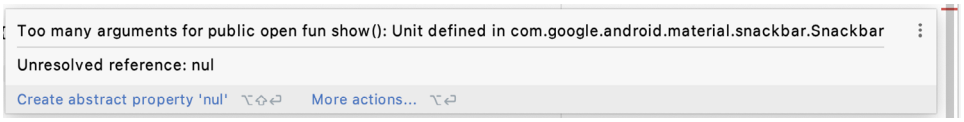

Figure 9-5

Hovering the mouse pointer over a marker for a line of code that is currently scrolled out of the viewing area of the editor will display a "lens" overlay containing the block of code where the problem is located (Figure 9-6) allowing it to be viewed without needing to scroll to that location in the editor:

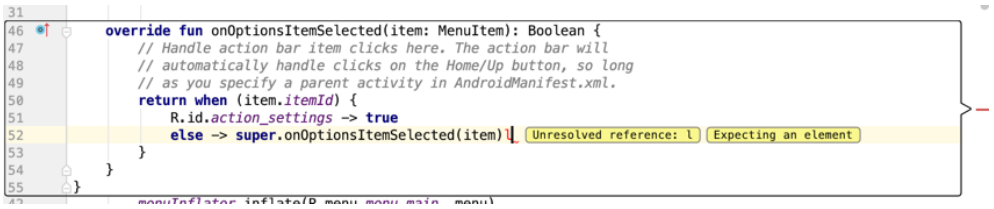

Figure 9-6

It is also worth noting that the lens overlay is not limited to warnings and errors in the sidebar. Hovering over any part of the sidebar will result in a lens appearing containing the code present at that location within the source file.

F – The Status Bar – Though the status bar is part of the main window, as opposed to the editor, it does contain some information about the currently active editing session. This information includes the current position of the cursor in terms of lines and characters and the encoding format of the file (UTF-8, ASCII, etc.). Clicking on these values in the status bar allows the corresponding setting to be changed. Clicking on the line number, for example, displays the Go to Line dialog.

Having provided an overview of the elements that comprise the Android Studio editor, the remainder of this chapter will explore the key features of the editing environment in more detail.

9.2 Code mode

The code editor has three modes in which it can be placed using the buttons located in the top right-hand corner of the editor panel. In Figure 9-7 below, for example, Code mode has been selected:

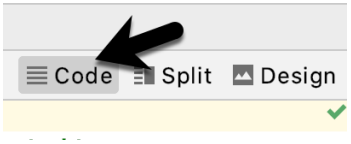

Figure 9-7

When in code mode, only the code editor panel is displayed and the Preview panel is hidden from view. In Split mode, the editor shows the Code and Preview panels side-by-side. In Design mode, only the Preview panel is displayed.

9.3 Splitting the editor window

By default, the editor will display a single panel showing the content of the currently selected file. A particularly useful feature when working simultaneously with multiple source code files is the ability to split the editor into multiple panes. To split the editor, right-click on a file tab within the editor window and select either the Split Vertically or Split Horizontally menu option. Figure 9-8, for example, shows the splitter in action with the editor split into three panels:

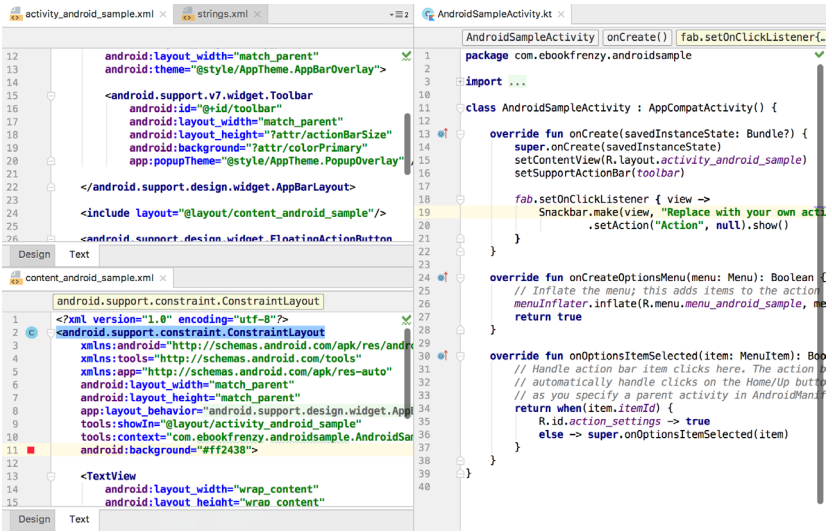

Figure 9-8

The orientation of a split panel may be changed at any time by right-clicking on the corresponding tab and selecting the Change Splitter Orientation menu option. Repeat these steps to unsplit a single panel, this time selecting the Unsplit option from the menu. All of the split panels may be removed by right-clicking on any tab and selecting the Unsplit All menu option.

Window splitting may be used to display different files, or to provide multiple windows onto the same file, allowing different areas of the same file to be viewed and edited concurrently.

9.4 Code completion

The Android Studio editor has a considerable amount of built-in knowledge of Kotlin and Compose programming syntax and the classes and methods that make up the Android SDK, as well as knowledge of your codebase. As code is typed, the editor scans what is being typed and, where appropriate, makes suggestions about what might be needed to complete a statement or reference. When a completion suggestion is detected by the editor, a panel will appear containing a list of suggestions. In Figure 9-9, for example, the editor is suggesting possibilities for the beginning of a String declaration:

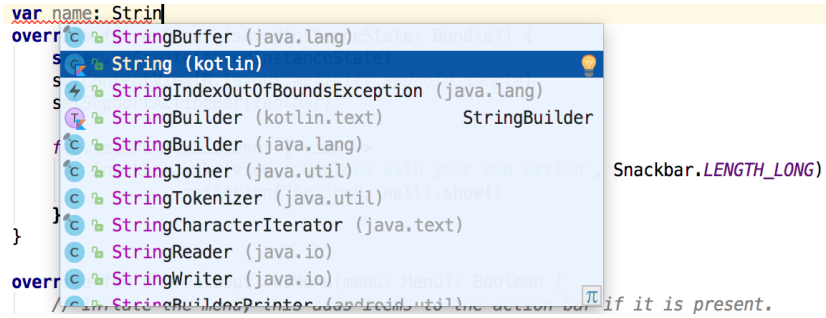

Figure 9-9

If none of the auto-completion suggestions are correct, simply keep typing and the editor will continue to refine the suggestions where appropriate. To accept the topmost suggestion, simply press the Enter or Tab key on the keyboard. To select a different suggestion, use the arrow keys to move up and down the list, once again using the Enter or Tab key to select the highlighted item.

Completion suggestions can be manually invoked using the Ctrl-Space keyboard sequence. This can be useful when changing a word or declaration in the editor. When the cursor is positioned over a word in the editor, that word will automatically highlight. Pressing Ctrl-Space will display a list of alternate suggestions. To replace the current word with the currently highlighted item in the suggestion list, simply press the Tab key.

In addition to the real-time auto-completion feature, the Android Studio editor also offers a system referred to as Smart Completion. Smart completion is invoked using the Shift-Ctrl-Space keyboard sequence and, when selected, will provide more detailed suggestions based on the current context of the code. Pressing the Shift-Ctrl-Space shortcut sequence a second time will provide more suggestions from a wider range of possibilities.

Code completion can be a matter of personal preference for many programmers. In recognition of this fact, Android Studio provides a high level of control over the auto-completion settings. These can be viewed and modified by selecting the *File -> Settings…* menu option (or *Android Studio -> Preferences…* on macOS) and choosing *Editor -> General -> Code Completion* from the settings panel as shown in Figure 9-10:

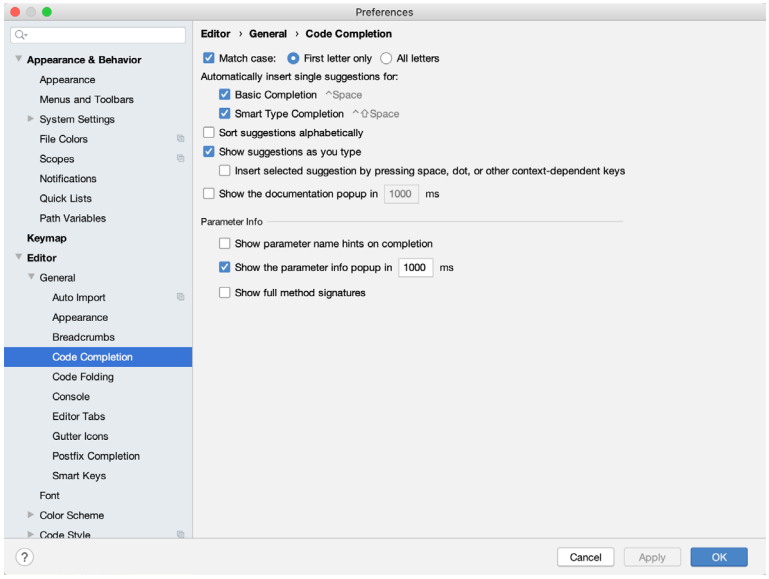

Figure 9-10

9.5 Statement completion

Another form of auto-completion provided by the Android Studio editor is statement completion. This can be used to automatically fill out the parentheses and braces for items such as methods and loop statements. Statement completion is invoked using the Shift-Ctrl-Enter (Shift-Cmd-Enter on macOS) keyboard sequence. Consider for example the following code:

```
fun myMethod()
```

Having typed this code into the editor, triggering statement completion will cause the editor to automatically add the braces to the method:

```
fun myMethod() {


}
```

9.6 Parameter information

It is also possible to ask the editor to provide information about the argument parameters accepted by a method. With the cursor positioned between the brackets of a method call, the Ctrl-P (Cmd-P on macOS) keyboard sequence will display the parameters known to be accepted by that method, with the most likely suggestion highlighted in bold:

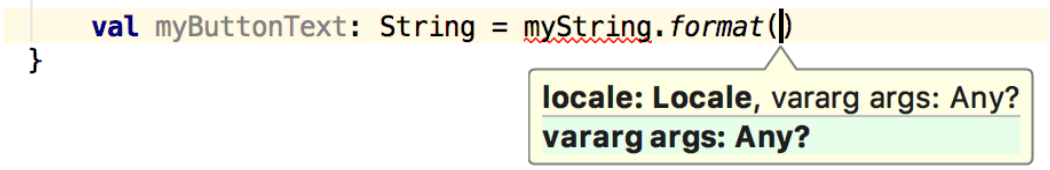

Figure 9-11

9.7 Parameter name hints

The code editor may be configured to display parameter name hints within method calls. Figure 9-12, for example, highlights the parameter name hints within the calls to the *make()* and *setAction()* methods of the Snackbar class:

Figure 9-12

The settings for this mode may be configured by selecting the *File -> Settings* menu (*Android Studio -> Preferences* on macOS) option followed by *Editor -> Inlay Hints -> Kotlin* in the left-hand panel. On the resulting screen, select the Parameter Hints item from the list and enable or disable the Show parameter hints option. To adjust the hint settings, click on the *Exclude list...* link and make any necessary adjustments.

