1

Getting Started with Modern Android Development Skills

The Android **Operating System** (**OS**) is one of the most favored platforms for mobile devices, with many users worldwide. The OS is used in cars and wearables such as smart watches, TVs, and phones, which makes the market quite wide for Android developers. Hence, there is a need for new developers to learn how to build Android applications utilizing new **Modern Android Development** (**MAD**) skills.

Android has come a long way since being launched in 2008 and used in the first **Integrated Development Environments** (**IDEs**), Eclipse and NetBeans. Today, Android Studio is the recommended IDE for Android development and, unlike before, when Java

was the preferred language, Kotlin is now the language of choice.

Android Studio includes support for Kotlin, Java, C++, and other programming languages, making this IDE suitable for developers with different skill sets.

Hence, by the end of this chapter, following the recipes, you will have Android Studio installed, have built your first Android application using Jetpack Compose, and have learned some Kotlin syntax, utilizing the preferred language for Android development. In addition, this introduction will prepare the base for you to understand advanced material that will be crucial for MAD.

In this chapter, we'll be covering the following recipes:

- Writing your first program in Kotlin using variables and idioms
- Creating a Hello, Android Community app using Android Studio
- Setting up your emulator in Android Studio
- Creating a button in Jetpack Compose
- Utilizing **`gradlew`** commands to clean and run your project in

Android Studio
- Understanding the Android project structure
- Debugging and logging in Android Studio

Technical requirements

Running the Android IDE and an emulator successfully can be daunting for your computer. You may have heard the joke about how machines running Android Studio can be used as heaters in winter. Well, there is some truth in that, so your computer should have the following specifications to ensure your system can cope with the IDE's demands:

- 64-bit Microsoft Windows, macOS, or Linux installed along with a stable internet connection. Recipes in this book have been developed within macOS. You can also use a Windows or Linux laptop, as there is no difference between using either.
- For Windows and Linux users, you can follow this link to install Android Studio: **https://developer.android.com/studio/install**.
- Minimum of 8 GB of RAM or more.

- Minimum of 8 GB of available disk space for Android Studio, the Android **Software Development Kit** (**SDK**), and the Android Emulator.
- A minimum screen resolution of 1280 x 800 is preferred.
- You can download Android Studio at **https://developer.android.com/studio**.

The complete source code for this chapter is available on GitHub at: **https://github.com/PacktPublishing/Modern-Android-13-Development-Cookbook/tree/main/chapter_one**

Writing your first program in Kotlin using variables and idioms

Kotlin is the recommended language for Android development; you can still use Java as your language of choice, as many legacy applications still heavily rely on Java. However, in this book, we will use Kotlin, and if this is the first time you are building Android applications using the Kotlin language, the Kotlin organization has excellent resources to help you get started with free practice exercises and self-paced assess-

ments called **Kotlin Koans** (**https://play.kotlinlang.org/koans/overview**).

In addition, you can use the Kotlin language for multiplatform development using **Kotlin Multiplatform Mobile** (**KMM**), in which you can share standard code between iOS and Android apps and write platform-specific code only where necessary. KMM is currently in Alpha.

Getting ready

In this recipe, you can either use the online Kotlin playground (**https://play.kotlinlang.org/**) to run your code or run the code in your Android Studio IDE. Alternatively, you can download and use the IntelliJ IDEA IDE if you plan on doing more Kotlin practice questions with Koans.

How to do it...

In this recipe, we will explore and modify a simple program that we will write in Kotlin; you can think of a program as instructions we give a computer or mobile devices to perform actions that we give them. For instance, we will create a greeting in our program and later write a different program.

For this recipe, you can choose either Android Studio or the free online IDE since we will touch on some Kotlin functionalities:

1. If you opt to use the Kotlin online playground for the first time, you will see something like the following screenshot, with a **`println`** statement that says **`Hello, world`**, but for our example, we will change that greeting to *Hello, Android Community*, and run the code.

Figure 1.1 – The online Kotlin editor

2. Let's look at another example; a popular algorithm problem used in interviews – reversing a string. For example, you have a string, **`Community`**, and we want to reverse the string so that the output will be **`ytinummoC`**. There are several ways to solve this problem, but we will solve it using the Kotlin idiomatic way.
3. Input the following code in the playground of your IDE or the Kotlin playground:

```
fun main() {
    val stringToBeReversed = "Community"
```

```
        println(reverseString(stringToBeReversed))
    }
    fun reverseString(stringToReverse: String): String {
        return stringToReverse.reversed()
    }
```

How it works...

It is essential to mention in Kotlin, that there are unique ways to keep your code cleaner, more precise, and simpler by taking advantage of the default parameter value and only setting the parameters you need to alter.

