Jetpack Compose 1.7 Essentials

Payload publishing

Jetpack Compose 1.7 Essentials

Jetpack Compose 1.7 Essentials

ISBN-13: 978-1-965764-03-9

Rev: 1.0

https://www.payloadbooks.com

Contents

Table of Contents

Chapter 1

1. Start Here

Welcome to this comprehensive guide to building Android applications using Jetpack Compose 1.7, Android Studio Ladybug, Material Design 3, and the Kotlin programming language. This book has been designed to teach developers the essential knowledge and skills to create modern, dynamic, and visually compelling applications.

We begin with foundational steps, including the setup of your development environment in Android Studio, followed by a detailed introduction to Kotlin, the language underpinning Android development. This section covers core aspects of Kotlin, such as data types, operators, control flow, functions, lambdas, and coroutines, establishing a strong basis in object-oriented programming principles.

With this foundation in place, we turn to Jetpack Compose, Google's innovative toolkit for building native user interfaces. This book presents an in-depth exploration of Compose components and layout structures, including rows, columns, boxes, flows, pagers, and lists, alongside an overview of Android project architecture and Android Studio's Compose development mode.

The following sections delve into more advanced topics, such as state management, modifiers, and navigation components—key elements in designing smooth and intuitive user interfaces. We also guide you through creating reusable layout components, securing applications with biometric authentication, and incorporating Google Maps functionality to enrich user engagement.

Furthermore, we cover specialized techniques, such as graphics rendering, animations, transitions, Kotlin Flows, and gesture handling. Practical data management solutions, including view models, Room database access, live data, and the Database Inspector, are discussed in depth. For developers interested in monetization, this guide also includes a dedicated section on implementing in-app billing.

The concluding chapters provide a comprehensive overview of app packaging and the publication process on the Google Play Store. Throughout the book, each concept is solidified through detailed, hands-on tutorials, complemented by downloadable source code to facilitate real-world application and access to over 55 online knowledge test quizzes.

With a basic understanding of programming, access to Android Studio and the Android SDK, and a Windows, Mac, or Linux system, you will be well-prepared to embark on this journey. It is our aim that, upon completion, you will be equipped with the skills necessary to design, build, and deploy professional-grade Android applications.

1.1 For Kotlin programmers

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

1.2 For new Kotlin programmers

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

1.3 Downloading the code samples

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

https://www.payloadbooks.com/product/compose17/

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

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

1.4 Feedback

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

1.5 Errata

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

https://www.payloadbooks.com/compose17_errata

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

1.6 Take the knowledge tests

Look for this section at the end of most chapters and use the link or scan the QR code to take a knowledge quiz to test and reinforce your understanding of the covered topic. Use the following link to review the full list of tests available for this book:

https://www.answertopia.com/p9jf

Chapter 2

2. Setting up an Android Studio Development Environment

Before any work can begin on developing an Android application, the first step is to configure a computer system to act as the development platform. This involves several steps consisting of installing the Android Studio Integrated Development Environment (IDE), including the Android Software Development Kit (SDK), the Kotlin plug-in and the OpenJDK Java development environment.

This chapter will cover the steps necessary to install the requisite components for Android application development on Windows, macOS, and Linux-based systems.

2.1 System requirements

Android application development may be performed on any of the following system types:

- Windows 8/10/11 64-bit
- macOS 10.14 or later running on Intel or Apple silicon
- Chrome OS device with Intel i5 or higher
- Linux systems with version 2.31 or later of the GNU C Library (glibc)
- Minimum of 8GB of RAM
- Approximately 8GB of available disk space
- 1280 x 800 minimum screen resolution

2.2 Downloading the Android Studio package

Most of the work involved in developing applications for Android will be performed using the Android Studio environment. The content and examples in this book were created based on Android Studio Ladybug 2024.2.1 using the Android API 35 SDK (VanillaIceCream), which, at the time of writing, are the latest stable releases.

Android Studio is, however, subject to frequent updates, so a newer version may have been released since this book was published.

The latest release of Android Studio may be downloaded from the primary download page, which can be found at the following URL:

https://developer.android.com/studio/index.html

If this page provides instructions for downloading a newer version of Android Studio, there may be differences between this book and the software. A web search for "Android Studio Ladybug" should provide the option to download the older version if these differences become a problem. Alternatively, visit the following web page to find Android Studio Ladybug 2024.2.1 in the archives:

https://developer.android.com/studio/archive

2.3 Installing Android Studio

Once downloaded, the exact steps to install Android Studio differ depending on the operating system on which the installation is performed.

2.3.1 Installation on Windows

Locate the downloaded Android Studio installation executable file (named *android-studio-<version>-windows.exe*) in a Windows Explorer window and double-click on it to start the installation process, clicking the *Yes* button in the User Account Control dialog if it appears.

Once the Android Studio setup wizard appears, work through the various screens to configure the installation to meet your requirements in terms of the file system location into which Android Studio should be installed and whether or not it should be made available to other system users. When prompted to select the components to install, ensure that the *Android Studio* and *Android Virtual Device* options are all selected.

Although there are no strict rules on where Android Studio should be installed on the system, the remainder of this book will assume that the installation was performed into *C:\Program Files\Android\Android Studio* and that the Android SDK packages have been installed into the user's *AppData\Local\Android\sdk* sub-folder. Once the options have been configured, click the *Install* button to begin the installation process.

On versions of Windows with a Start menu, the newly installed Android Studio can be launched from the entry added to that menu during the installation. The executable may be pinned to the taskbar for easy access by navigating to the *Android Studio\bin* directory, right-clicking on the *studio64* executable, and selecting the *Pin to Taskbar* menu option (on Windows 11, this option can be found by selecting *Show more options* from the menu).

2.3.2 Installation on macOS

Android Studio for macOS is downloaded as a disk image (.dmg) file. Once the *android-studio-<version>-mac.dmg* file has been downloaded, locate it in a Finder window and double-click on it to open it, as shown in Figure 2-1:

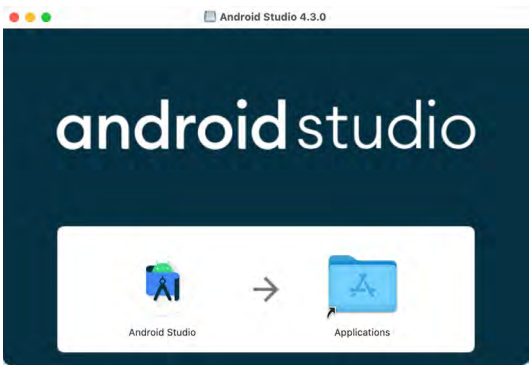

Figure 2-1

To install the package, drag the Android Studio icon and drop it onto the Applications folder. The Android Studio package will then be installed into the Applications folder of the system, a process that will typically take a few seconds to complete.

To launch Android Studio, locate the executable in the Applications folder using a Finder window and double-click on it.

For future, easier access to the tool, drag the Android Studio icon from the Finder window and drop it onto the dock.

2.3.3 Installation on Linux

Having downloaded the Linux Android Studio package, open a terminal window, change directory to the location where Android Studio is to be installed, and execute the following command:

```
tar xvfz /<path to package>/android-studio-<version>-linux.tar.gz
```

Note that the Android Studio bundle will be installed into a subdirectory named *android-studio*. Therefore, assuming that the above command was executed in */home/demo*, the software packages will be unpacked into */home/demo/android-studio*.

To launch Android Studio, open a terminal window, change directory to the *android-studio/bin* sub-directory, and execute the following command:

```
./studio.sh
```

2.4 The Android Studio setup wizard

If you have previously installed an earlier version of Android Studio, the first time this new version is launched, a dialog may appear providing the option to import settings from a previous Android Studio version. If you have settings from a previous version and would like to import them into the latest installation, select the appropriate option and location. Alternatively, indicate that you do not need to import any previous settings and click the OK button to proceed.