9.8 Code generation

In addition to completing code as it is typed the editor can, under certain conditions, also generate code for you. The list of available code generation options shown in Figure 9-13 can be accessed using the Alt-Insert (Cmd-N on macOS) keyboard shortcut when the cursor is at the location in the file where the code is to be generated.

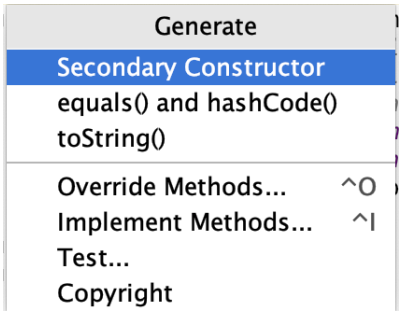

Figure 9-13

For example, consider a situation where we want to be notified when an Activity in our project is about to be destroyed by the operating system. This can be achieved by overriding the *onStop()* lifecycle method of the Activity superclass. To have Android Studio generate a stub method for this, simply select the *Override Methods…* option from the code generation list and select the *onStop()* method from the resulting list of available methods:

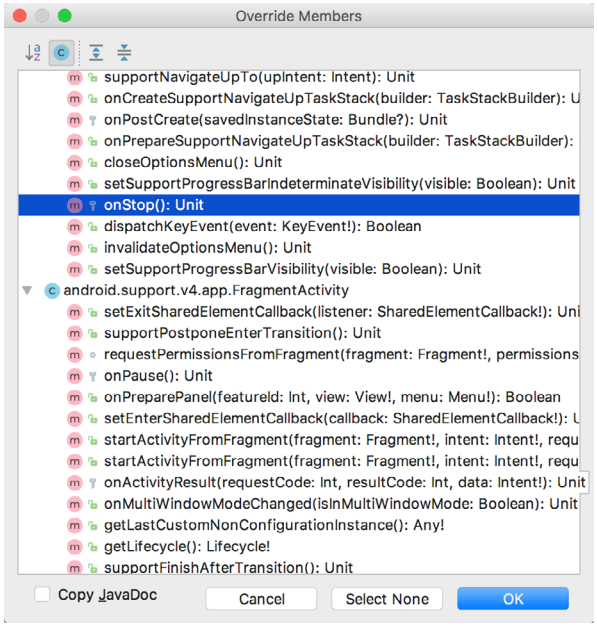

Figure 9-14

Having selected the method to override, clicking on OK will generate the stub method at the current cursor location in the Kotlin source file as follows:

```
override fun onStop() {
    super.onStop()
}
```

9.9 Code folding

Once a source code file reaches a certain size, even the most carefully formatted and well-organized code can become overwhelming and difficult to navigate. Android Studio takes the view that it is not always necessary to have the content of every code block visible at all times. Code navigation can be made easier through the use of the code folding feature of the Android Studio editor. Code folding is controlled using markers appearing in the editor gutter at the beginning and end of each block of code in a source file. Figure 9-15, for example, highlights

the start and end markers for code that is not currently folded:

Figure 9-15

Clicking on either of these markers will fold the statement such that only the signature line is visible as shown in Figure 9-16:

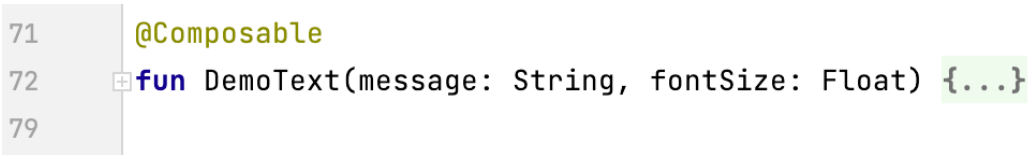

Figure 9-16

To unfold a collapsed section of code, click on the '+' marker in the editor gutter. To see the hidden code without unfolding it, hover the mouse pointer over the "{…}" indicator as shown in Figure 9-17. The editor will then display the lens overlay containing the folded code block:

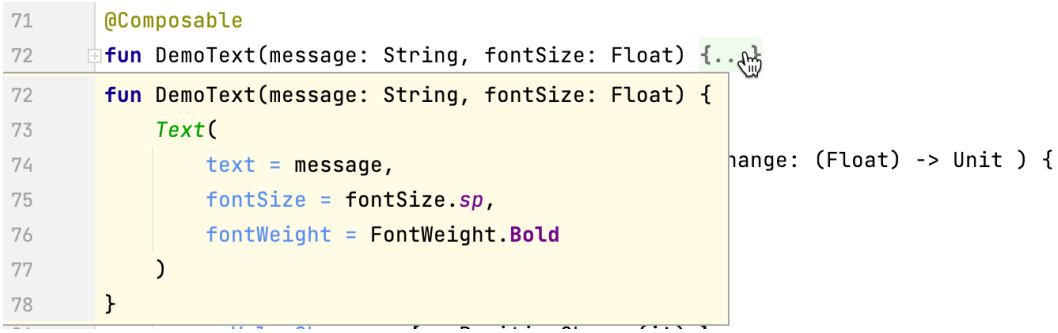

Figure 9-17

All of the code blocks in a file may be folded or unfolded using the Ctrl-Shift-Plus and Ctrl-Shift-Minus keyboard sequences.

By default, the Android Studio editor will automatically fold some code when a source file is opened. To configure the conditions under which this happens, select *File -> Settings…* (*Android Studio -> Preferences…* on macOS) and choose the *Editor -> General -> Code Folding* entry in the resulting settings panel (Figure 9-18):

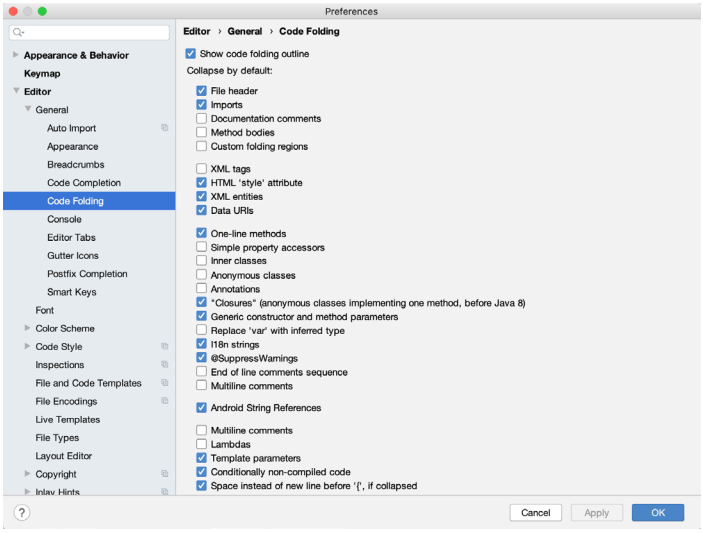

Figure 9-18

9.10 Quick documentation lookup

Context-sensitive Kotlin and Android documentation can be accessed by hovering the cursor over the declaration for which documentation is required. This will display a popup containing the relevant reference documentation for the item. Figure 9-19, for example, shows the documentation for the Android Bundle class.

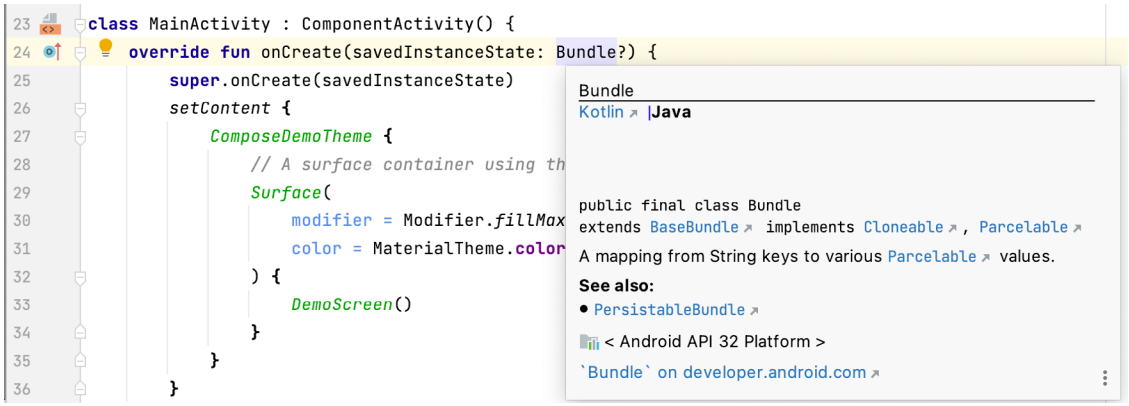

Figure 9-19

Once displayed, the documentation popup can be moved around the screen as needed.

9.11 Code reformatting

In general, the Android Studio editor will automatically format code in terms of indenting, spacing, and nesting of statements and code blocks as they are added. In situations where lines of code need to be reformatted (a common occurrence, for example, when cutting and pasting sample code from a website), the editor provides a source code reformatting feature which, when selected, will automatically reformat code to match the prevailing code style.

To reformat source code, press the Ctrl-Alt-L (Cmd-Opt-L on macOS) keyboard shortcut sequence. To display the Reformat File dialog (Figure 9-20) use the Ctrl-Alt-Shift-L (Cmd-Opt-Shift-L on macOS). This dialog provides the option to reformat only the currently selected code, the entire source file currently active in the

editor, or only code that has changed as the result of a source code control update.

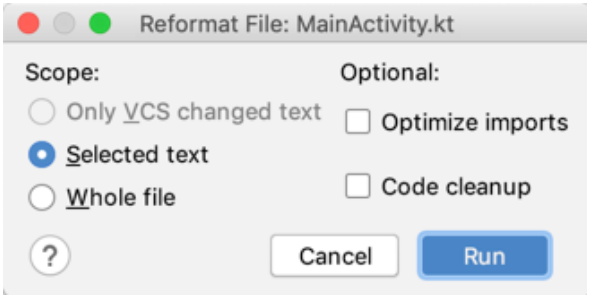

Figure 9-20

The full range of code style preferences can be changed from within the project settings dialog. Select the *File -> Settings* menu option (*Android Studio -> Preferences…* on macOS) and choose Code Style in the left-hand panel to access a list of supported programming and markup languages. Selecting a language will provide access to a vast array of formatting style options, all of which may be modified from the Android Studio default to match your preferred code style. To configure the settings for the Rearrange code option in the above dialog, for example, unfold the Code Style section, select Kotlin and, from the Kotlin settings, select the Arrangement tab.