`fun` is a word in Kotlin programming language that stands for *function*, and a function in Kotlin is a section of a program that performs a specific task. The name of the function in our first example is **`main()`**, and in our **`main()`** function, we do not have any inputs. Functions, in general, have names so that we are able to distinguish them from each other if our code base is complex.

In addition, in Java, a function is similar to a method. The function name has two parentheses and curly braces, and **`println`**, which tells the system to print a line of text.

If you have used Java, you might notice that the Kotlin programming language

is very similar to Java. However, developers now talk about how great Kotlin language is for developers because it provides more expressive syntax and sophisticated type systems and handles the Null pointer problem that Java had for many years. To take full advantage of the Kotlin language power and write more concise code, knowing about Kotlin idioms can be beneficial. Kotlin idioms are frequently used collections that help to manipulate data and make Android developers' experience more effortless.

In our second example, we have two functions, **`main()`** and **`reverseString()`**. **`main()`** takes nothing in its input, but **`reverseString()`** does take in a String input. You will also notice we use **`val`**, which is a unique word used by Kotlin to refer to an immutable value that can only be set to one value, as compared to **`var`**, which is a mutable variable, meaning it can be resigned.

We create a **`val stringToBeReversed`** which is a String and call it **`"Community"`**, then call **`println`** inside the **`main()`** function, and pass in the text we want to print in our **`reverseString()`** function. Furthermore, in this example, our **`reverseString`** function takes in a **`string`** argument from the

String object and then returns a string type.

Figure 1.2 – The reversed string on the Kotlin playground

There is more to learn, and it is fair to acknowledge that what we have covered in this recipe is just a tiny part of what you can do with Kotlin idioms. This recipe aimed to introduce concepts that we might touch on or use in later chapters, not in depth, however, since we will explore more of Kotlin in later chapters. Hence, it's good to know what Kotlin idioms are and why they are essential for now.

See also

A better understanding of the Kotlin syntax and popular use cases will be vital for your day-to-day work, so look at the following resources:

- The JetBrains Academy has a great free Kotlin Basics course here: **https://hyperskill.org/tracks/18**.

- The Kotlin documentation is also a great resource to keep handy: **https://kotlinlang.org/docs/home.html**.

Creating a Hello, Android Community app using Android Studio

We will create our first Android application now that we have installed Android Studio. In addition, we will use Compose – just to mention in advance, in this recipe, we will not go in depth about Compose, as we have a dedicated chapter on Compose, which is *Chapter 2*, *Creating Screens Using a Declarative UI and Exploring Compose Principles*.

Getting ready

Before you begin, it's helpful to know where your Android projects are for consistency. By default, Android Studio creates a package in your home directory, and the package name is **`AndroidStudioProjects`**; here, you will find all the projects you create.

You can also decide where the folder should be if you want to change it. In addition, ensure you are using the latest version of Android Studio to utilize all the great features. To find out what the latest Android version is, you can use the following link: **https://developer.android.com/studio/releases**.

How to do it...

In the Android Studio IDE, a project template is an Android app that has all the necessary parts to create an application and helps you get started and set up.

So, step by step, we will make our first Android application and launch it on the emulator:

1. Start Android Studio by clicking on the Android Studio icon in your dock or wherever you have stored Android Studio.
2. You will see a welcome Android Studio window open up, and you can click on **New Project**. Alternatively, you can go to **File** and click **New Project**.
3. Select **Empty Compose Activity** and click on **Next**.

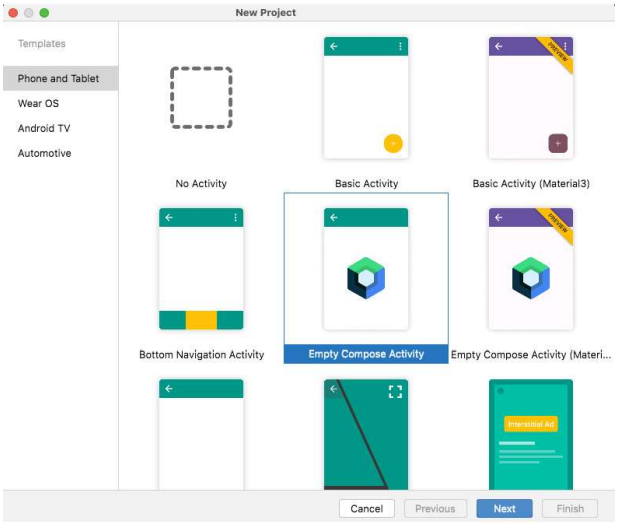

Figure 1.3 – Creating an empty Compose activity

4. Once the empty Compose activity screen loads (*Figure 1.4*), you will see fields including **Name**, **Package name**, **Save Location**, **Language**, and **Minimum SDK**. For this chapter, you can name the project `Android Community` and leave the other settings as is. You will also notice that the language is **Kotlin** by default.