If you are installing Android Studio for the first time, the initial dialog that appears once the setup process starts may resemble that shown in Figure 2-2 below:

Figure 2-2

If this dialog appears, click the Next button to display the Install Type screen (Figure 2-3). On this screen, select the Standard installation option before clicking Next.

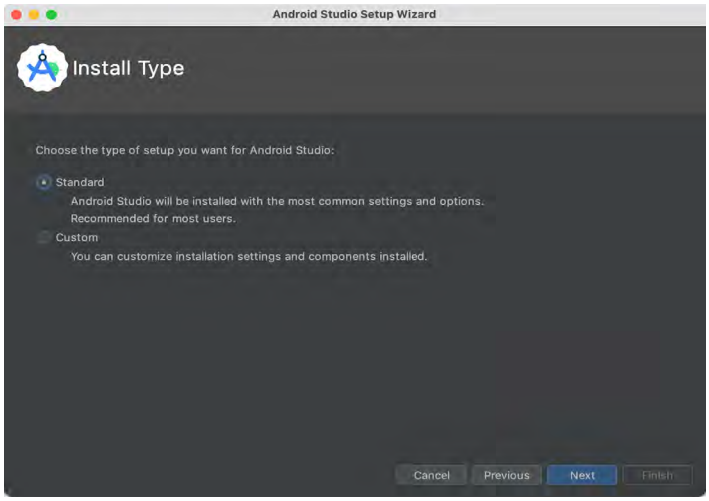

Figure 2-3

On the Select UI Theme screen, select either the Darcula or Light theme based on your preferences. After making a choice, click Next, and review the options in the Verify Settings screen before proceeding to the License Agreement screen. Select each license category and enable the Accept checkbox. Finally, click the Finish button to initiate the installation.

After these initial setup steps have been taken, click the Finish button to display the Welcome to Android Studio screen using your chosen UI theme:

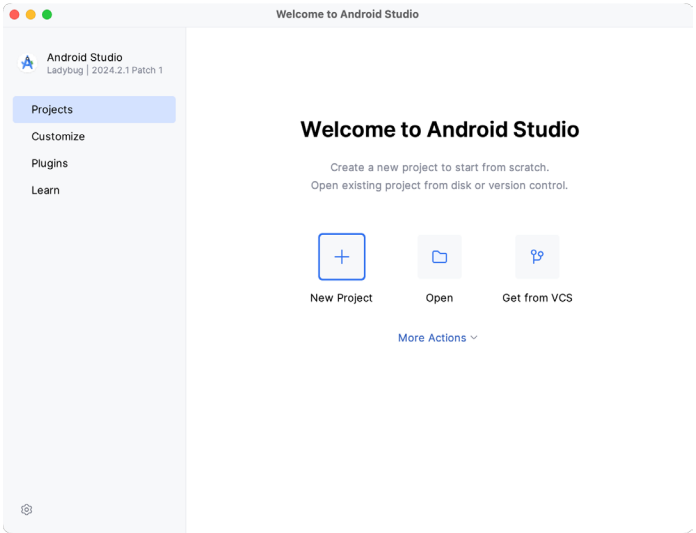

Figure 2-4

2.5 Installing additional Android SDK packages

The steps performed so far have installed the Android Studio IDE and the current set of default Android SDK packages. Before proceeding, it is worth taking some time to verify which packages are installed and to install any missing or updated packages.

This task can be performed by clicking on the *More Actions* link within the welcome dialog and selecting the

SDK Manager option from the drop-down menu. Once invoked, the *Android SDK* screen of the Settings dialog will appear as shown in Figure 2-5:

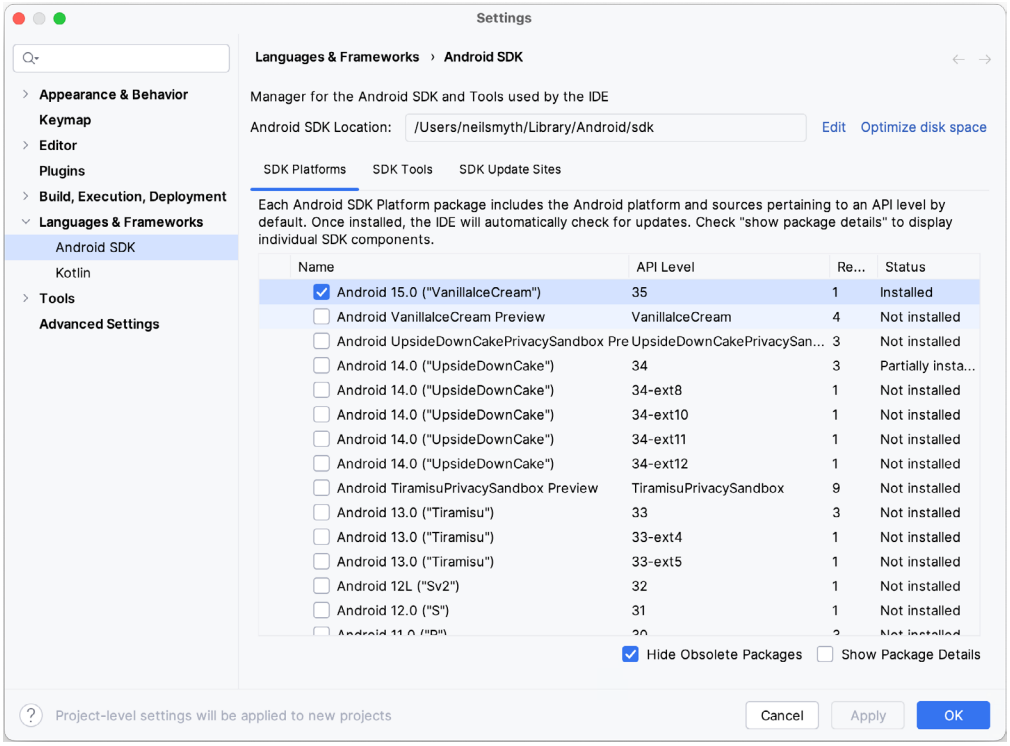

Figure 2-5

Google pairs each release of Android Studio with a maximum supported Application Programming Interface (API) level of the Android SDK. In the case of Android Studio Ladybug, this is Android VanillaIceCream (API Level 35). This information can be confirmed using the following link:

https://developer.android.com/studio/releases#api-level-support

Immediately after installing Android Studio for the first time, it is likely that only the latest supported version of the Android SDK has been installed. To install older versions of the Android SDK, select the checkboxes corresponding to the versions and click the *Apply* button. The rest of this book assumes that the Android VanillaIceCream (API Level 35) SDK is installed.

Most of the examples in this book will support older versions of Android as far back as Android 8.0 (Oreo). This ensures that the apps run on a wide range of Android devices. Within the list of SDK versions, enable the checkbox next to Android 8.0 (Oreo) and click the Apply button. Click the OK button to install the SDK in the resulting confirmation dialog. Subsequent dialogs will seek the acceptance of licenses and terms before performing the installation. Click Finish once the installation is complete.

It is also possible that updates will be listed as being available for the latest SDK. To access detailed information about the packages that are ready to be updated, enable the *Show Package Details* option located in the lower right-hand corner of the screen. This will display information similar to that shown in Figure 2-6:

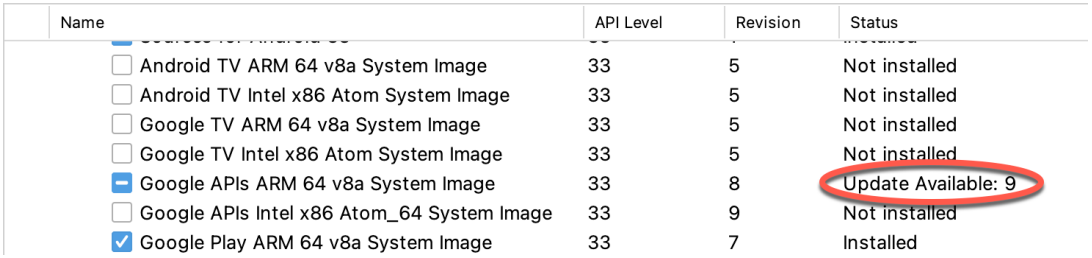

Figure 2-6

The above figure highlights the availability of an update. To install the updates, enable the checkbox to the left of the item name and click the *Apply* button.