9.12 Finding sample code

The Android Studio editor provides a way to access sample code relating to the currently highlighted entry within the code listing. This feature can be useful for learning how a particular Android class or method is used. To find sample code, highlight a method or class name in the editor, right-click on it and select the *Find Sample Code* menu option. The Find Sample Code panel (Figure 9-21) will appear beneath the editor with a list of matching samples. Selecting a sample from the list will load the corresponding code into the right-hand panel:

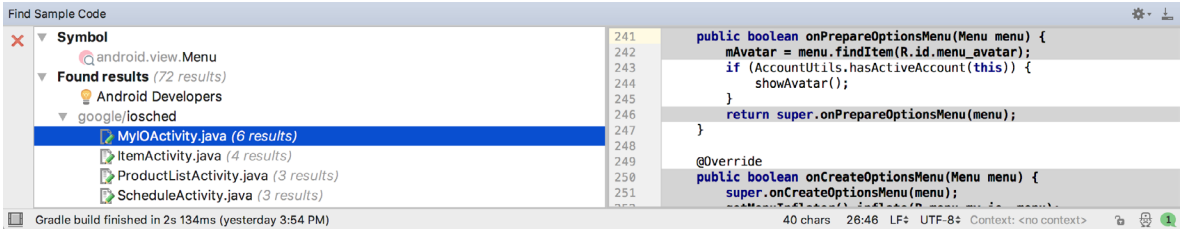

Figure 9-21

9.13 Live templates

As you write Android code you will find that there are common constructs that are used frequently. For example, a common requirement is to display a popup message to the user using the Android Toast class. Live templates are a collection of common code constructs that can be entered into the editor by typing the initial characters followed by a special key (set to the Tab key by default) to insert template code. To experience this in action, type toast in the code editor followed by the Tab key, and Android Studio will insert the following code at the cursor position ready for editing:

```
Toast.makeText(, "", Toast.LENGTH_SHORT).show()
```

To list and edit existing templates, change the special key, or add your own templates, open the Preferences dialog and select Live Templates from the Editor section of the left-hand navigation panel:

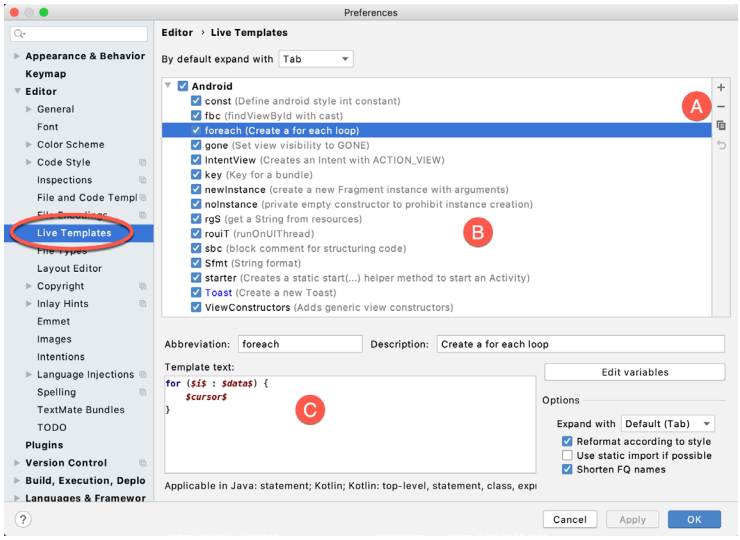

Figure 9-22

Add, remove, duplicate or reset templates using the buttons marked A in Figure 9-22 above. To modify a template, select it from the list (B) and change the settings in the panel marked C.

9.14 Summary

The Android Studio editor goes to great length to reduce the amount of typing needed to write code and to make that code easier to read and navigate. In this chapter, we have covered many of the key editor features including code completion, code generation, editor window splitting, code folding, reformatting, documentation lookup, and live templates.

Chapter 12

12. Kotlin Data Types, Variables and Nullability

Both this and the following few chapters are intended to introduce the basics of the Kotlin programming language. This chapter will focus on the various data types available for use within Kotlin code. This will also include an explanation of constants, variables, type casting and Kotlin's handling of null values.

As outlined in the previous chapter, entitled *"An Introduction to Kotlin"* a useful way to experiment with the language is to use the Kotlin online playground environment. Before starting this chapter, therefore, open a browser window, navigate to *https://play.kotlinlang.org* and use the playground to try out the code in both this and the other Kotlin introductory chapters that follow.

12.1 Kotlin data types

When we look at the different types of software that run on computer systems and mobile devices, from financial applications to graphics intensive games, it is easy to forget that computers are really just binary machines. Binary systems work in terms of 0 and 1, true or false, set and unset. All the data sitting in RAM, stored on disk drives and flowing through circuit boards and buses are nothing more than sequences of 1s and 0s. Each 1 or 0 is referred to as a bit and bits are grouped together in blocks of 8, each group being referred to as a byte. When people talk about 32-bit and 64-bit computer systems they are talking about the number of bits that can be handled simultaneously by the CPU bus. A 64-bit CPU, for example, is able to handle data in 64-bit blocks, resulting in faster performance than a 32-bit based system.

Humans, of course, don't think in binary. We work with decimal numbers, letters and words. For a human to easily ('easily' being a relative term in this context) program a computer, some middle ground between human and computer thinking is needed. This is where programming languages such as Kotlin come into play. Programming languages allow humans to express instructions to a computer in terms and structures we understand, and then compile that down to a format that can be executed by a CPU.

One of the fundamentals of any program involves data, and programming languages such as Kotlin define a set of *data types* that allow us to work with data in a format we understand when programming. For example, if we want to store a number in a Kotlin program we could do so with syntax similar to the following:

```
val mynumber = 10
```

In the above example, we have created a variable named *mynumber* and then assigned to it the value of 10. When we compile the source code down to the machine code used by the CPU, the number 10 is seen by the computer in binary as:

```
1010
```

Similarly, we can express a letter, the visual representation of a digit ('0' through to '9') or punctuation mark (referred to in computer terminology as *characters*) using the following syntax:

```
val myletter = 'c'
```

Once again, this is understandable by a human programmer, but gets compiled down to a binary sequence for the CPU to understand. In this case, the letter 'c' is represented by the decimal number 99 using the ASCII table (an internationally recognized standard that assigns numeric values to human readable characters). When

converted to binary, it is stored as:

```
10101100011
```

Now that we have a basic understanding of the concept of data types and why they are necessary we can take a closer look at some of the more commonly used data types supported by Kotlin.

12.1.1 Integer data types

Kotlin integer data types are used to store whole numbers (in other words a number with no decimal places). All integers in Kotlin are signed (in other words capable of storing positive, negative and zero values).

Kotlin provides support for 8, 16, 32 and 64 bit integers (represented by the Byte, Short, Int and Long types respectively).

12.1.2 Floating point data types

The Kotlin floating-point data types are able to store values containing decimal places. For example, 4353.1223 would be stored in a floating-point data type. Kotlin provides two floating-point data types in the form of Float and Double. Which type to use depends on the size of value to be stored and the level of precision required. The Double type can be used to store up to 64-bit floating-point numbers. The Float data type, on the other hand, is limited to 32-bit floating-point numbers.

12.1.3 Boolean data type

Kotlin, like other languages, includes a data type for the purpose of handling true or false (1 or 0) conditions. Two Boolean constant values (*true* and *false*) are provided by Kotlin specifically for working with Boolean data types.

12.1.4 Character data type

The Kotlin Char data type is used to store a single character of rendered text such as a letter, numerical digit, punctuation mark or symbol. Internally characters in Kotlin are stored in the form of 16-bit Unicode grapheme clusters. A grapheme cluster is made of two or more Unicode code points that are combined to represent a single visible character.

The following lines assign a variety of different characters to Character type variables:

```
val myChar1 = 'f'
val myChar2 = ':'
val myChar3 = 'X'
```

Characters may also be referenced using Unicode code points. The following example assigns the 'X' character to a variable using Unicode:

```
val myChar4 = '\u0058'
```

Note the use of single quotes when assigning a character to a variable. This indicates to Kotlin that this is a Char data type as opposed to double quotes which indicate a String data type.

12.1.5 String data type

The String data type is a sequence of characters that typically make up a word or sentence. In addition to providing a storage mechanism, the String data type also includes a range of string manipulation features allowing strings to be searched, matched, concatenated and modified. Double quotes are used to surround single line strings during assignment, for example:

```
val message = "You have 10 new messages."
```

Alternatively, a multi-line string may be declared using triple quotes

```
val message = """You have 10 new messages,
                              5 old messages
               and 6 spam messages."""
```

The leading spaces on each line of a multi-line string can be removed by making a call to the *trimMargin()* function of the String data type:

```
val message = """You have 10 new messages,
                              5 old messages
               and 6 spam messages.""".trimMargin()
```

Strings can also be constructed using combinations of strings, variables, constants, expressions, and function calls using a concept referred to as string interpolation. For example, the following code creates a new string from a variety of different sources using string interpolation before outputting it to the console:

```
val username = "John"
val inboxCount = 25
val maxcount = 100
val message = "$username has $inboxCount messages. Message capacity remaining is
${maxcount - inboxCount} messages"


println(message)
```

When executed, the code will output the following message:

```
John has 25 messages. Message capacity remaining is 75 messages.
```

12.1.6 Escape sequences

In addition to the standard set of characters outlined above, there is also a range of special characters (also referred to as escape characters) available for specifying items such as a new line, tab or a specific Unicode value within a string. These special characters are identified by prefixing the character with a backslash (a concept referred to as escaping). For example, the following assigns a new line to the variable named newline:

```
var newline = '\n'
```

In essence, any character that is preceded by a backslash is considered to be a special character and is treated accordingly. This raises the question as to what to do if you actually want a backslash character. This is achieved by escaping the backslash itself:

```
var backslash = '\\'
```

The complete list of special characters supported by Kotlin is as follows:

- \n - New line
- \r - Carriage return
- \t - Horizontal tab
- \\ - Backslash
- \" - Double quote (used when placing a double quote into a string declaration)
- \' - Single quote (used when placing a single quote into a string declaration)
- \$ - Used when a character sequence containing a $ is misinterpreted as a variable in a string template.
- \unnnn – Double byte Unicode scalar where nnnn is replaced by four hexadecimal digits representing the

Unicode character.