As for **Minimum SDK**, our target is **API 21: Android 5.0 (Lollipop)**, which indicates the minimum version of Android that your app can run, which, in our case, is approximately 98.8% of devices. You can also click on the dropdown and learn more about the minimum SDK.

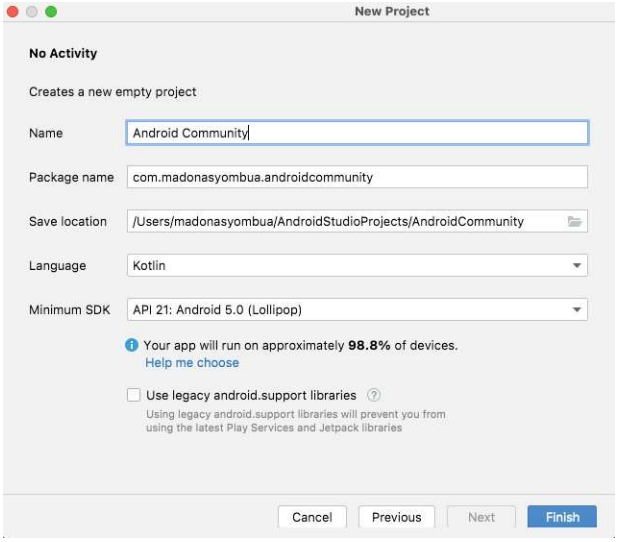

Figure 1.4 – Naming your empty Compose activity

Click **Finish** and wait for Gradle to sync.

5. Go ahead and play around with the packages, and you will notice a **`MainActivity`** class that extends a **`ComponentActivity()`** which extends **`Activity()`**; inside, we have a **`fun onCreate`**, which is an override from the **`ComponentActivity`**. You will also see a **`setContent{}`**, which is a function used to set the content of a Composable function. The **`setContent{}`** function takes a lambda expression that contains the UI elements that should be displayed, and in our case it holds the theme of our application. In the **`Greeting()`**function, we will change what is provided and add our own greeting which is **`"Hello, Android`**

Community" and run, and we will have created our first **Greeting**:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?)
    {
        super.onCreate(savedInstanceState)
        setContent {
            AndroidCommunityTheme {
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color =
                    MaterialTheme.colors.background
                ) {
                    Greeting("Hello, Android
                              Community")
                }
            }
        }
    }
```

6. Let's go ahead and modify the **Greeting()** function and assign the **name** argument to the text:

```
@Composable
fun Greeting(name: String) {
    Text(
        text = name
    )
}
```

In addition, you can also just pass **"Hello, Android Community"** into the default implementation, and this will produce the same UI.

7. Like in an XML view, you can easily view the UI you are building without running the app in an emulator using **`@Preview(showBackground = true)`**, so let's go ahead and add this to our code if it is not available. By default, the project comes with a template that has a **`Preview()`**:

```
@Preview(showBackground = true)
@Composable
fun DefaultPreview() {
}
```

8. Finally, when you run the application, you should have a screen like in *Figure 1.5*. In the following recipe, we will look at how you can set up your emulator step by step, so do not worry about that yet.

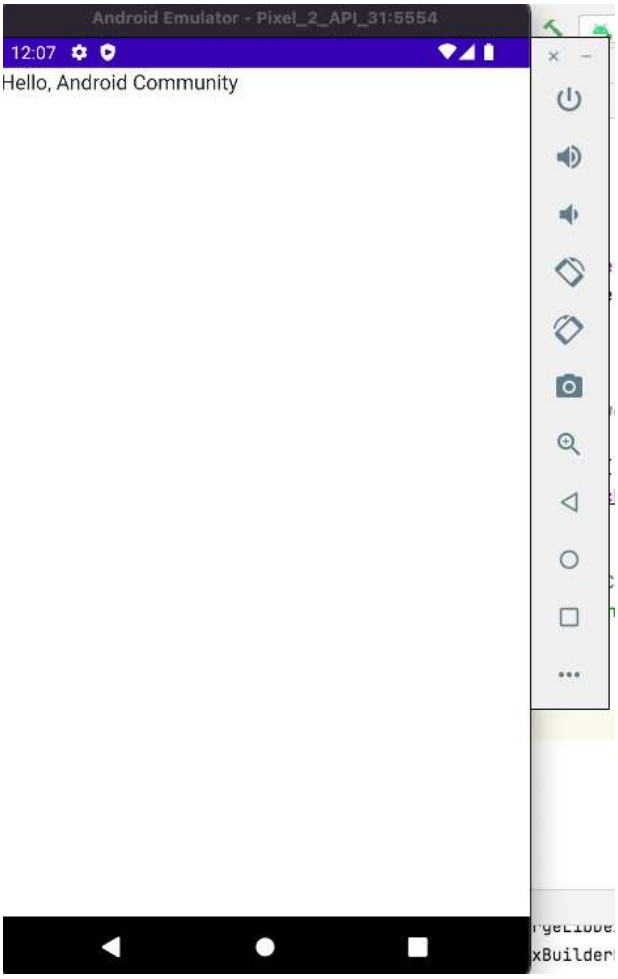

Figure 1.5 – Screen displaying Hello, Android Community

How it works...

The key benefits of using Jetpack Compose for creating your view are that it speeds up the development time since you use the same language to write your entire code base (Kotlin) and it is easier to test. You can also create reusable components that you can customize to your needs.

Therefore, ensuring lower chances of errors and having to write views with XML because it is tedious and cumber-

some. The `onCreate()` function is considered the entry point to the application in Android. Furthermore, we use `modifier` functions to add behavior and decorate the composable. We'll talk more about what `modifier` and `Surface` functions can do in the next chapter.

Setting up your emulator in Android Studio

Android Studio is a reliable and mature IDE. As a result, Android Studio has been the favored IDE for developing Android applications since 2014. Of course, you can still use other IDEs, but the advantage of Android Studio is that you do not need to install the Android SDK separately.

Getting ready

You need to have done the previous recipe to be able to follow along with this recipe since we will be setting up our emulator in order to run the project we just created.

How to do it...

This chapter seeks to be friendly to beginners and also move you smoothly to-

ward more advanced Android as you work through the recipes.

Let's follow these steps to see how you can set up your emulator and run your project in the *Creating a Hello, Android Community App using Android Studio* recipe:

1. Navigate to **Tools | Device Manager**. Once the device manager is ready, you have two options: **Virtual** or **Physical**. **Virtual** means you will be using an emulator, and **Physical** means you will be enabling your Android phone to debug Android applications. For our purposes, we will choose **Virtual**.
2. Click on **Create device**, and the **Virtual Device Configuration** screen will pop up.
3. Pick **Phone**. You will notice Android Studio has other categories, such as **TV**, **Wear OS**, **Tablet**, and **Automotive**. Let's use **Phone** for now, and in a future chapter, we will try using **Wear OS**. Click on **Next**.

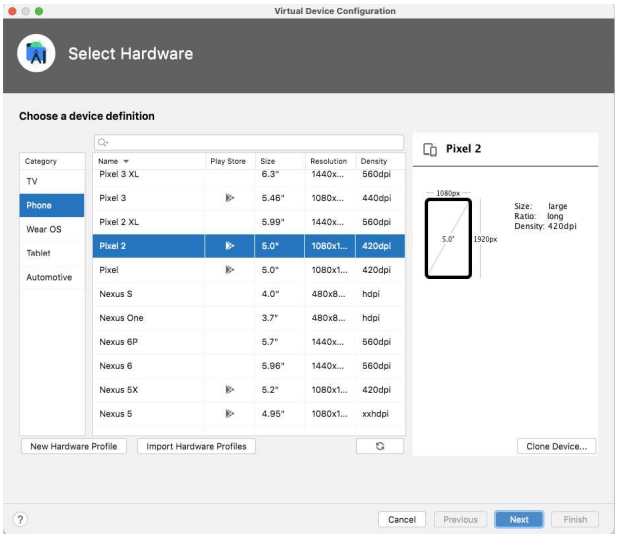