In addition to the Android SDK packages, several tools are also installed for building Android applications. To view the currently installed packages and check for updates, remain within the SDK settings screen and select the SDK Tools tab as shown in Figure 2-7:

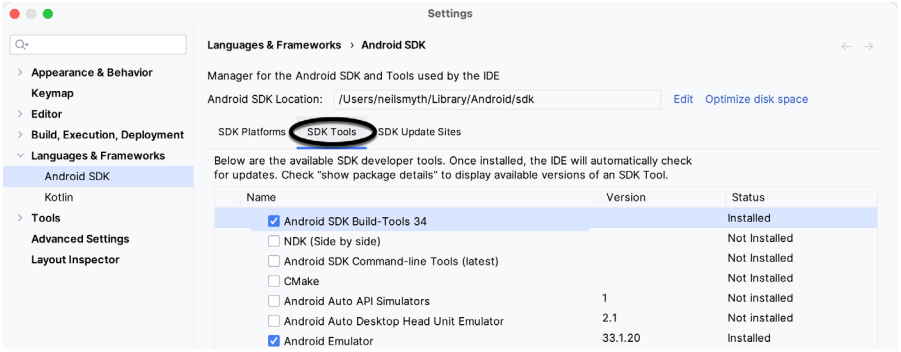

Figure 2-7

Within the Android SDK Tools screen, make sure that the following packages are listed as *Installed* in the Status column:

- Android SDK Build-tools
- Android Emulator
- Android SDK Platform-tools
- Google Play Services
- Intel x86 Emulator Accelerator (HAXM installer)*
- Google USB Driver (Windows only)
- Layout Inspector image server for API 31-35

*Note that the Intel x86 Emulator Accelerator (HAXM installer) requires an Intel processor with VT-x support enabled. It cannot be installed on Apple silicon-based Macs or AMD-based PCs.

If any of the above packages are listed as *Not Installed* or requiring an update, select the checkboxes next to those packages and click the *Apply* button to initiate the installation process. If the HAXM emulator settings dialog appears, select the recommended memory allocation:

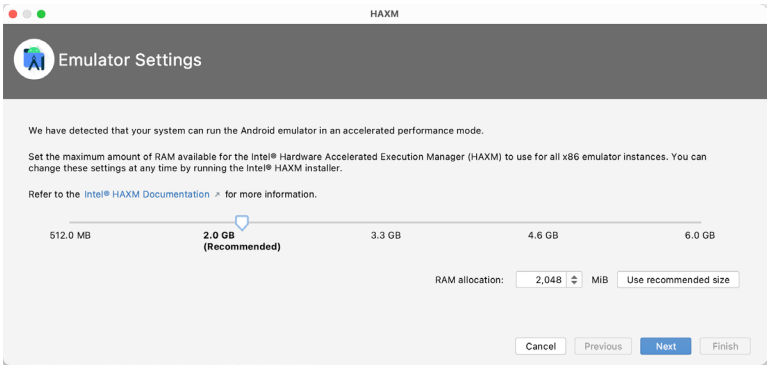

Figure 2-8

Once the installation is complete, review the package list and ensure that the selected packages are listed as *Installed* in the *Status* column. If any are listed as *Not installed,* make sure they are selected and click the *Apply* button again.

2.6 Installing the Android SDK Command-line Tools

Android Studio includes tools that allow some tasks to be performed from your operating system command line. To install these tools on your system, open the SDK Manager, select the SDK Tools tab, and locate the *Android SDK Command-line Tools (latest)* package as shown in Figure 2-9:

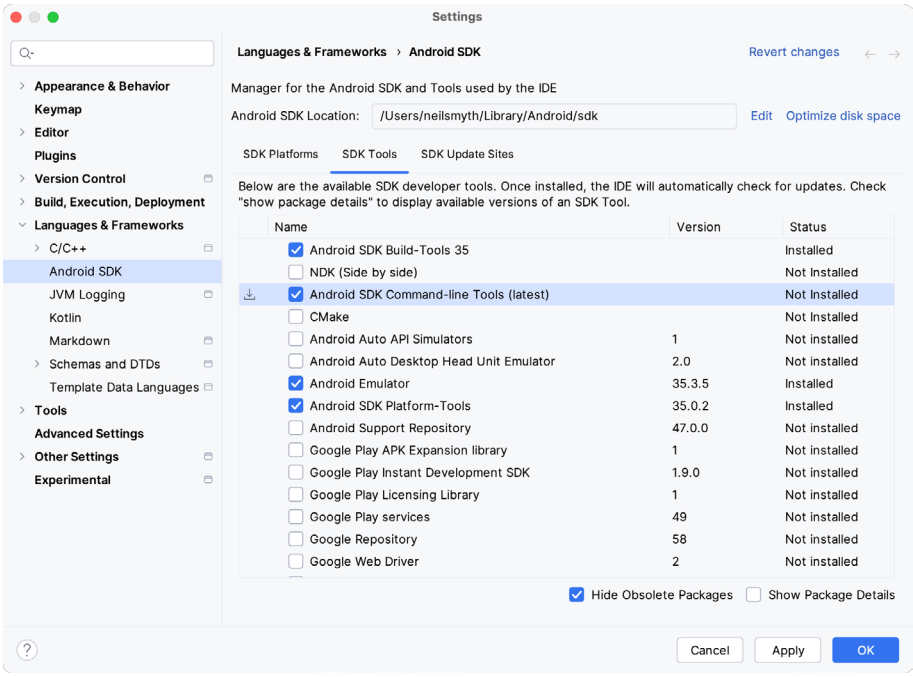

Figure 2-9

If the command-line tools package is not already installed, enable it and click Apply, followed by OK to complete the installation. When the installation completes, click Finish and close the SDK Manager dialog.

For the operating system on which you are developing to be able to find these tools, it will be necessary to add them to the system's *PATH* environment variable.

Regardless of your operating system, you will need to configure the PATH environment variable to include the following paths (where *<path_to_android_sdk_installation>* represents the file system location into which you installed the Android SDK):

```
<path_to_android_sdk_installation>/sdk/cmdline-tools/latest/bin
<path_to_android_sdk_installation>/sdk/platform-tools
```

You can identify the location of the SDK on your system by launching the SDK Manager and referring to the *Android SDK Location:* field located at the top of the settings panel, as highlighted in Figure 2-10:

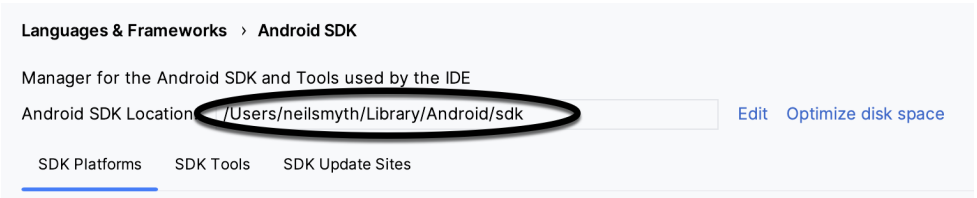

Figure 2-10

Once the location of the SDK has been identified, the steps to add this to the PATH variable are operating system dependent:

2.6.1 Windows 8.1

1. On the start screen, move the mouse to the bottom right-hand corner of the screen and select Search from the resulting menu. In the search box, enter Control Panel. When the Control Panel icon appears in the results area, click on it to launch the tool on the desktop.