12.2 Mutable variables

Variables are essentially locations in computer memory reserved for storing the data used by an application. Each variable is given a name by the programmer and assigned a value. The name assigned to the variable may then be used in the Kotlin code to access the value assigned to that variable. This access can involve either reading the value of the variable or, in the case of *mutable variables*, changing the value.

12.3 Immutable variables

Often referred to as a *constant,* an immutable variable is similar to a mutable variable in that it provides a named location in memory to store a data value. Immutable variables differ in one significant way in that once a value has been assigned it cannot subsequently be changed.

Immutable variables are particularly useful if there is a value which is used repeatedly throughout the application code. Rather than use the value each time, it makes the code easier to read if the value is first assigned to a constant which is then referenced in the code. For example, it might not be clear to someone reading your Kotlin code why you used the value 5 in an expression. If, instead of the value 5, you use an immutable variable named *interestRate* the purpose of the value becomes much clearer. Immutable values also have the advantage that if the programmer needs to change a widely used value, it only needs to be changed once in the constant declaration and not each time it is referenced.

12.4 Declaring mutable and immutable variables

Mutable variables are declared using the *var* keyword and may be initialized with a value at creation time. For example:

```
var userCount = 10
```

If the variable is declared without an initial value, the type of the variable must also be declared (a topic which will be covered in more detail in the next section of this chapter). The following, for example, is a typical declaration where the variable is initialized after it has been declared:

```
var userCount: Int
userCount = 42
```

Immutable variables are declared using the *val* keyword.

```
val maxUserCount = 20
```

As with mutable variables, the type must also be specified when declaring the variable without initializing it:

```
val maxUserCount: Int
maxUserCount = 20
```

When writing Kotlin code, immutable variables should always be used in preference to mutable variables whenever possible.

12.5 Data types are objects

All of the above data types are objects, each of which provides a range of functions and properties that may be used to perform a variety of different type specific tasks. These functions and properties are accessed using so-called dot notation. Dot notation involves accessing a function or property of an object by specifying the variable name followed by a dot followed in turn by the name of the property to be accessed or function to be called.

A string variable, for example, can be converted to uppercase via a call to the *toUpperCase()* function of the String class:

```
val myString = "The quick brown fox"
val uppercase = myString.toUpperCase()
```

Similarly, the length of a string is available by accessing the length property:

```
val length = myString.length
```

Functions are also available within the String class to perform tasks such as comparisons and checking for the presence of a specific word. The following code, for example, will return a *true* Boolean value since the word "fox" appears within the string assigned to the *myString* variable:

```
val result = myString.contains("fox")
```

All of the number data types include functions for performing tasks such as converting from one data type to another such as converting an Int to a Float:

```
val myInt = 10
val myFloat = myInt.toFloat()
```

A detailed overview of all of the properties and functions provided by the Kotlin data type classes is beyond the scope of this book (there are hundreds). An exhaustive list for all data types can, however, be found within the Kotlin reference documentation available online at:

https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/

12.6 Type annotations and type inference

Kotlin is categorized as a statically typed programming language. This essentially means that once the data type of a variable has been identified, that variable cannot subsequently be used to store data of any other type without inducing a compilation error. This contrasts to loosely typed programming languages where a variable, once declared, can subsequently be used to store other data types.

There are two ways in which the type of a variable will be identified. One approach is to use a type annotation at the point the variable is declared in the code. This is achieved by placing a colon after the variable name followed by the type declaration. The following line of code, for example, declares a variable named userCount as being of type Int:

```
val userCount: Int = 10
```

In the absence of a type annotation in a declaration, the Kotlin compiler uses a technique referred to as *type inference* to identify the type of the variable. When relying on type inference, the compiler looks to see what type of value is being assigned to the variable at the point that it is initialized and uses that as the type. Consider, for example, the following variable declarations:

```
var signalStrength = 2.231
val companyName = "My Company"
```

During compilation of the above lines of code, Kotlin will infer that the *signalStrength* variable is of type Double (type inference in Kotlin defaults to Double for all floating-point numbers) and that the companyName constant is of type String.

When a constant is declared without a type annotation it must be assigned a value at the point of declaration:

```
val bookTitle = "Android Studio Development Essentials"
```

If a type annotation is used when the constant is declared, however, the value can be assigned later in the code. For example:

```
val iosBookType = false
val bookTitle: String
```

```
if (iosBookType) {
         bookTitle = "iOS App Development Essentials"
} else {
         bookTitle = "Android Studio Development Essentials"
}
```

12.7 Nullable type

Kotlin nullable types are a concept that does not exist in most other programming languages (with the exception of the *optional* type in Swift). The purpose of nullable types is to provide a safe and consistent approach to handling situations where a variable may have a null value assigned to it. In other words, the objective is to avoid the common problem of code crashing with the null pointer exception errors that occur when code encounters a null value where one was not expected.

By default, a variable in Kotlin cannot have a null value assigned to it. Consider, for example, the following code:

```
val username: String = null
```

An attempt to compile the above code will result in a compilation error similar to the following:

```
Error: Null cannot be a value of a non-null string type String
```

If a variable is required to be able to store a null value, it must be specifically declared as a nullable type by placing a question mark (**?**) after the type declaration:

```
val username: String? = null
```

The *username* variable can now have a null value assigned to it without triggering a compiler error. Once a variable has been declared as nullable, a range of restrictions are then imposed on that variable by the compiler to prevent it being used in situations where it might cause a null pointer exception to occur. A nullable variable, cannot, for example, be assigned to a variable of non-null type as is the case in the following code:

```
val username: String? = null
val firstname: String = username
```

The above code will elicit the following error when encountered by the compiler:

```
Error: Type mismatch: inferred type is String? but String was expected
```

The only way that the assignment will be permitted is if some code is added to check that the value assigned to the nullable variable is non-null:

```
val username: String? = null

if (username != null) {
         val firstname: String = username
}
```

In the above case, the assignment will only take place if the username variable references a non-null value.

12.8 The safe call operator

A nullable variable also cannot be used to call a function or to access a property in the usual way. Earlier in this chapter, the *toUpperCase()* function was called on a String object. Given the possibility that this could cause a function to be called on a null reference, the following code will be disallowed by the compiler:

```
val username: String? = null
val uppercase = username.toUpperCase()
```

The exact error message generated by the compiler in this situation reads as follows:

```
Error: (Only safe (?.) or non-null asserted (!!.) calls are allowed on a nullable
receiver of type String?
```

In this instance, the compiler is essentially refusing to allow the function call to be made because no attempt has been made to verify that the variable is non-null. One way around this is to add some code to verify that something other than null value has been assigned to the variable before making the function call:

```
if (username != null) {
         val uppercase = username.toUpperCase()
}
```

A much more efficient way to achieve this same verification, however, is to call the function using the *safe call operator* (represented by *?.*) as follows:

```
val uppercase = username?.toUpperCase()
```

In the above example, if the username variable is null, the *toUpperCase()* function will not be called and execution will proceed at the next line of code. If, on the other hand, a non-null value is assigned the *toUpperCase()* function will be called and the result assigned to the *uppercase* variable.

In addition to function calls, the safe call operator may also be used when accessing properties:

```
val uppercase = username?.length
```

12.9 Not-null assertion

The *not-null assertion* removes all of the compiler restrictions from a nullable type, allowing it to be used in the same ways as a non-null type, even if it has been assigned a null value. This assertion is implemented using double exclamation marks after the variable name, for example:

```
val username: String? = null
val length = username!!.length
```

The above code will now compile, but will crash with the following exception at runtime since an attempt is being made to call a function on a non existent object:

```
Exception in thread "main" kotlin.KotlinNullPointerException
```

Clearly, this causes the very issue that nullable types are designed to avoid. Use of the not-null assertion is generally discouraged and should only be used in situations where you are certain that the value will not be null.

12.10 Nullable types and the let function

Earlier in this chapter, we looked at how the safe call operator can be used when making a call to a function belonging to a nullable type. This technique makes it easier to check if a value is null without having to write an *if* statement every time the variable is accessed. A similar problem occurs when passing a nullable type as an argument to a function which is expecting a non-null parameter. As an example, consider the *times()* function of the Int data type. When called on an Int object and passed another integer value as an argument, the function multiplies the two values and returns the result. When the following code is executed, for example, the value of 200 will be displayed within the console:

```
val firstNumber = 10
val secondNumber = 20

val result = firstNumber.times(secondNumber)
print(result)
```

The above example works because the secondNumber variable is a non-null type. A problem, however, occurs if

the secondNumber variable is declared as being of nullable type:

```
val firstNumber = 10
val secondNumber: Int? = 20

val result = firstNumber.times(secondNumber)
print(result)
```

Now the compilation will fail with the following error message because a nullable type is being passed to a function that is expecting a non-null parameter:

```
Error: Type mismatch: inferred type is Int? but Int was expected
```

A possible solution to this problem is to simply write an *if* statement to verify that the value assigned to the variable is non-null before making the call to the function:

```
val firstNumber = 10
val secondNumber: Int? = 20

if (secondNumber != null) {
    val result = firstNumber.times(secondNumber)
    print(result)
}
```

A more convenient approach to addressing the issue, however, involves use of the *let* function. When called on a nullable type object, the let function converts the nullable type to a non-null variable named *it* which may then be referenced within a lambda statement.

```
secondNumber?.let {
    val result = firstNumber.times(it)
    print(result)
}
```

Note the use of the safe call operator when calling the *let* function on secondVariable in the above example. This ensures that the function is only called when the variable is assigned a non-null value.

12.11 Late initialization (lateinit)

As previously outlined, non-null types need to be initialized when they are declared. This can be inconvenient if the value to be assigned to the non-null variable will not be known until later in the code execution. One way around this is to declare the variable using the *lateinit* modifier. This modifier designates that a value will be initialized with a value later. This has the advantage that a non-null type can be declared before it is initialized, with the disadvantage that the programmer is responsible for ensuring that the initialization has been performed before attempting to access the variable. Consider the following variable declaration:

```
var myName: String
```

Clearly, this is invalid since the variable is a non-null type but has not been assigned a value. Suppose, however, that the value to be assigned to the variable will not be known until later in the program execution. In this case, the lateinit modifier can be used as follows:

```
lateinit var myName: String
```

With the variable declared in this way, the value can be assigned later, for example:

```
myName = "John Smith"
print("My Name is " + myName)
```

Of course, if the variable is accessed before it is initialized, the code will fail with an exception:

```
lateinit var myName: String

print("My Name is " + myName)

Exception in thread "main" kotlin.UninitializedPropertyAccessException: lateinit
property myName has not been initialized
```

To verify whether a lateinit variable has been initialized, check the *isInitialized* property on the variable. To do this, we need to access the properties of the variable by prefixing the name with the '::' operator:

```
if (::myName.isInitialized) {
    print("My Name is " + myName)
}
```

12.12 The Elvis operator

The Kotlin Elvis operator can be used in conjunction with nullable types to define a default value that is to be returned if a value or expression result is null. The Elvis operator (?:) is used to separate two expressions. If the expression on the left does not resolve to a null value that value is returned, otherwise the result of the rightmost expression is returned. This can be thought of as a quick alternative to writing an if-else statement to check for a null value. Consider the following code:

```
if (myString != null) {
    return myString
} else {
    return "String is null"
}
```

The same result can be achieved with less coding using the Elvis operator as follows:

```
return myString ?: "String is null"
```

12.13 Type casting and type checking

When compiling Kotlin code, the compiler can typically infer the type of an object. Situations will occur, however, where the compiler is unable to identify the specific type. This is often the case when a value type is ambiguous or an unspecified object is returned from a function call. In this situation it may be necessary to let the compiler know the type of object that your code is expecting or to write code that checks whether the object is of a particular type.