Figure 1.6 – Selecting a virtual device

4. In *Figure 1.7*, you will see a list of **Recommended system images**. You can choose any or use the default one, which is **S** in our case for Android 12, although you might want to use the latest API, **33**, and then click **Next**.

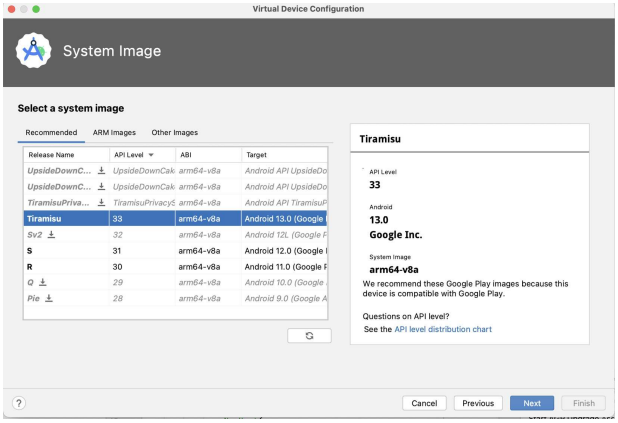

Figure 1.7 – Selecting a system image

5. You will now arrive at the **Android Virtual Device** (**AVD**) screen, where you can name your virtual device. You can enter a name or just leave the default, **Pixel 2 API 31**, and then hit **Finish**.

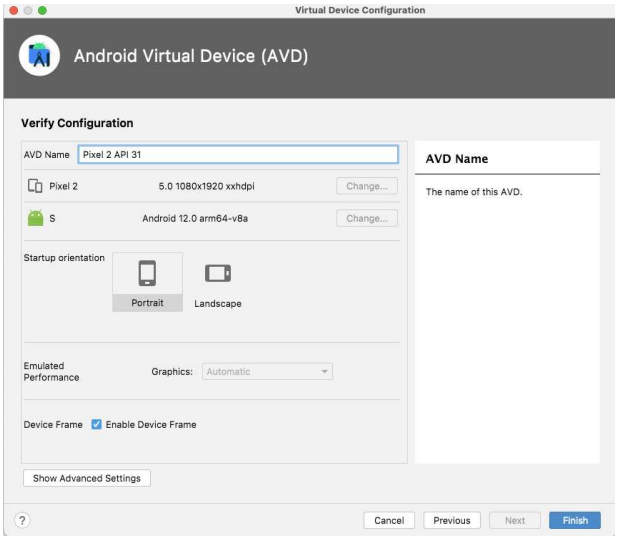

Figure 1.8 – Final steps of setting up the AVD

6. Test your virtual device by running it and ensure it works as expected. You should see something similar to *Figure 1.9* once you run your application on the emulator.

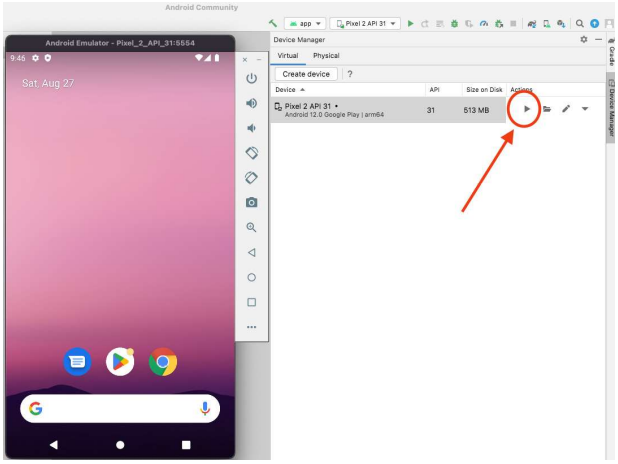

Figure 1.9 – The Device Manager section to run the emulator

IMPORTANT NOTE

To create a physical testing device, you must go to **Settings** *on your Android phone and select* **About phone** |

Software information | Build number *and hold down the button as you release until you see* **You are now four steps away from being a developer**. *Once the count is complete, you will see a notification saying* **Developer options successfully enabled**. *All you need now is to use a* **Universal Serial Bus** *(***USB***) and toggle USB debugging. Finally, you will see that your physical phone is ready for testing.*

How it works...

Testing and ensuring your applications display the expected outcome is very important. That is why Android Studio uses the emulator to help developers ensure their application functions as it would on standard devices. Furthermore, Android phones come with a developer's option ready for developers to use, which makes it even easier for the different number of devices that Android supports and also for helping reproduce bugs that are hard to find in emulators.

Creating a button in Jetpack Compose

We must note that we cannot cover all views in just one recipe; we have a chapter dedicated to learning more about Jetpack Compose, so in the project we have created, we will just try to create two more additional views for our project.

Getting ready

Open the **Android Community** project, as that is the project we will be building upon in this recipe.

How to do it...

Let's start by implementing a simple button in Compose:

1. Let's go ahead and organize our code and align the text to the center by adding a **`Column()`** to organize our views. This should be added to the **`setContent{}`** function:

```
Column(
    modifier = Modifier
        .fillMaxSize()
        .wrapContentSize(Alignment.Center),
    horizontalAlignment = Alignment.CenterHorizontally
) {
    Greeting("Hello, Android Community")
  }
}
```

2. Now, create a function and call it **`SampleButton`**; we will pass nothing

in this example. However, we will have a **`RowScope{}`**, which defines the **`modifier`** functions applicable to our button in this case, and we will give our button a name: **`click me`**.

3. In Compose, when you create a button, you can set its shape, icon, and elevation, check whether it is enabled or not, check its content, and more. You can check how to customize your button by command-clicking on the **`Button()`** component:

```
@Composable
fun SampleButton() {
    Button(
        onClick = { /*TODO*/ },
        modifier = Modifier
            .fillMaxWidth()
            .padding(24.dp),
        shape = RoundedCornerShape(20.dp),
        border = BorderStroke(2.dp, Color.Blue),
        colors = ButtonDefaults.buttonColors(
            contentColor = Color.Gray,
            backgroundColor = Color.White
        )
    ) {
        Text(
            text = stringResource(id =
                       R.string.click_me),
            fontSize = 14.sp,
            modifier = Modifier.padding(horizontal =
                           30.dp, vertical = 6.dp)
        )
    }
}
```

In our **`SampleButton`**, **`onClick`** does not do anything; our button has a modifier of the maximum fill width, padding of 24 **density-independent pixels** (**dp**), and round corners with a radius of 20 dp.

We have also set the button's color and added **`click me`** as text. We set our font size to 14 **scale-independent pixels** (**sp**), as this helps in ensuring that the text will adjust well for both the screen and users' preferences.

4. Also, click **Split** in the top right to preview your screen elements, or you can click on the **Design** section to view the entire screen without the code.

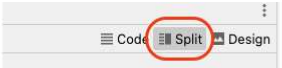

Figure 1.10 – View both the code and design in Android Studio

5. Finally, let's call our **`SampleButton`** function, where the **`Greeting`** function is, and run the app:

```
Column(
    modifier = Modifier
        .fillMaxSize()
        .wrapContentSize(Alignment.Center),
    horizontalAlignment = Alignment.CenterHorizontally
) {
    Greeting("Hello, Android Community")
```