2. Within the Control Panel, use the Category menu to change the display to Large Icons. From the list of icons, select the one labeled System.

3. In the Environment Variables dialog, locate the Path variable in the System variables list, select it, and click the *Edit…* button. Using the *New* button in the edit dialog, add two new entries to the path. For example, assuming the Android SDK was installed into *C:\Users\demo\AppData\Local\Android\Sdk*, the following entries would need to be added:

```
C:\Users\demo\AppData\Local\Android\Sdk\cmdline-tools\latest\bin
C:\Users\demo\AppData\Local\Android\Sdk\platform-tools
```

4. Click OK in each dialog box and close the system properties control panel.

Open a command prompt window by pressing Windows + R on the keyboard and entering *cmd* into the Run dialog. Within the Command Prompt window, enter:

```
echo %Path%
```

The returned path variable value should include the paths to the Android SDK platform tools folders. Verify that the *platform-tools* value is correct by attempting to run the *adb* tool as follows:

```
adb
```

The tool should output a list of command-line options when executed.

Similarly, check the *tools* path setting by attempting to run the AVD Manager command-line tool (don't worry if the avdmanager tool reports a problem with Java - this will be addressed later):

```
avdmanager
```

If a message similar to the following message appears for one or both of the commands, it is most likely that an incorrect path was appended to the Path environment variable:

```
'adb' is not recognized as an internal or external command,
operable program or batch file.
```

2.6.2 Windows 10

Right-click on the Start menu, select Settings from the resulting menu and enter "Edit the system environment variables" into the *Find a setting* text field. In the System Properties dialog, click the *Environment Variables...* button. Follow the steps outlined for Windows 8.1 starting from step 3.

2.6.3 Windows 11

Right-click on the Start icon located in the taskbar and select Settings from the resulting menu. When the Settings dialog appears, scroll down the list of categories and select the "About" option. In the About screen, select *Advanced system settings* from the Related links section. When the System Properties window appears, click the *Environment Variables...* button. Follow the steps outlined for Windows 8.1 starting from step 3.

2.6.4 Linux

This configuration can be achieved on Linux by adding a command to the *.bashrc* file in your home directory (specifics may differ depending on the particular Linux distribution in use). Assuming that the Android SDK bundle package was installed into */home/demo/Android/sdk*, the export line in the *.bashrc* file would read as follows:

```
export PATH=/home/demo/Android/sdk/platform-tools:/home/demo/Android/sdk/cmdline-
tools/latest/bin:/home/demo/android-studio/bin:$PATH
```

Note also that the above command adds the *android-studio/bin* directory to the PATH variable. This will enable the *studio.sh* script to be executed regardless of the current directory within a terminal window.

2.6.5 macOS

Several techniques may be employed to modify the $PATH environment variable on macOS. Arguably the cleanest method is to add a new file in the */etc/paths.d* directory containing the paths to be added to $PATH. Assuming an Android SDK installation location of */Users/demo/Library/Android/sdk*, the path may be configured by creating a new file named *android-sdk* in the */etc/paths.d* directory containing the following lines:

```
/Users/demo/Library/Android/sdk/cmdline-tools/latest/bin
/Users/demo/Library/Android/sdk/platform-tools
```

Note that since this is a system directory, it will be necessary to use the *sudo* command when creating the file. For example:

```
sudo vi /etc/paths.d/android-sdk
```

2.7 Android Studio memory management

Android Studio is a large and complex software application with many background processes. Although Android Studio has been criticized in the past for providing less than optimal performance, Google has made significant performance improvements in recent releases and continues to do so with each new version. These improvements include allowing the user to configure the amount of memory used by both the Android Studio IDE and the background processes used to build and run apps. This allows the software to take advantage of systems with larger amounts of RAM.

If you are running Android Studio on a system with sufficient unused RAM to increase these values (this feature is only available on 64-bit systems with 5GB or more of RAM) and find that Android Studio performance appears to be degraded, it may be worth experimenting with these memory settings. Android Studio may also notify you that performance can be increased via a dialog similar to the one shown below:

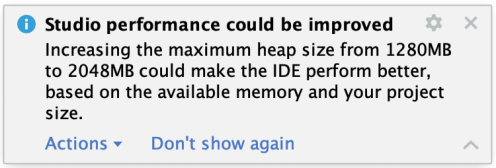

Figure 2-11

To view and modify the current memory configuration, select the *File -> Settings...* main menu option (*Android Studio -> Settings...* on macOS) and, in the resulting dialog, select *Appearance & Behavior* followed by the *Memory Settings* option listed under *System Settings* in the left-hand navigation panel, as illustrated in Figure 2-12 below:

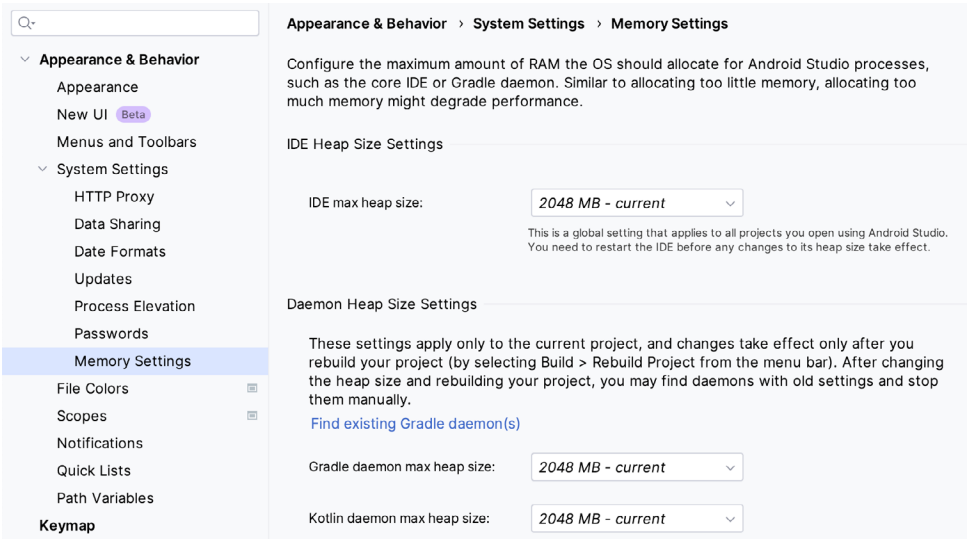

Figure 2-12

When changing the memory allocation, be sure not to allocate more memory than necessary or than your system can spare without slowing down other processes.

The IDE heap size setting adjusts the memory allocated to Android Studio and applies regardless of the currently loaded project. On the other hand, when a project is built and run from within Android Studio, several background processes (referred to as daemons) perform the task of compiling and running the app. When compiling and running large and complex projects, build time could be improved by adjusting the daemon heap settings. Unlike the IDE heap settings, these daemon settings apply only to the current project and can only be accessed when a project is open in Android Studio. To display the SDK Manager from within an open project, select the *Tools -> SDK Manager...* menu option from the main menu.

2.8 Updating Android Studio and the SDK

From time to time, new versions of Android Studio and the Android SDK are released. New versions of the SDK are installed using the Android SDK Manager. Android Studio will typically notify you when an update is ready to be installed.

To manually check for Android Studio updates, use the *Help -> Check for Updates...* menu option from the Android Studio main window (*Android Studio -> Check for Updates...* on macOS).

2.9 Summary

Before beginning the development of Android-based applications, the first step is to set up a suitable development environment. This consists of the Android SDKs and Android Studio IDE (which also includes the OpenJDK development environment). This chapter covers the steps necessary to install these packages on Windows, macOS, and Linux.