Letting the compiler know the type of object that is expected is known as *type casting* and is achieved within Kotlin code using the *as* cast operator. The following code, for example, lets the compiler know that the result returned from the *getSystemService()* method needs to be treated as a KeyguardManager object:

```
val keyMgr = getSystemService(Context.KEYGUARD_SERVICE) as KeyguardManager
```

The Kotlin language includes both safe and unsafe cast operators. The above cast is an unsafe cast and will cause the app to throw an exception if the cast cannot be performed. A safe cast, on the other hand, uses the *as?* operator and returns null if the cast cannot be performed:

```
val keyMgr = getSystemService(Context.KEYGUARD_SERVICE) as? KeyguardManager
```

A type check can be performed to verify that an object conforms to a specific type using the *is* operator, for example:

```
if (keyMgr is KeyguardManager) {
```

```
    // It is a KeyguardManager object
}
```

12.14 Summary

This chapter has begun the introduction to Kotlin by exploring data types together with an overview of how to declare variables. The chapter has also introduced concepts such as nullable types, type casting and type checking and the Elvis operator, each of which is an integral part of Kotlin programming and designed specifically to make code writing less prone to error.

Chapter 18

18. An Overview of Compose

Now that Android Studio has been installed and the basics of the Kotlin programing language covered, it is time to start introducing Jetpack Compose.

Jetpack Compose is an entirely new approach to developing apps for all of Google's operating system platforms. The basic goals of Compose are to make app development easier, faster, and less prone to the types of bugs that typically appear when developing software projects. These elements have been combined with Compose-specific additions to Android Studio that allow Compose projects to be tested in near real-time using an interactive preview of the app during the development process.

Many of the advantages of Compose originate from the fact that it is both *declarative* and *data-driven*, topics which will be explained in this chapter.

The discussion in this chapter is intended as a high-level overview of Compose and does not cover the practical aspects of implementation within a project. Implementation and practical examples will be covered in detail in the remainder of the book.

18.1 Development before Compose

To understand the meaning and advantages of the Compose declarative syntax, it helps to understand how user interface layouts were designed before the introduction of Compose. Previously, Android apps were still built entirely using Android Studio together with a collection of associated frameworks that make up the Android Development Kit.

To aid in the design of the user interface layouts that make up the screens of an app, Android Studio includes a tool called the Layout Editor. The Layout Editor is a powerful tool that allows XML files to be created which contain the individual components that make up a screen of an app.

The user interface layout of a screen is designed within the Layout Editor by dragging components (such as buttons, text, text fields, and sliders) from a widget palette to the desired location on the layout canvas. Selecting a component in a scene provides access to a range of property panels where the attributes of the components can be changed.

The layout behavior of the screen (in other words how it reacts to different device screen sizes and changes to device orientation between portrait and landscape) is defined by configuring a range of constraints that dictate how each component is positioned and sized in relation to both the containing window and the other components in the layout.

Finally, any components that need to respond to user events (such as a button tap or slider motion) are connected to methods in the app source code where the event is handled.

At various points during this development process, it is necessary to compile and run the app on a simulator or device to test that everything is working as expected.

18.2 Compose declarative syntax

Compose introduces a declarative syntax that provides an entirely different way of implementing user interface layouts and behavior from the Layout Editor approach. Instead of manually designing the intricate details of the layout and appearance of components that make up a scene, Compose allows the scenes to be described using

a simple and intuitive syntax. In other words, Compose allows layouts to be created by declaring how the user interface should appear without having to worry about the complexity of how the layout is built.

This essentially involves declaring the components to be included in the layout, stating the kind of layout manager in which they are to be contained (column, row, box, list, etc.), and using modifiers to set attributes such as the text on a button, the foreground color of a label, or the handler to be called in the event of a tap gesture. Having made these declarations, all the intricate and complicated details of how to position, constrain and render the layout are handled automatically by Compose.

Compose declarations are structured hierarchically, which also makes it easy to create complex views by composing together small, re-usable custom sub-views.

While a layout is being declared and tested, Android Studio provides a preview canvas that changes in real-time to reflect the appearance of the layout. Android Studio also includes an *interactive preview* mode which allows the app to be launched within the preview canvas and fully tested without the need to build and run on a simulator or device.

Coverage of the Compose declaration syntax begins with the chapter entitled *"Composable Functions Overview"*.

18.3 Compose is data-driven

When we say that Compose is data-driven, this is not to say that it is no longer necessary to handle events generated by the user (in other words the interaction between the user and the app user interface). It is still necessary, for example, to know when the user taps a button or moves a slider and to react in some app-specific way. Being data-driven relates more to the relationship between the underlying app data and the user interface and logic of the app.

Before the introduction of Compose, an Android app would contain code responsible for checking the current values of data within the app. If data was likely to change over time, code had to be written to ensure that the user interface always reflected the latest state of the data (perhaps by writing code to frequently check for changes to the data, or by providing a refresh option for the user to request a data update). Similar challenges arise when keeping the user interface state consistent and making sure issues like toggle button settings are stored appropriately. Requirements such as these can become increasingly complex when multiple areas of an app depend on the same data sources.

Compose addresses this complexity by providing a system that is based on *state*. Data that is stored as state ensures that any changes to that data are automatically reflected in the user interface without the need to write any additional code to detect the change. Any user interface component that accesses a state is essentially *subscribed* to that state. When the state is changed anywhere in the app code, any subscriber components to that data will be destroyed and recreated to reflect the data change in a process called *recomposition*. This ensures that when any state on which the user interfaces is dependent changes, all components that rely on that data will automatically update to reflect the latest state.

State and recomposition will be covered in the chapter entitled *"An Overview of Compose State and Recomposition"*.

18.4 Summary

Jetpack introduces a different approach to app development than that offered by the Android Studio Layout Editor. Rather than directly implement the way in which a user interface is to be rendered, Compose allows the user interface to be declared in descriptive terms and then does all the work of deciding the best way to perform the rendering when the app runs.

Compose is also data-driven in that data changes drive the behavior and appearance of the app. This is achieved through states and recomposition.

This chapter has provided a very high-level view of Jetpack Compose. The remainder of this book will explore Compose in greater depth.

Chapter 25

25. Composing Layouts with Row and Column

User interface design is largely a matter of selecting the appropriate interface components, deciding how those views will be positioned on the screen, and then implementing navigation between the different screens of the app.

As is to be expected, Compose includes a wide range of user interface components for use when developing an app. Compose also provides a set of layout composables to define both how the user interface is organized and how the layout responds to factors such as changes in screen orientation and size.

This chapter will introduce the Row and Column composables included with Compose and explain how these can be used to create user interface designs with relative ease.

25.1 Creating the RowColDemo project

Launch Android Studio and select the New Project option from the welcome screen. Within the resulting new project dialog, choose the *Empty Compose Activity* template before clicking on the Next button.

Enter *RowColDemo* into the Name field and specify *com.example.rowcoldemo* as the package name. Before clicking on the Finish button, change the Minimum API level setting to API 26: Android 8.0 (Oreo). Edit the *Gradle Scripts -> build.gradle (Project: RowColDemo)* file, increase the *compose_ui_version* number from 1.1.1 to 1.2.1 (or the latest Compose 1.2 revision), and click on the Sync Now link.

Within the *MainActivity.kt* file, delete the Greeting function and add a new empty composable named MainScreen:

```
@Composable
fun MainScreen() {


}
```

Next, edit the *onCreateActivity()* method and DefaultPreview function to call MainScreen instead of Greeting. As we work through the examples in this chapter, row and column-based layouts will be built using instances of a custom component named TextCell which displays text within a black border with a small amount of padding to provide space between adjoining components. Before proceeding, add this function to the *MainActivity.kt* file as follows:

```
.
.
import androidx.compose.foundation.border
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.*
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.text.style.TextAlign
```

```
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
.
.
@Composable
fun TextCell(text: String, modifier: Modifier = Modifier) {

    val cellModifier = Modifier
        .padding(4.dp)
        .size(100.dp, 100.dp)
        .border(width = 4.dp, color = Color.Black)

    Text(text = text, cellModifier.then(modifier),
                fontSize = 70.sp,
                fontWeight = FontWeight.Bold,
                textAlign = TextAlign.Center)
}
```

25.2 Row composable

The Row composable, as the name suggests, lays out its children horizontally on the screen. For example, add a simple Row composable to the MainScreen function as follows:

```
.
.
@Composable
fun MainScreen() {
    Row {
        TextCell("1")
        TextCell("2")
        TextCell("3")
    }
}
```

When rendered, the Row declared above will appear as illustrated in Figure 25-1 below:

Figure 25-1

25.3 Column composable

The Column composable performs the same purpose as the Row with the exception that its children are arranged vertically. The following example places the same three composables within a Column:

```
.
.
@Composable
fun MainScreen() {
```

```
    Column {
        TextCell("1")
        TextCell("2")
        TextCell("3")
    }
}
```

The rendered output from the code will appear as shown in Figure 25-2:

Figure 25-2

25.4 Combining Row and Column composables

Row and Column composables can, of course, be embedded within each other to create table style layouts. Try, for example, the following composition containing a mixture of embedded Row and Column layouts:

```
@Composable
fun MainScreen() {
    Column {
        Row {
            Column {
                TextCell("1")
                TextCell("2")
                TextCell("3")
            }

            Column {
                TextCell("4")
                TextCell("5")
                TextCell("6")
            }

            Column {
                TextCell("7")
                TextCell("8")
            }
        }

        Row {
```

```
            TextCell("9")
            TextCell("10")
            TextCell("11")
        }
    }
}
```

Figure 25-3 illustrates the layout generated by the above code:

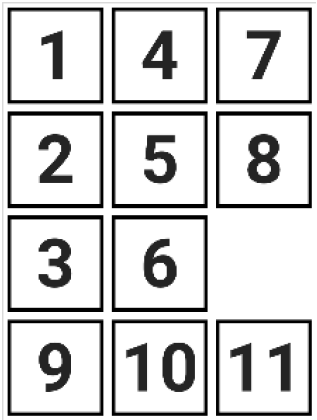

Figure 25-3

Using this technique, Row and Column layouts may be embedded within each other to achieve just about any level of layout complexity.