```
        SampleButton()
    }
```

6. Compile and run the program; your app should look similar to *Figure 1.11*.

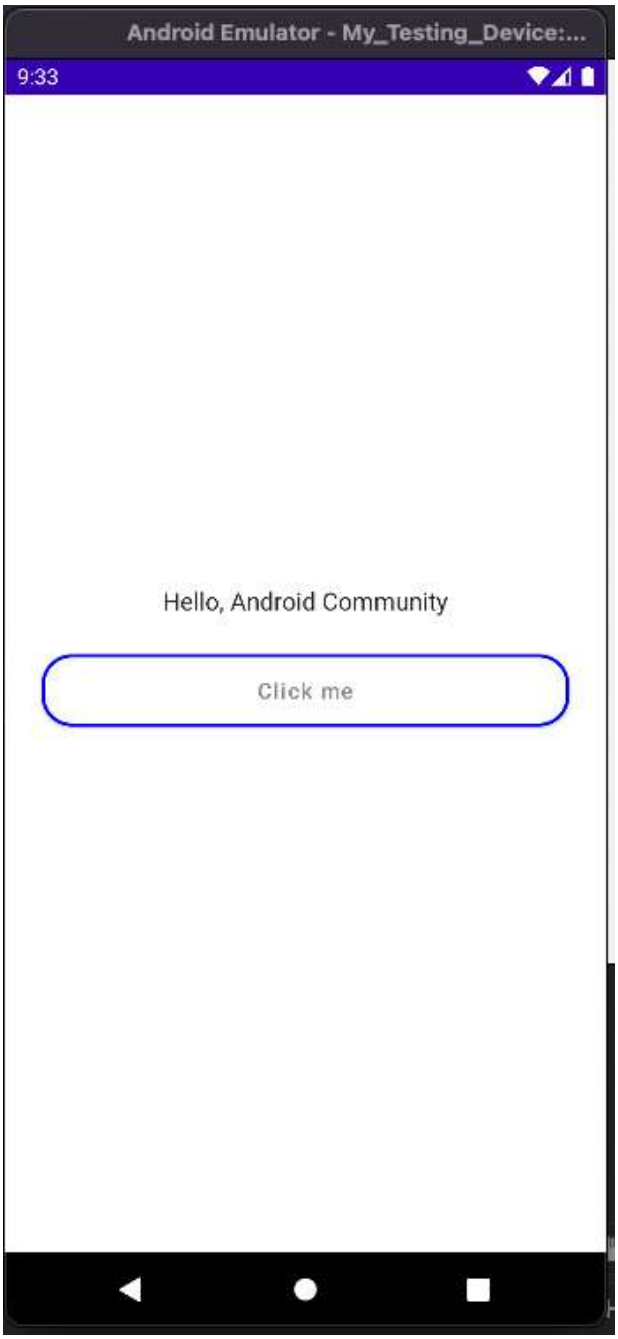

Figure 1.11 – A screenshot showing the text and a button

How it works...

A Composable app comprises several composable functions, just normal functions annotated with **`@Composable`**. As Google documentation explains, the annotation tells the Compose to add exceptional support to the procedure for updating and maintaining your UI over time. Compose also lets you structure your code into small maintainable chunks that you can adjust and reuse at any given point.

There's more...

Since it's hard to cover all views in a single recipe, we will work on more views in *Chapter 2*, *Creating Screens Using a Declarative UI and Exploring Compose Principles*, explore the best practices, and test our composable views.

Utilizing gradlew commands to clean and run your project in Android Studio

The **`gradlew`** command is a robust Gradle wrapper that has excellent usage. In Android Studio, however, you do not need to install it because it is a

script that comes packaged within the project.

Getting ready

For now, however, we will not look into all the Gradle commands but instead use the most popular ones to clean, build, provide info, debug, and scan our project to find any issues when we run our application. You can run the commands in your laptop's terminal as long as you are in the correct directory or use the terminal provided by Android Studio.

How to do it...

Follow these steps to check and confirm whether Gradle works as anticipated:

1. You can check the version by simply running **`./gradlew`**.

Figure 1.12 – gradlew version

2. To build and clean your project, you can run the **`./gradlew clean`** and **`./gradlew build`** commands. If any-

thing is wrong with your project, the build will fail, and you can investigate the error. In addition, in Android, you can always run your project without using the Gradle commands, and just utilize the IDE run and clean options. We will discuss this topic in depth in *Chapter 12, Android Studio Tips and Tricks to Help You during Development*.

3. The following are a few more useful **`gradlew`** commands; for example, when your build fails and you want to know what went wrong, use the commands to investigate or click on the error message (see *Figure 1.13*):
    1. Run with the **`--stacktrace`** option to get the stack trace
    2. Run with the **`--info`** or **`--debug`** option to get more log output
    3. Run with **`--scan`** to get full insights

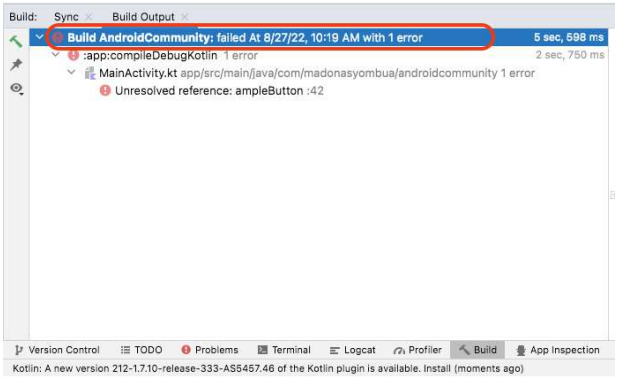

Figure 1.13 – Build error output

How it works

Gradle is a general-purpose build tool that proves to be very powerful in Android development. In addition, you can create and publish your custom plugins to encapsulate your conventions and build functionality. Advantages of Gradle include incremental build works for test execution, compilation, and any other task that happens in your build system.

See also

More about Gradle and what it does can be found here: **https://gradle.org/**.

Understanding the Android project structure

If this is your first time looking at the Android project folder, you might wonder where to add your code and what the packages mean. This recipe will walk through what each folder holds and what code goes where.

Getting ready

If you open your project, you will notice many folders. The main folders in your Android project are listed here:

- The `manifest` folder
- The `java` folder (`test`/`androidTest`)
- The `Res Resource` folder
- `Gradle Scripts`

How to do it...

Let's navigate through each folder as we learn what is stored, where, and why:

1. In *Figure 1.14*, you can see the **Packages** dropdown; click on that, and a window with **Project**, **Packages**, **Project Files**, and more will pop up.