Chapter 13

13. Kotlin Operators and Expressions

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

13.1 Expression syntax in Kotlin

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

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

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

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

13.2 The Basic assignment operator

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

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

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

13.3 Kotlin arithmetic operators

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

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

The following table lists the primary Kotlin arithmetic operators:

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

/	Division
+	Addition
-	Subtraction
%	Remainder/Modulo

Table 13-1

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

For example:

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

13.4 Augmented assignment operators

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

```
x = x + y
```

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

```
x += y
```

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

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

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

Table 13-2

13.5 Increment and decrement operators

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

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

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

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

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

value of 10:

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

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

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

13.6 Equality operators

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

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

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

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

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

result = x < y
```

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

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

Table 13-3

13.7 Boolean logical operators

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

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

```
val flag = true // variable is true
```

```
val secondFlag = !flag // secondFlag set to false
```

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

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

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

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

13.8 Range operator

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

The syntax for the range operator is as follows:

```
x..y
```

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

13.9 Bitwise operators

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

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

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

```
10101011
```

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

```
00000011
```

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

13.9.1 Bitwise inversion

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

```
00000011 NOT
```

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

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

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

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

13.9.2 Bitwise AND

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

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

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

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

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

13.9.3 Bitwise OR

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

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

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

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

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

13.9.4 Bitwise XOR

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

```
10101011 XOR
```

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

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

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

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

When executed, we get the following output from print:

```
Result is 168
```

13.9.5 Bitwise left shift

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

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

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

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

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

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

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

13.9.6 Bitwise right shift

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

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

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

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

```
val x = 171
```

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

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

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

13.10 Take the knowledge test

Click the link below or scan the QR code to test your knowledge and understanding of Kotlin operators and expressions:

https://www.answertopia.com/3xrq

13.11 Summary

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

Chapter 22

22. An Overview of Compose State and Recomposition

State is the cornerstone of how the Compose system is implemented. As such, a clear understanding of state is an essential step in becoming a proficient Compose developer. In this chapter, we will explore and demonstrate the basic concepts of state and explain the meaning of related terms such as *recomposition*, *unidirectional data flow,* and *state hoisting*. The chapter will also cover saving and restoring state through *configuration changes*.

22.1 The basics of state

In declarative languages such as Compose, *state* is generally referred to as "a value that can change over time". At first glance, this sounds much like any other data in an app. A standard Kotlin variable, for example, is by definition designed to store a value that can change at any time during execution. State, however, differs from a standard variable in two significant ways.

First, the value assigned to a state variable in a composable function needs to be remembered. In other words, each time a composable function containing state (a *stateful function*) is called, it must remember any state values from the last time it was invoked. This is different from a standard variable which would be re-initialized each time a call is made to the function in which it is declared.

The second key difference is that a change to any state variable has far reaching implications for the entire hierarchy tree of composable functions that make up a user interface. To understand why this is the case, we now need to talk about recomposition.

22.2 Introducing recomposition

When developing with Compose, we build apps by creating hierarchies of composable functions. As previously discussed, a composable function can be thought of as taking data and using that data to generate sections of a user interface layout. These elements are then rendered on the screen by the Compose runtime system. In most cases, the data passed from one composable function to another will have been declared as a state variable in a parent function. This means that any change of state value in a parent composable will need to be reflected in any child composables to which the state has been passed. Compose addresses this by performing an operation referred to as *recomposition*.

Recomposition occurs whenever a state value changes within a hierarchy of composable functions. As soon as Compose detects a state change, it works through all of the composable functions in the activity and recomposes any functions affected by the state value change. Recomposing simply means that the function gets called again and passed the new state value.

Recomposing the entire composable tree for a user interface each time a state value changes would be a highly inefficient approach to rendering and updating a user interface. Compose avoids this overhead using a technique called *intelligent recomposition* that involves only recomposing those functions directly affected by the state change. In other words, only functions that read the state value will be recomposed when the value changes.

22.3 Creating the StateExample project

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

Enter *StateExample* into the Name field and specify *com.example.stateexample* as the package name. Before clicking on the Finish button, change the Minimum API level setting to API 26: Android 8.0 (Oreo). On completion of the project creation process, the StateExample project should be listed in the Project tool window located along the left-hand edge of the Android Studio main window.

22.4 Declaring state in a composable

The first step in declaring a state value is to wrap it in a MutableState object. MutableState is a Compose class which is referred to as an *observable type*. Any function that reads a state value is said to have *subscribed* to that observable state. As a result, any changes to the state value will trigger the recomposition of all subscribed functions.

Within Android Studio, open the *MainActivity.kt* file, delete the Greeting composable and modify the class so that it reads as follows:

```
package com.example.stateexample
.
.
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            StateExampleTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    DemoScreen(Modifier.padding(innerPadding))
                }
            }
        }
    }
}


@Composable
fun DemoScreen(modifier: Modifier = Modifier) {
    MyTextField()
}

@Composable
fun MyTextField() {

}

@Preview(showBackground = true)
@Composable
```

```
fun GreetingPreview() {
    StateExampleTheme {
        DemoScreen()
    }
}
```

The objective here is to implement MyTextField as a stateful composable function containing a state variable and an event handler that changes the state based on the user's keyboard input. The result is a text field in which the characters appear as they are typed.

MutableState instances are created by making a call to the *mutableStateOf()* runtime function, passing through the initial state value. The following, for example, creates a MutableState instance initialized with an empty String value:

```
var textState = { mutableStateOf("") }
```

This provides an observable state which will trigger a recomposition of all subscribed functions when the contained value is changed. The above declaration is, however, missing a key element. As previously discussed, state must be remembered through recompositions. As currently implemented, the state will be reinitialized to an empty string each time the function in which it is declared is recomposed. To retain the current state value, we need to use the *remember* keyword:

```
var myState = remember { mutableStateOf("") }
```

Remaining within the *MainActivity.kt* file, add some imports and modify the MyTextField composable as follows:

```
.
.
import androidx.compose.material3.*
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.foundation.layout.Column
.
.
@Composable
fun MyTextField() {

    var textState = remember { mutableStateOf("") }

    val onTextChange = { text : String ->
        textState.value = text
    }

    TextField(
        value = textState.value,
        onValueChange = onTextChange
    )
}
```

Test the code using the Preview panel in interactive mode and confirm that keyboard input appears in the TextField as it is typed.

When looking at Compose code examples, you may see MutableState objects declared in different ways. When using the above format, it is necessary to read and set the *value* property of the MutableState instance. For example, the event handler to update the state reads as follows:

```
val onTextChange = { text: String ->
     textState.value = text
}
```

Similarly, the current state value is assigned to the TextField as follows:

```
TextField(
    value = textState.value,
    onValueChange = onTextChange
)
```

A more common and concise approach to declaring state is to use Kotlin property delegates via the *by* keyword as follows (note that two additional libraries need to be imported when using property delegates):

```
.
.
import androidx.compose.runtime.getValue
import androidx.compose.runtime.setValue
.
.
@Composable
fun MyTextField() {

    var textState by remember { mutableStateOf("") }
.
.
```

We can now access the state value without needing to directly reference the MutableState *value* property within the event handler:

```
val onTextChange = { text: String ->
     textState = text
}
```

This also makes reading the current value more concise:

```
TextField(
    value = textState,
    onValueChange = onTextChange
 )
```

A third technique separates the access to a MutableState object into a *value* and a *setter function* as follows:

```
var (textValue, setText) = remember { mutableStateOf("") }
```

When changing the value assigned to the state we now do so by calling the *setText* setter, passing through the new value:

```
val onTextChange = { text: String ->
     setText(text)
}
```

The state value is now accessed by referencing *textValue*:

```
TextField(
    value = textValue,
    onValueChange = onTextChange
)
```

In most cases, the use of the *by* keyword and property delegates is the most commonly used technique because it results in cleaner code. Before continuing with the chapter, revert the example to use the *by* keyword.

22.5 Unidirectional data flow

Unidirectional data flow is an approach to app development whereby state stored in a composable should not be directly changed by any child composable functions. Consider, for example, a composable function named FunctionA containing a state value in the form of a Boolean value. This composable calls another composable function named FunctionB that contains a Switch component. The objective is for the switch to update the state value each time the switch position is changed by the user. In this situation, adherence to unidirectional data flow prohibits FunctionB from directly changing the state value.

Instead, FunctionA would declare an event handler (typically in the form of a lambda) and pass it as a parameter to the child composable along with the state value. The Switch within FunctionB would then be configured to call the event handler each time the switch position changes, passing it the current setting value. The event handler in FunctionA will then update the state with the new value.

Make the following changes to the *MainActivity.kt* file to implement FunctionA and FunctionB together with a corresponding modification to the preview composable:

```
.
.
@Composable
fun DemoScreen(modifier: Modifier = Modifier) {

    var textState by remember { mutableStateOf("") }

    val onTextChange = { text : String ->
        textState = text
    }

    Column {
        MyTextField(text = textState, onTextChange = onTextChange)
        FunctionA()
    }
}
.
.
@Composable
fun FunctionA() {

    var switchState by remember { mutableStateOf(true) }

    val onSwitchChange = { value : Boolean ->
```

```
            switchState = value
        }
}

@Composable
fun FunctionB(switchState: Boolean, onSwitchChange : (Boolean) -> Unit ) {
        Switch(
            checked = switchState,
            onCheckedChange = onSwitchChange
        )
}
```

Preview the app using interactive mode and verify that clicking the switch changes the slider position between on and off states.

We can now use this example to break down the state process into the following individual steps which occur when FunctionA is called:

1. The *switchState* state variable is initialized with a true value.

2. The *onSwitchChange* event handler is declared to accept a Boolean parameter which it assigns to *switchState* when called.

3. FunctionB is called and passed both *switchState* and a reference to the *onSwitchChange* event handler.

4. FunctionB calls the built-in Switch component and configures it to display the state assigned to *switchState*. The Switch component is also configured to call the *onSwitchChange* event handler when the user changes the switch setting.

5. Compose renders the Switch component on the screen.

The above sequence explains how the Switch component gets rendered on the screen when the app first launches. We can now explore the sequence of events that occur when the user slides the switch to the "off" position:

1. The switch is moved to the "off" position.

2. The Switch component calls the *onSwitchChange* event handler passing through the current switch position value (in this case *false*).

3. The *onSwitchChange* lambda declared in FunctionA assigns the new value to *switchState*.

4. Compose detects that the *switchState* state value has changed and initiates a recomposition.

5. Compose identifies that FunctionB contains code that reads the value of *switchState* and therefore needs to be recomposed.

6. Compose calls FunctionB with the latest state value and the reference to the event handler.

7. FunctionB calls the Switch composable and configures it with the state and event handler.

8. Compose renders the Switch on the screen, this time with the switch in the "off" position.

The key point to note about this process is that the value assigned to *switchState* is only changed from within FunctionA and never directly updated by FunctionB. The Switch setting is not moved from the "on" position to the "off" position directly by FunctionB. Instead, the state is changed by calling upwards to the event handler

located in FunctionA, and allowing recomposition to regenerate the Switch with the new position setting.

As a general rule, data is passed down through a composable hierarchy tree while events are called upwards to handlers in ancestor components as illustrated in Figure 22-1:

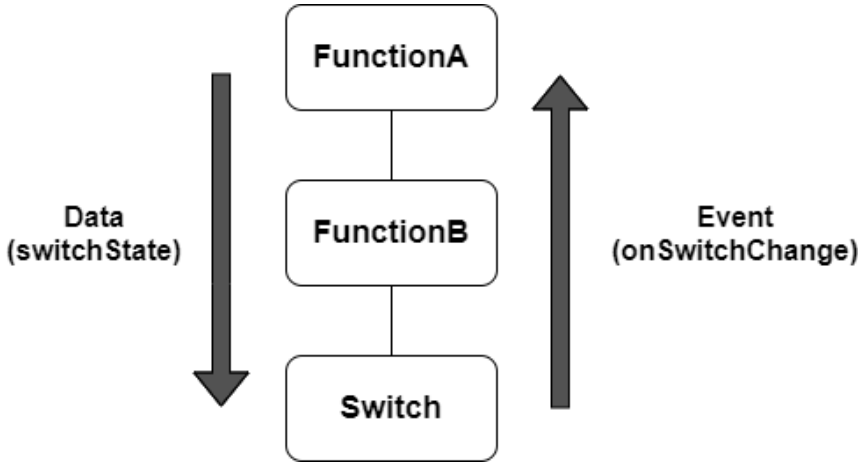

Figure 22-1

22.6 State hoisting

If you look up the word "hoist" in a dictionary it will likely be defined as the act of raising or lifting something. The term *state hoisting* has a similar meaning in that it involves moving state from a child composable up to the calling (parent) composable or a higher ancestor. When the child composable is called by the parent, it is passed the state along with an event handler. When an event occurs in the child composable that requires an update to the state, a call is made to the event handler passing through the new value as outlined earlier in the chapter. This has the advantage of making the child composable stateless and, therefore, easier to reuse. It also allows the state to be passed down to other child composables later in the app development process.

Consider our MyTextField example from earlier in the chapter:

```
@Composable
fun DemoScreen(modifier: Modifier = Modifier) {
    MyTextField()
}

@Composable
fun MyTextField() {

    var textState by remember { mutableStateOf("") }

    val onTextChange = { text : String ->
        textState = text
    }

    TextField(
        value = textState,
        onValueChange = onTextChange
```

```
    )
}
```

The self-contained nature of the MyTextField composable means that it is not a particularly useful component. One issue is that the text entered by the user is not accessible to the calling function and, therefore, cannot be passed to any sibling functions. It is also not possible to pass a different state and event handler through to the function, thereby limiting its re-usability.

To make the function more useful we need to hoist the state into the parent DemoScreen function as follows:

```
.
.
@Composable
fun DemoScreen(modifier: Modifier = Modifier) {

    var textState by remember { mutableStateOf("") }

    val onTextChange = { text : String ->
        textState = text
    }

    MyTextField(text = textState, onTextChange = onTextChange)
}

@Composable
fun MyTextField(text: String, onTextChange : (String) -> Unit) {

    var textState by remember { mutableStateOf("") }

    val onTextChange = { text : String ->
        textState = text
    }

    TextField(
        value = text,
        onValueChange = onTextChange
    )
}
.
.
```

With the state hoisted to the parent function, MyTextField is now a stateless, reusable composable which can be called and passed any state and event handler. Also, the text entered by the user is now accessible by the parent function and may be passed down to other composables if necessary.

State hoisting is not limited to moving to the immediate parent of a composable. State can be raised any number of levels upward within the composable hierarchy and subsequently passed down through as many layers of children as needed (within reason). This will often be necessary when multiple children need access to the same state. In such a situation, the state will need to be hoisted up to an ancestor that is common to both children.

In Figure 22-2 below, for example, both NameField and NameText need access to *textState*. The only way to make the state available to both composables is to hoist it up to the MainScreen function since this is the only ancestor both composables have in common:

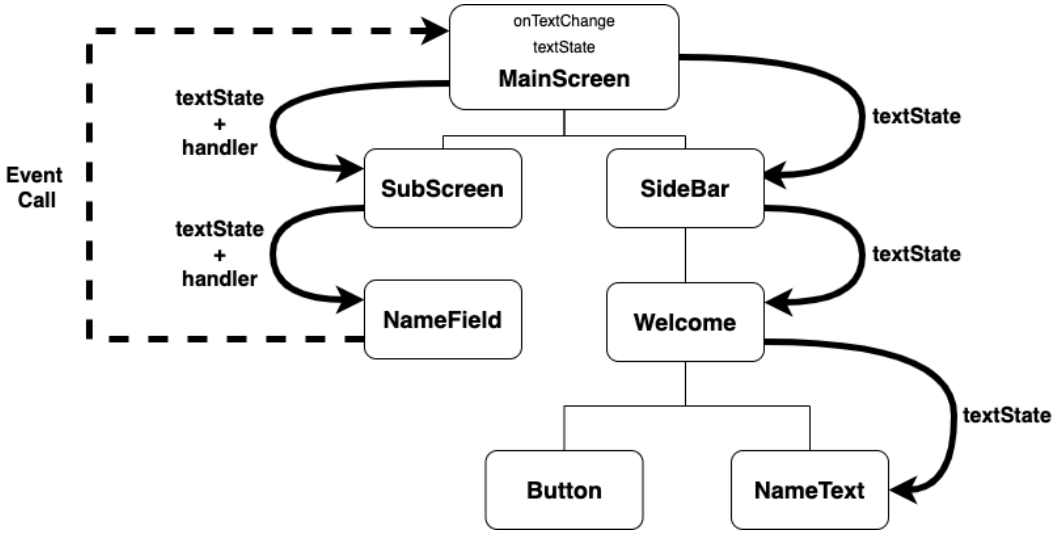

Figure 22-2

The solid arrows indicate the path of *textState* as it is passed down through the hierarchy to the NameField and NameText functions (in the case of the NameField, a reference to the event handler is also passed down), while the dotted line represents the calls from NameField function to an event handler declared in MainScreen as the text changes.

When adding state to a function, take some time to decide whether hoisting state to the caller (or higher) might make for a more re-usable and flexible composable. While situations will arise where state is only needed to be used locally in a composable, in most cases it probably makes sense to hoist the state up to an ancestor.

22.7 How high to hoist?

Hoisting is a good thing, but how high is too high? What goes up must come down, so remember that you must pass the state through each generation of descendants until you reach the lowest subscribing composable. Often, multiple children will subscribe to a single state, so the state should usually only be hoisted as high as the closest common ancestor.

Suppose functions F and J in Figure 22-3 below subscribe to the same state. Function D is the closest ancestor, so it is an excellent candidate to hold the state:

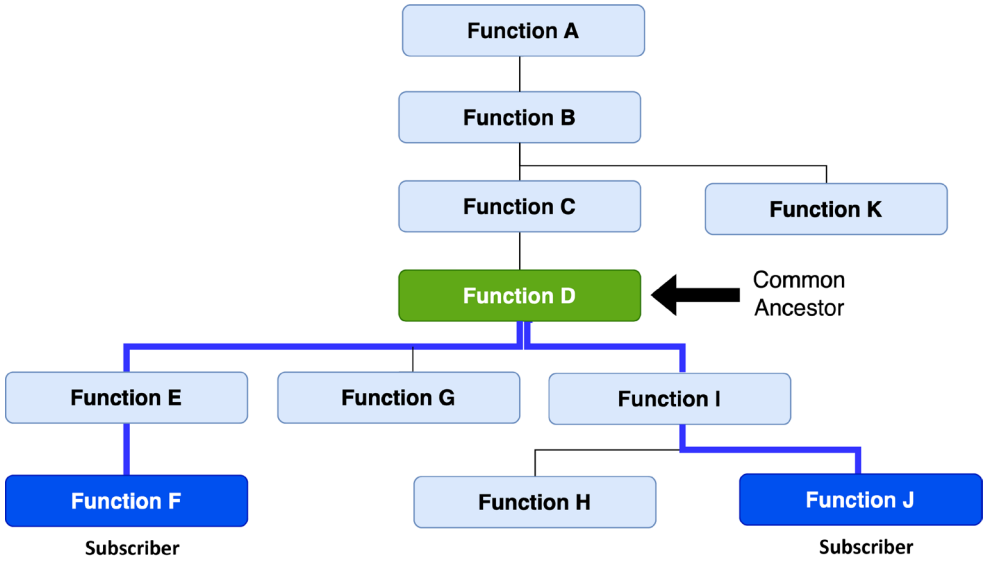

Figure 22-3

If function K also needs to subscribe, the state must be hoisted to function B:

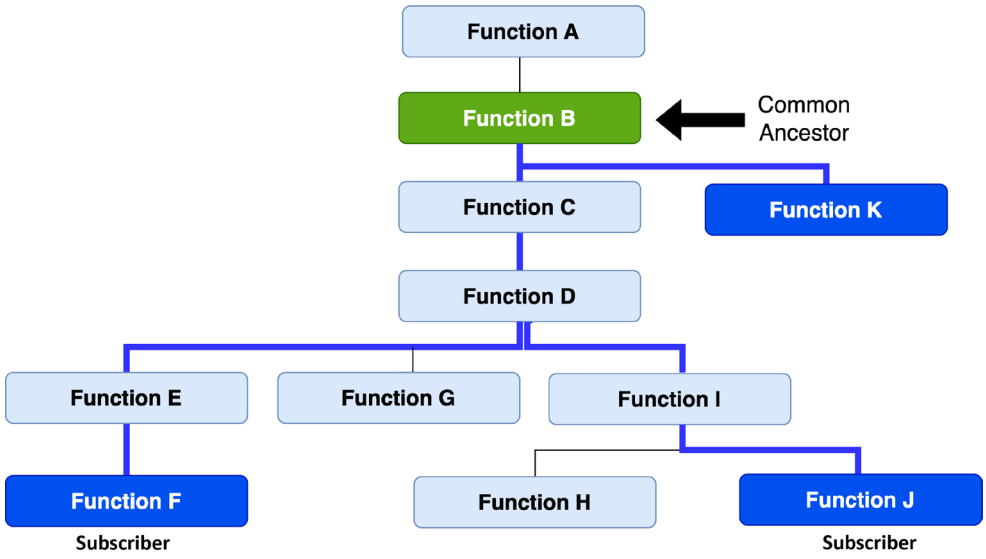

Figure 22-4

If you have declared a state in the lowest common ancestor and find yourself passing the state through an excessive number of descendants to reach all the subscribed children, consider using CompositionLocalProvider, a topic covered in the chapter entitled *"An Introduction to Composition Local"*.

22.8 Saving state through configuration changes

We now know that the *remember* keyword can be used to save state values through recompositions. This technique does not, however, retain state between *configuration changes*. A configuration change generally occurs when some aspect of the device changes in a way that alters the appearance of an activity (such as rotating the orientation of the device between portrait and landscape or changing a system-wide font setting).

Changes such as these will cause the entire activity to be destroyed and recreated. The reasoning behind this is that these changes affect resources such as the layout of the user interface and simply destroying and recreating impacted activities is the quickest way for an activity to respond to the configuration change. The result is a newly initialized activity with no memory of any previous state values.

To experience the effect of a configuration change, run the StateExample app on an emulator or device and, once running, enter some text so that it appears in the TextField before changing the orientation from portrait to landscape. When using the emulator, device rotation may be simulated using the rotation button located in the emulator toolbar. To complete the rotation on Android 11 or older, it may also be necessary to tap on the rotation button. This appears in the toolbar of the device or emulator screen as shown in Figure 22-5:

Figure 22-5

Before performing the rotation on Android 12 or later, you may need to enter the Settings app, select the Display category and enable the *Auto-rotate screen* option.

Note that after rotation, the TextField is now blank and the text entered has been lost. In situations where state needs to be retained through configuration changes, Compose provides the *rememberSaveable* keyword. When *rememberSaveable* is used, the state will be retained not only through recompositions, but also configuration changes. Modify the *textState* declaration to use *rememberSaveable* as follows:

```
.
.
import androidx.compose.runtime.saveable.rememberSaveable
.
.
@Composable
fun DemoScreen(modifier: Modifier = Modifier) {

    var textState by rememberSaveable { mutableStateOf("") }
.
.
```

Build and run the app once again, enter some text and perform another rotation. Note that the text is now preserved following the configuration change.

22.9 Take the knowledge test

Click the link below or scan the QR code to test your knowledge and understanding of Compose state and recomposition:

https://www.answertopia.com/c3sb

22.10 Summary

When developing apps with Compose it is vital to have a clear understanding of how state and recomposition work together to make sure that the user interface is always up to date. In this chapter, we have explored state and described how state values are declared, updated, and passed between composable functions. You should also have a better understanding of recomposition and how it is triggered in response to state changes.

We also introduced the concept of unidirectional data flow and explained how data flows down through the compose hierarchy while data changes are made by making calls upward to event handlers declared within ancestor stateful functions.

An important goal when writing composable functions is to maximize re-usability. This can be achieved, in part, by hoisting state out of a composable up to the calling parent or a higher function in the compose hierarchy.

Finally, the chapter described configuration changes and explained how such changes result in the destruction and recreation of entire activities. Ordinarily, state is not retained through configuration changes unless specifically configured to do so using the *rememberSaveable* keyword.

Chapter 38

38. 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.

38.1 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.

38.2 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.

38.3 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 38-1 below:

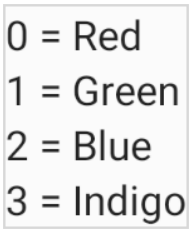

Figure 38-1

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.

38.4 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()
    }
}
```

38.5 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)
    }
.
.
}
```

38.6 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 38-2, 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:

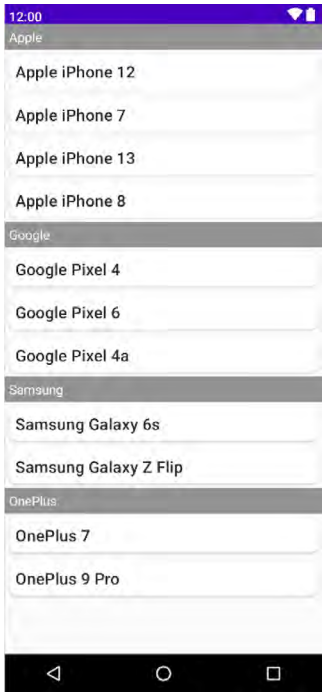

Figure 38-2

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 38-2 above.

38.7 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
}
```

38.8 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(GridCells.Adaptive(minSize = 60.dp),
    state = rememberLazyGridState(),
```

```
    contentPadding = PaddingValues(10.dp)
) {
    items(30) { index ->
        Card(
            colors = CardDefaults.cardColors(
                containerColor =  MaterialTheme.colorScheme.primary
            ),
            modifier = Modifier.padding(5.dp).fillMaxSize()) {

            Text(
                "$index",
                textAlign = TextAlign.Center,
                fontSize = 30.sp,
                color = Color.White,
                modifier = Modifier.width(120.dp)
            )
        }
    }
}
```

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

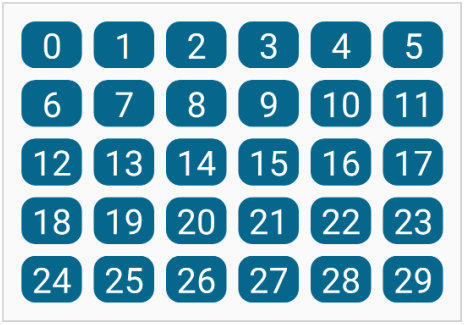

Figure 38-3

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

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

    items(15) { index ->
        Card(colors = CardDefaults.cardColors(
            containerColor =  MaterialTheme.colorScheme.primary
        ),
            modifier = Modifier.padding(5.dp).fillMaxSize()) {
            Text(
```

```
                "$index",
                fontSize = 35.sp,
                color = Color.White,
                textAlign = TextAlign.Center,
                modifier = Modifier.width(120.dp))
        }
    }
}
```

The layout from the above code will appear as illustrated in Figure 38-4 below:

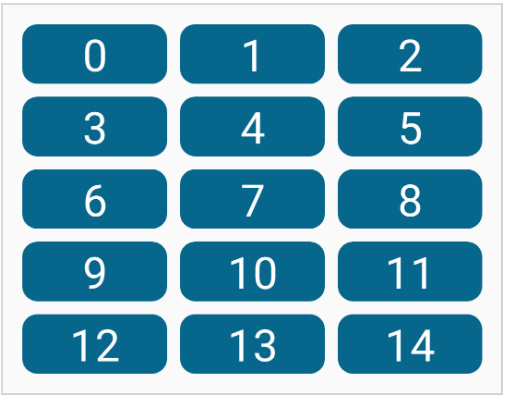

Figure 38-4

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

When rendered, the above Card component will appear as shown in Figure 38-5:

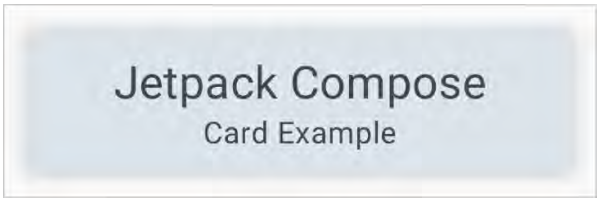

Figure 38-5

38.9 Take the knowledge test

Click the link below or scan the QR code to test your knowledge and understanding of row and column lists:

https://www.answertopia.com/iifq

38.10 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.

Chapter 47

47. Working with ViewModels in Compose

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

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

47.1 What is Android Jetpack?

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

47.2 The "old" architecture

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

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

47.3 Modern Android architecture

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

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

47.4 The ViewModel component

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

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

A ViewModel is implemented as a separate class and contains *state* values containing the model data and functions that can be called to manage that data. The activity containing the user interface *observes* the model state values such that any value changes trigger a recomposition. User interface events relating to the model data such as a button click are configured to call the appropriate function within the ViewModel. This is, in fact, a direct implementation of the *unidirectional data flow* concept described in the chapter entitled *"An Overview of Compose State and Recomposition"*. The diagram in Figure 47-1 illustrates this concept as it relates to activities and ViewModels:

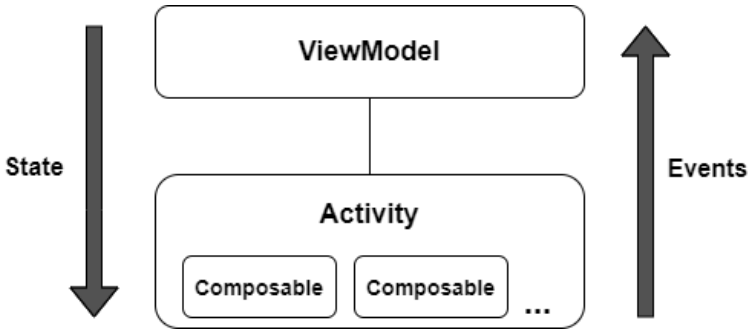

Figure 47-1

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

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

47.5 ViewModel implementation using state

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

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

```
class MyViewModel : ViewModel() {

    var customerCount by mutableStateOf(0)
}
```

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

```
class MyViewModel : ViewModel() {

    var customerCount by mutableStateOf(0)

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

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

47.6 Connecting a ViewModel state to an activity

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

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            ViewModelDemoTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    TopLevel(modifier = Modifier.padding(innerPadding))
                }
            }
        }
    }
}

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

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

```
        Button(
            onClick = addCount,
        ) {
            Text(text = "Add a Customer")
        }
    }
}
```

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

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

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

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

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

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

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

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

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

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

in the new customer count value appearing in the Text composable.

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

47.7 ViewModel implementation using LiveData

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

```
class MyViewModel : ViewModel() {

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

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

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

47.8 Observing ViewModel LiveData within an activity

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

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

}
```

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

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

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

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

47.9 Take the knowledge test

Click the link below or scan the QR code to test your knowledge and understanding of view models in Compose:

https://www.answertopia.com/dooi

47.10 Summary

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

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z