25.5 Layout alignment

Both the Row and Column composables will occupy an area of space within the user interface layout depending on child elements, other composables, and any size-related modifiers that may have been applied. By default, the group of child elements within a Row or Column will be aligned with the top left-hand corner of the content area (assuming the app is running on a device configured with a left-to-right reading locale). We can see this effect if we increase the size of our original example Row composable:

```
@Composable
fun MainScreen() {
    Row(modifier = Modifier.size(width = 400.dp, height = 200.dp)) {
        TextCell("1")
        TextCell("2")
        TextCell("3")
    }
}
```

Before making this change, the Row was *wrapping* its children (in other words sizing itself to match the content). Now that the Row is larger than the content we can see that the default alignment has placed the children in the top left-hand corner of the Row component:

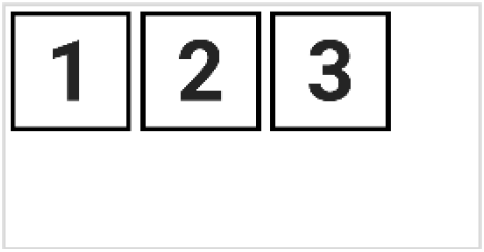

Figure 25-4

This default alignment in the vertical axis may be changed by passing through a new value using the *verticalAlignment* parameter of the Row composable. For example, to position the children in the vertical center of the available space, the *Alignment.CenterVertically* value would be passed to the Row as follows:

```
.
.
import androidx.compose.ui.Alignment
.
.
@Composable
fun MainScreen() {
    Row(verticalAlignment = Alignment.CenterVertically,
        modifier = Modifier.size(width = 400.dp, height = 200.dp)) {
        TextCell("1")
        TextCell("2")
        TextCell("3")
    }
}
```

This will cause the content to be positioned in the vertical center of the Row's area as illustrated below:

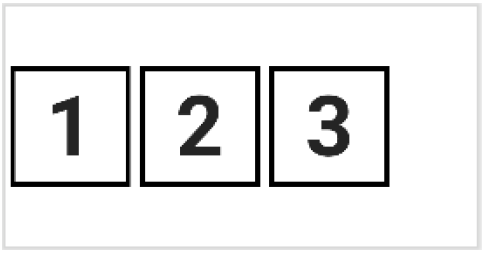

Figure 25-5

The following is a list of alignment values accepted by the Row vertical alignment parameter:

- **Alignment.Top** - Aligns the content at the top of the Row content area.
- **Alignment.CenterVertically** - Positions the content in the vertical center of the Row content area.
- **Alignment.Bottom** - Aligns the content at the bottom of the Row content area.

When working with the Column composable, the *horizontalAlignment* parameter is used to configure alignment along the horizontal axis. Acceptable values are as follows:

- **Alignment.Start** - Aligns the content at the horizontal start of the Column content area.
- **Alignment.CenterHorizontally** - Positions the content in the horizontal center of the Column content area
- **Alignment.End** - Aligns the content at the horizontal end of the Column content area.

In the following example, the Column's children have been aligned with the end of the Column content area:

```
.
.
@Composable
fun MainScreen() {
    Column(horizontalAlignment = Alignment.End,
           modifier = Modifier.width(250.dp)) {
        TextCell("1")
        TextCell("2")
        TextCell("3")
    }
}
```

When rendered, the resulting column will appear as shown in Figure 25-6:

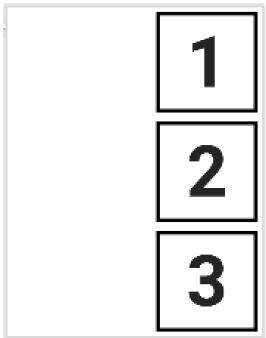

Figure 25-6

When working with alignment it is worth remembering that it works on the opposite axis to the flow of the containing composable. For example, while the Row organizes children horizontally, alignment operates on the vertical axis. Conversely, alignment operates on the horizontal axis for the Column composable while children are arranged vertically. The reason for emphasizing this point will become evident when we introduce arrangements.

25.6 Layout arrangement positioning

Unlike the alignment settings, arrangement controls child positioning along the same axis as the container (i.e. horizontally for Rows and vertically for Columns). Arrangement values are set on Row and Column instances using the *horizontalArrangement* and *verticalArrangement* parameters respectively. Arrangement properties can be categorized as influencing either position or child spacing.

The following positional settings are available for the Row component via the *horizontalArrangement* parameter:

- **Arrangement.Start** - Aligns the content at the horizontal start of the Row content area.
- **Arrangement.Center** - Positions the content in the horizontal center of the Row content area.

- **Arrangement.End** - Aligns the content at the horizontal end of the Row content area.

The above settings can be visualized as shown in Figure 25-7:

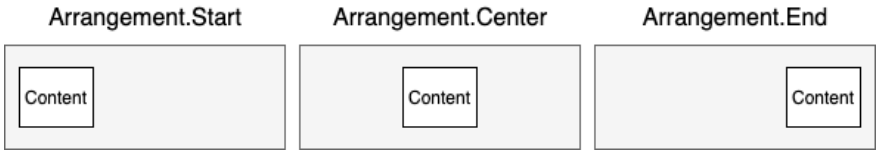

Figure 25-7

The Column composable, on the other hand, accepts the following values for the *verticalArrangement* parameter:

- **Arrangement.Top** - Aligns the content at the top of the Column content area.
- **Arrangement.Center** - Positions the content in the vertical center of the Column content area.
- **Arrangement.Bottom** - Aligns the content at the bottom of the Column content area.

Figure 25-8 illustrates these *verticalArrangement* settings:

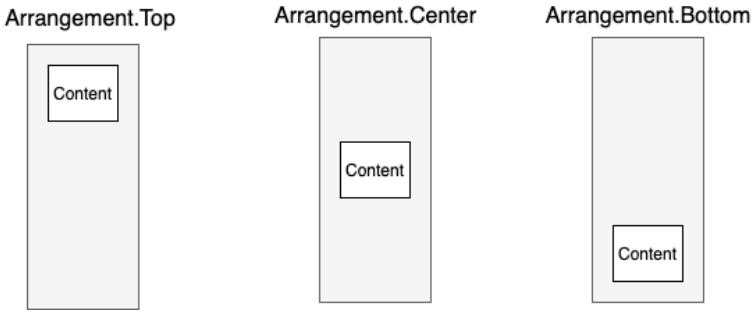

Figure 25-8

Using our example once again, the following change moves the child elements to the end of the Row content area:

```
Row(horizontalArrangement = Arrangement.End,
        modifier = Modifier.size(width = 400.dp, height = 200.dp)) {
        TextCell("1")
        TextCell("2")
        TextCell("3")
}
```

The above code will generate the following user interface layout:

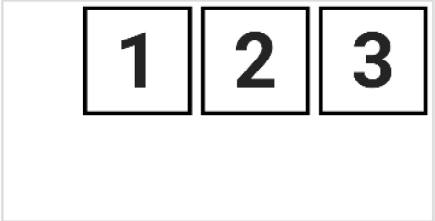

Figure 25-9

Similarly, the following positions child elements at the bottom of the containing Column:

```
Column(verticalArrangement = Arrangement.Bottom,
        modifier = Modifier.height(400.dp)) {
    TextCell("1")
    TextCell("2")
    TextCell("3")
}
```

The above composable will render within the Preview panel as illustrated in Figure 25-10 below:

Figure 25-10

25.7 Layout arrangement spacing

Arrangement spacing controls how the child components in a Row or Column are spaced across the content area. These settings are still defined using the *horizontalArrangement* and *verticalArrangement* parameters, but require one of the following values:

- **Arrangement.SpaceEvenly** - Children are spaced equally, including space before the first and after the last child.
- **Arrangement.SpaceBetween** - Children are spaced equally, with no space allocation before the first and after the last child.
- **Arrangement.SpaceAround** - Children are spaced equally, including half spacing before the first and after the last child.

In the following declaration, the children of a Row are positioned using the SpaceEvenly setting:

```
Row(horizontalArrangement = Arrangement.SpaceEvenly,
                    modifier = Modifier.width(1000.dp)) {
        TextCell("1")
        TextCell("2")
        TextCell("3")
}
```

The above code gives us the following layout with equal gaps at the beginning and end of the row and between each child:

Figure 25-11

Figure 25-12, on the other hand, shows the same row configured with the SpaceBetween setting. Note that the row has no leading or trailing spacing:

Figure 25-12

Finally, Figure 25-13 shows the effect of applying the SpaceAround setting which adds full spacing between children and half the spacing on the leading and trailing ends:

Figure 25-13

25.8 Row and Column scope modifiers

The children of a Row or Column are said to be within the *scope* of the parent. These two scopes (RowScope and ColumnScope) provide a set of additional modifier functions that can be applied to change the behavior and appearance of individual children within a Row or Column. The Android Studio code editor provides a visual indicator when children are within a scope. In Figure 25-14, for example, the editor indicates that the RowScope modifier functions are available to the three child composables:

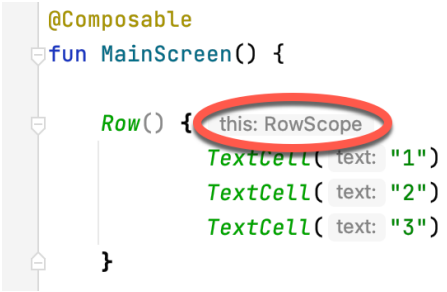

Figure 25-14

When working with the Column composable, a similar ColumnScope indicator will appear.

ColumnScope includes the following modifiers for controlling the position of child components:

- **Modifier.align()** - Allows the child to be aligned horizontally using *Alignment.CenterHorizontally*, *Alignment.Start*, and *Alignment.End* values.
- **Modifier.alignBy()** - Aligns a child horizontally with other siblings on which the *alignBy()* modifier has also been applied.
- **Modifier.weight()** - Sets the height of the child relative to the weight values assigned to its siblings.

RowScope provides the following additional modifier functions to Row children:

- **Modifier.align()** - Allows the child to be aligned vertically using *Alignment.CenterVertically*, *Alignment.Top,* and *Alignment.Bottom* values.
- **Modifier.alignBy()** - Aligns a child with other siblings on which the alignBy() modifier has also been applied. Alignment may be performed by baseline or using custom alignment line configurations.
- **Modifier.alignByBaseline()** - Aligns the baseline of a child with any siblings that have also been configured by either the *alignBy()* or *alignByBaseline()* modifier.
- **Modifier.paddingFrom()** - Allows padding to be added to the alignment line of a child.
- **Modifier.weight()** - Sets the width of the child relative to the weight values assigned to its siblings.

The following Row declaration, for example, sets different alignments on each of the three TextCell children:

```
Row(modifier = Modifier.height(300.dp)) {
    TextCell("1", Modifier.align(Alignment.Top))
    TextCell("2", Modifier.align(Alignment.CenterVertically))
    TextCell("3", Modifier.align(Alignment.Bottom))
}
```

When previewed, this will generate a layout resembling Figure 25-15:

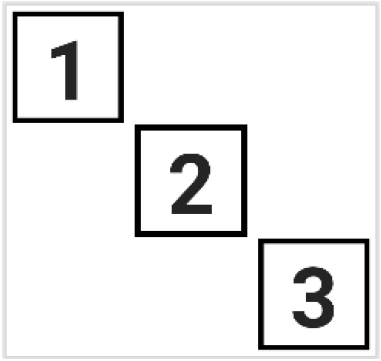

Figure 25-15

The baseline alignment options are especially useful for aligning text content with differing font sizes. Consider, for example, the following Row configuration:

```
Row {
  Text(
      text = "Large Text",
      fontSize = 40.sp,
      fontWeight = FontWeight.Bold
  )
  Text(
      text = "Small Text",
      fontSize = 32.sp,
      fontWeight = FontWeight.Bold
  )
}
```

This code consists of a Row containing two Text composables, each using a different font size resulting in the following layout:

Figure 25-16

The Row has aligned the two Text composables along their top edges causing the text content to be out of alignment relative to the text baselines. To resolve this problem we can apply the *alignByBaseline()* modifier to both children as follows:

```
Row {
    Text(
      text = "Large Text",
      Modifier.alignByBaseline(),
      fontSize = 40.sp,
      fontWeight = FontWeight.Bold
    )
    Text(
      text = "Small Text",
      Modifier.alignByBaseline(),
      fontSize = 32.sp,
      fontWeight = FontWeight.Bold,
    )
}
```

Now when the layout is rendered, the baselines of the two Text composables will be aligned as illustrated in Figure 25-17:

Figure 25-17

As an alternative, the *alignByBaseline()* modifier may be replaced by a call to the *alignBy()* function, passing through FirstBaseline as the alignment parameter:

```
Modifier.alignBy(FirstBaseline)
```

When working with multi-line text, passing LastBaseline through to the *alignBy()* modifier function will cause appropriately configured sibling components to align with the baseline of the last line of text:

```
.
.
import androidx.compose.ui.layout.LastBaseline
.
.
@Composable
fun MainScreen() {
    Row {
```

```
        Text(
            text = "Large Text\nMore Text",
            Modifier.alignBy(LastBaseline),
            fontSize = 40.sp,
            fontWeight = FontWeight.Bold
        )
        Text(
            text = "Small Text",
            Modifier.alignByBaseline(),
            fontSize = 32.sp,
            fontWeight = FontWeight.Bold,
        )
    }
}
```

Now when the layout appears the baseline of the text content of the second child will align with the baseline of the last line of text in the first child:

Figure 25-18

Using the FirstBaseline in the above example would, of course, align the baseline of the small text composable with the baseline of the first line of text in the multi-line component:

Figure 25-19

In the examples we have looked at so far we have specified the baseline as the alignment line for both children. If we need the alignment to be offset for a child, we can do so using the *paddingFrom()* modifier. The following example adds an additional 80dp vertical offset to the first baseline alignment position of the small text composable:

```
.
.
import androidx.compose.ui.layout.FirstBaseline
.
.
@Composable
fun MainScreen() {
    Row {
```

```
        Text(
            text = "Large Text\nMore Text",
            Modifier.alignBy(FirstBaseline),
            fontSize = 40.sp,
            fontWeight = FontWeight.Bold
        )
        Text(
            text = "Small Text",
            modifier = Modifier.paddingFrom(
                alignmentLine = FirstBaseline, before = 80.dp, after = 0.dp),
            fontSize = 32.sp,
            fontWeight = FontWeight.Bold
        )
    }
}
```

When rendered, the above layout will appear as shown in Figure 25-20:

Figure 25-20

25.9 Scope modifier weights

The RowScope weight modifier allows the width of each child to be specified relative to its siblings. This works by assigning each child a weight percentage (between 0.0 and 1.0). Two children assigned a weight of 0.5, for example, would each occupy half of the available space. Modify the MainScreen function one last time as follows to demonstrate the use of the weight modifier:

```
@Composable
fun MainScreen() {
    Row {
        TextCell("1", Modifier.weight(weight = 0.2f, fill = true))
        TextCell("2", Modifier.weight(weight = 0.4f, fill = true))
        TextCell("3", Modifier.weight(weight = 0.3f, fill = true))
    }
}
```

Rebuild and refresh the preview panel, at which point the layout should resemble that shown in Figure 25-21 below:

Figure 25-21

Siblings that do not have a weight modifier applied will appear at their preferred size leaving the weighted children to share the remaining space.

ColumnScope also provides *align()*, *alignBy()*, and *weight()* modifiers, though these all operate on the horizontal axis. Unlike RowScope, there is no concept of baselines when working with ColumnScope.

25.10 Summary

The Compose Row and Column components provide an easy way to layout child composables in horizontal and vertical arrangements. When embedded within each other, the Row and Column allow table style layouts of any level of complexity to be created. Both layout components include options for customizing the alignment, spacing, and positioning of children. Scope modifiers allow the positioning, and sizing behavior of individual children to be defined, including aligning and sizing children relative to each other.

Chapter 33

33. An Overview of Lists and Grids in Compose

It is a common requirement when designing user interface layouts to present information in either scrollable list or grid configurations. For basic list requirements, the Row and Column components can be re-purposed to provide vertical and horizontal lists of child composables. Extremely large lists, however, are likely to cause degraded performance if rendered using the standard Row and Column composables. For lists containing large numbers of items, Compose provides the LazyColumn and LazyRow composables. Similarly, grid-based layouts can be presented using the LazyVerticalGrid composable.

This chapter will introduce the basics of list and grid creation and management in Compose in preparation for the tutorials in subsequent chapters.

33.1 Standard vs. lazy lists

Part of the popularity of lists is that they provide an effective way to present large amounts of items in a scrollable format. Each item in a list is represented by a composable which may, itself, contain descendant composables. When a list is created using the Row or Column component, all of the items it contains are also created at initialization, regardless of how many are visible at any given time. While this does not necessarily pose a problem for smaller lists, it can be an issue for lists containing many items.

Consider, for example, a list that is required to display 1000 photo images. It can be assumed with a reasonable degree of certainty that only a small percentage of items will be visible to the user at any one time. If the application was permitted to create each of the 1000 items in advance, however, the device would very quickly run into memory and performance limitations.

When working with longer lists, the recommended course of action is to use LazyColumn, LazyRow, and LazyVerticalGrid. These components only create those items that are visible to the user. As the user scrolls, items that move out of the viewable area are destroyed to free up resources while those entering view are created just in time to be displayed. This allows lists of potentially infinite length to be displayed with no performance degradation.

Since there are differences in approach and features when working with Row and Column compared to the lazy equivalents, this chapter will provide an overview of both types.

33.2 Working with Column and Row lists

Although lacking some of the features and performance advantages of the LazyColumn and LazyRow, the Row and Column composables provide a good option for displaying shorter, basic lists of items. Lists are declared in much the same way as regular rows and columns with the exception that each list item is usually generated programmatically. The following declaration, for example, uses the Column component to create a vertical list containing 100 instances of a composable named MyListItem:

```
Column {
    repeat(100) {
        MyListItem()
    }
```

```
}
```

Similarly, the following example creates a horizontal list containing the same items:

```
Row {
    repeat(100) {
        MyListItem()
    }
}
```

The MyListItem composable can be anything from a single Text composable to a complex layout containing multiple composables.

33.3 Creating lazy lists

Lazy lists are created using the LazyColumn and LazyRow composables. These layouts place children within a LazyListScope block which provides additional features for managing and customizing the list items. For example, individual items may be added to a lazy list via calls to the *item()* function of the LazyListScope:

```
LazyColumn {
    item {
        MyListItem()
    }
}
```

Alternatively, multiple items may be added in a single statement by calling the *items()* function:

```
LazyColumn {
    items(1000) { index ->
        Text("This is item $index");
    }
}
```

LazyListScope also provides the *itemsIndexed()* function which associates the item content with an index value, for example:

```
val colorNamesList = listOf("Red", "Green", "Blue", "Indigo")

LazyColumn {
    itemsIndexed(colorNamesList) { index, item ->
        Text("$index = $item")
    }
}
```

When rendered, the above lazy column will appear as shown in Figure 33-1 below:

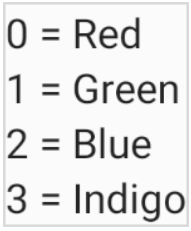

Figure 33-1

Lazy lists also support the addition of headers to groups of items in a list using the *stickyHeader()* function. This topic will be covered in more detail later in the chapter.

33.4 Enabling scrolling with ScrollState

While the above Column and Row list examples will display a list of items, only those that fit into the viewable screen area will be accessible to the user. This is because lists are not scrollable by default. To make Row and Column-based lists scrollable, some additional steps are needed. LazyList and LazyRow, on the other hand, support scrolling by default.

The first step in enabling list scrolling when working with Row and Column-based lists is to create a ScrollState instance. This is a special state object designed to allow Row and Column parents to remember the current scroll position through recompositions. A ScrollState instance is generated via a call to the *rememberScrollState()* function, for example:

```
val scrollState = rememberScrollState()
```

Once created, the scroll state is passed as a parameter to the Column or Row composable using the *verticalScroll()* and *horizontalScroll()* modifiers. In the following example, vertical scrolling is being enabled in a Column list:

```
Column(Modifier.verticalScroll(scrollState)) {
    repeat(100) {
        MyListItem()
    }
}
```

Similarly, the following code enables horizontal scrolling on a LazyRow list:

```
Row(Modifier.horizontalScroll(scrollState))  {
    repeat(1000) {
        MyListItem()
    }
}
```

33.5 Programmatic scrolling

We generally think of scrolling as being something a user performs through dragging or swiping gestures on the device screen. It is also important to know how to change the current scroll position from within code. An app screen might, for example, contain buttons which can be tapped to scroll to the start and end of a list. The steps to implement this behavior differ between Row and Columns lists and the lazy list equivalents.

When working with Row and Column lists, programmatic scrolling can be performed by calling the following functions on the ScrollState instance:

- **animateScrollTo(value: Int)** - Scrolls smoothly to the specified pixel position in the list using animation.
- **scrollTo(value: Int)** - Scrolls instantly to the specified pixel position.

Note that the value parameters in the above function represent the list position in pixels instead of referencing a specific item number. It is safe to assume that the start of the list is represented by pixel position 0, but the pixel position representing the end of the list may be less obvious. Fortunately, the maximum scroll position can be identified by accessing the *maxValue* property of the scroll state instance:

```
val maxScrollPosition = scrollState.maxValue
```

To programmatically scroll LazyColumn and LazyRow lists, functions need to be called on a LazyListState instance which can be obtained via a call to the *rememberLazyListState()* function as follows:

```
val listState = rememberLazyListState()
```

Once the list state has been obtained, it must be applied to the LazyRow or LazyColumn declaration as follows:

```
.
.
LazyColumn(
    state = listState,
{
.
.
```

Scrolling can then be performed via calls to the following functions on the list state instance:

- **animateScrollToItem(index: Int)** - Scrolls smoothly to the specified list item (where 0 is the first item).
- **scrollToItem(index: Int)** - Scrolls instantly to the specified list item (where 0 is the first item).

In this case, the scrolling position is referenced by the index of the item instead of pixel position.

One complication is that all four of the above scroll functions are *coroutine* functions. As outlined in the chapter titled *"Coroutines and LaunchedEffects in Jetpack Compose"*, coroutines are a feature of Kotlin that allows blocks of code to execute asynchronously without blocking the thread from which they are launched (in this case the *main thread* which is responsible for making sure the app remains responsive to the user). Coroutines can be implemented without having to worry about building complex 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. One of the key requirements of coroutine functions is that they must be launched from within a *coroutine scope*.

As with ScrollState and LazyListState, we need access to a CoroutineScope instance that will be remembered through recompositions. This requires a call to the *rememberCoroutineScope()* function as follows:

```
val coroutineScope = rememberCoroutineScope()
```

Once we have a coroutine scope, we can use it to launch the scroll functions. The following code, for example, declares a Button component configured to launch the *animateScrollTo()* function within the coroutine scope. In this case, the button will cause the list to scroll to the end position when clicked:

```
.
.
Button(onClick = {
    coroutineScope.launch {
        scrollState.animateScrollTo(scrollState.maxValue)
    }
.
.
}
```

33.6 Sticky headers

Sticky headers is a feature only available within lazy lists that allows list items to be grouped under a corresponding header. Sticky headers are created using the LazyListScope *stickyHeader()* function.

The headers are referred to as being sticky because they remain visible on the screen while the current group is scrolling. Once a group scrolls from view, the header for the next group takes its place. Figure 33-2, for example,

shows a list with sticky headers. Note that although the Apple group is scrolled partially out of view, the header remains in position at the top of the screen:

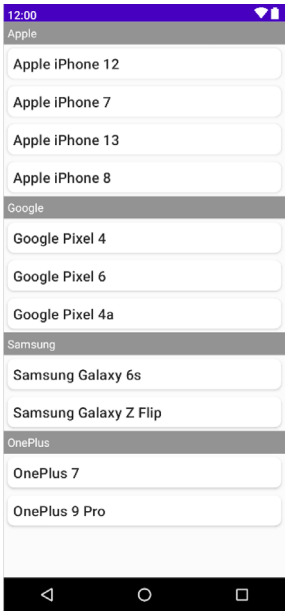

Figure 33-2

When working with sticky headers, the list content must be stored in an Array or List which has been mapped using the Kotlin *groupBy()* function. The *groupBy()* function accepts a lambda which is used to define the *selector* which defines how data is to be grouped. This selector then serves as the key to access the elements of each group. Consider, for example, the following list which contains mobile phone models:

```
val phones = listOf("Apple iPhone 12", "Google Pixel 4", "Google Pixel 6",
    "Samsung Galaxy 6s", "Apple iPhone 7", "OnePlus 7", "OnePlus 9 Pro",
         "Apple iPhone 13", "Samsung Galaxy Z Flip", "Google Pixel 4a",
            "Apple iPhone 8")
```

Now suppose that we want to group the phone models by manufacturer. To do this we would use the first word of each string (in other words, the text before the first space character) as the selector when calling *groupBy()* to map the list:

```
val groupedPhones = phones.groupBy { it.substringBefore(' ') }
```

Once the phones have been grouped by manufacturer, we can use the *forEach* statement to create a sticky header for each manufacture name, and display the phones in the corresponding group as list items:

```
groupedPhones.forEach { (manufacturer, models) ->
    stickyHeader {
        Text(
            text = manufacturer,
            color = Color.White,
            modifier = Modifier
                .background(Color.Gray)
                .padding(5.dp)
                .fillMaxWidth()
```

```
        )
    }

    items(models) { model ->
        MyListItem(model)
    }
}
```

In the above *forEach* lambda, *manufacturer* represents the selector key (for example "Apple") and *models* an array containing the items in the corresponding manufacturer group ("Apple iPhone 12", "Apple iPhone 7", and so on for the Apple selector):

```
groupedPhones.forEach { (manufacturer, models) ->
```

The selector key is then used as the text for the sticky header, and the *models* list is passed to the *items()* function to display all the group elements, in this case using a custom composable named MyListItem for each item:

```
items(models) { model ->
    MyListItem(model)
}
```

When rendered, the above code will display the list shown in Figure 33-2 above.

33.7 Responding to scroll position

Both LazyRow and LazyColumn allow actions to be performed when a list scrolls to a specified item position. This can be particularly useful for displaying a "scroll to top" button that appears only when the user scrolls towards the end of the list.

The behavior is implemented by accessing the *firstVisibleItemIndex* property of the LazyListState instance which contains the index of the item that is currently the first visible item in the list. For example, if the user scrolls a LazyColumn list such that the third item in the list is currently the topmost visible item, *firstVisibleItemIndex* will contain a value of 2 (since indexes start counting at 0). The following code, for example, could be used to display a "scroll to top" button when the first visible item index exceeds 8:

```
val firstVisible = listState.firstVisibleItemIndex

if (firstVisible > 8) {
    // Display scroll to top button
}
```

33.8 Creating a lazy grid

Grid layouts may be created using the LazyVerticalGrid composable. The appearance of the grid is controlled by the *cells* parameter that can be set to either *adaptive* or *fixed* mode. In adaptive mode, the grid will calculate the number of rows and columns that will fit into the available space, with even spacing between items and subject to a minimum specified cell size. Fixed mode, on the other hand, is passed the number of rows to be displayed and sizes each column width equally to fill the width of the available space.

The following code, for example, declares a grid containing 30 cells, each with a minimum width of 60dp:

```
LazyVerticalGrid(
    cells = GridCells.Adaptive(minSize = 60.dp),
    state = rememberLazyListState(),
    contentPadding = PaddingValues(10.dp)
```

```
) {
        items(30) { index ->
            Card(backgroundColor = Color.Blue,
                modifier = Modifier.padding(5.dp).fillMaxSize()) {
                Text(
                     "$index",
                     fontSize = 35.sp,
                     color = Color.White,
                     textAlign = TextAlign.Center)
            }
        }
}
```

When called, the LazyVerticalGrid composable will fit as many items as possible into each row without making the column width greater than 60dp as illustrated in the figure below:

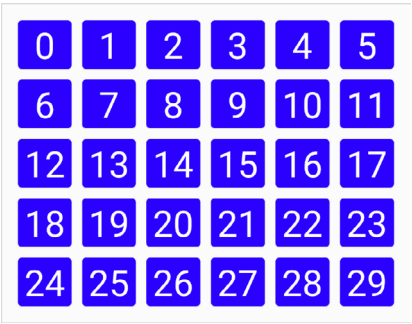

Figure 33-3

The following code organizes items in a grid containing three columns:

```
LazyVerticalGrid(
    cells = GridCells.Fixed(3),
    state = rememberLazyListState(),
    contentPadding = PaddingValues(10.dp)
) {

        items(15) { index ->
            Card(backgroundColor = Color.Blue,
                modifier = Modifier.padding(5.dp).fillMaxSize()) {
                Text(
                    "$index",
                    fontSize = 35.sp,
                    color = Color.White,
                    textAlign = TextAlign.Center)
            }
        }
}
```

The layout from the above code will appear as illustrated in Figure 33-4 below:

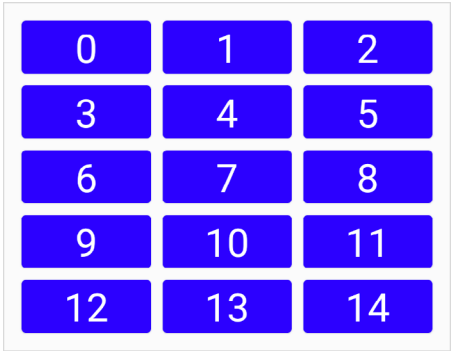

Figure 33-4

Both the above grid examples used a Card composable containing a Text component for each cell item. The Card component provides a surface into which to group content and actions relating to a single content topic and is often used as the basis for list items. Although we provided a Text composable as the child, the content in a card can be any composable, including containers such as Row, Column, and Box layouts. A key feature of Card is the ability to create a shadow effect by specifying an elevation:

```
Card(
    modifier = Modifier
        .fillMaxWidth()
        .padding(15.dp),
    elevation = 10.dp
) {
    Column(horizontalAlignment = Alignment.CenterHorizontally,
        modifier = Modifier.padding(15.dp)
    ) {
        Text("Jetpack Compose", fontSize = 30.sp)
        Text("Card Example", fontSize = 20.sp)
    }
}
```

When rendered, the above Card component will appear as shown in Figure 33-5:

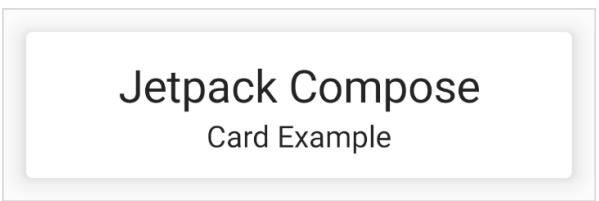

Figure 33-5

33.9 Summary

Lists in Compose may be created using either standard or lazy list components. The lazy components have the advantage that they can present large amounts of content without impacting the performance of the app or the device on which it is running. This is achieved by creating list items only when they become visible and destroying them as they scroll out of view. Lists can be presented in row, column, and grid formats and can be static or scrollable. It is also possible to programmatically scroll lists to specific positions and to trigger events based on the current scroll position.