2. You can opt to view your project using the Android logo, via **Project**, or the `Project` highlighted section next to the drop-down menu. The Project view is best when you have many modules in your application and want to add specific code. Feel free to click on the sections and see what they hold.

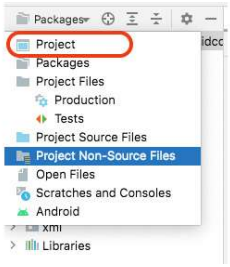

Figure 1.14 – Android Studio project structure

3. The **`manifest`** folder is the source of truth for the Android application; it contains **`AndroidManifest.xml`**. Click inside the file, and you will notice you have an intent launcher that launches the Android application on your emulator.
4. In addition, the version number is typically set in Gradle and then merged into **`manifest`** and in the manifest is where we add all needed permissions. You will also notice the package name, metadata, data extraction rules, theme, and icon; if you have a unique icon, you can add one here.

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

    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="AndroidCommunity"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.AndroidCommunity"
        tools:targetApi="31">
        <activity
            android:name=".MainActivity"
            android:exported="true"
            android:label="AndroidCommunity"
            android:theme="@style/Theme.AndroidCommunity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Figure 1.15 – Android Studio project structure manifest file

IMPORTANT NOTE

Making your icon adaptive is the new favored way to add icons to your applica-

tions. Adaptive icons display differently depending on individual user theming in MAD. See **https://developer.android.com/develop/ui/views/launch/icon_design_adaptive**.

5. The **`java`** folder contains all the Kotlin (**`.kt`**) and Java (**`.java`**) files we create as we build our Android applications. For example, in *Figure 1.16*, we have a package with **(androidTest)** and **(test)**, and this is where we add our tests. Go ahead and click on all the folders and see what they contain.

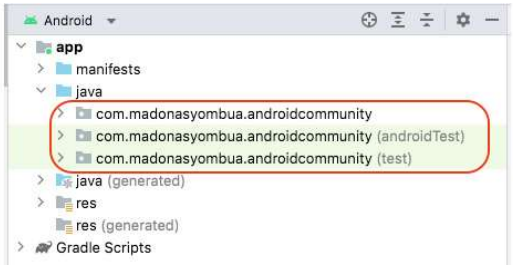

Figure 1.16 – Android Studio project structure Java folders

6. In the **`androidTest`** folder, we write our UI tests to test the UI functionalities, and in the test folder, we write our unit test. Unit testing tests small pieces of our code to ensure the required behavior is as anticipated. **Test-Driven Development** (**TDD**) is excellent and valuable during app development. Some companies follow this rule, but some do not en-

force it. However, it is a great skill to have, as it is good practice to always test your code.

The **`res`** folder contains XML layouts, UI strings, drawable images, and Mipmap icons. On the other hand, the **`values`** folder contains many useful XML files such as **`dimensions`**, **`colors`**, and **`themes`**. Go ahead and click on the **`res`** folder to get familiar with what is there, as we will use it in the next chapter.

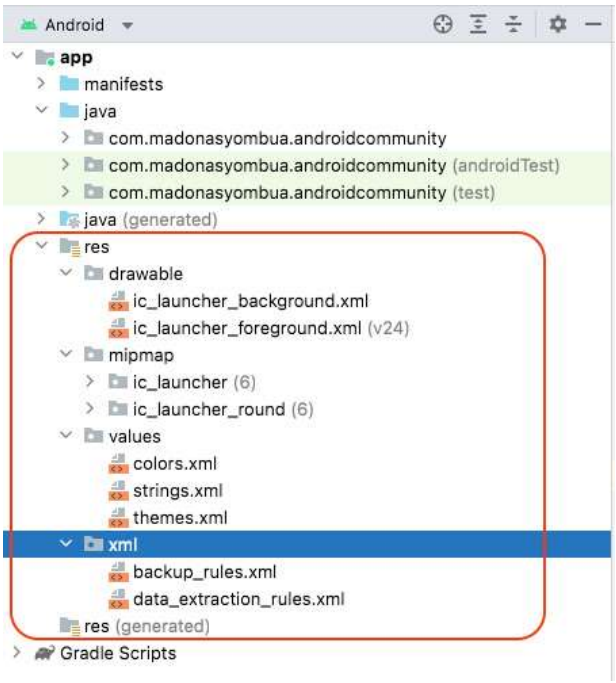

Figure 1.17 – Android Studio project structure res folder

IMPORTANT NOTE

Unless you are building a new project from scratch, many applications still use XML layouts, and developers opt to develop new screens with Jetpack Compose as an advancement now. Therefore, you

might have to maintain or know how to write views in XML.

7. Finally, in **`Gradle Scripts`**, you will see the files that define the build configuration we can apply in our modules. For example, in **`build.gradle(Project: AndroidCommunity)`**, you will see a top-level file where you can add configuration options common to all your sub-project modules.

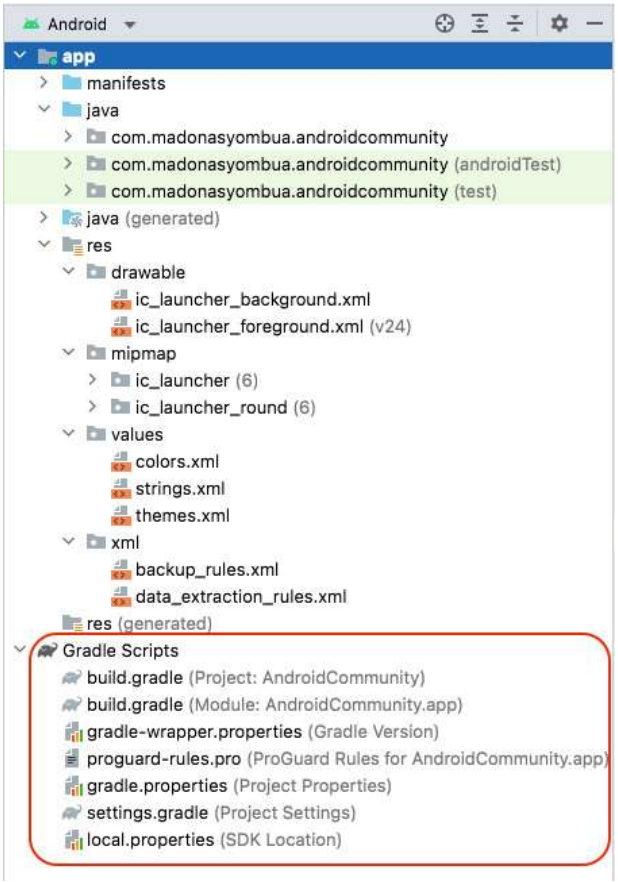

Figure 1.18 – Gradle scripts in the Android Studio project structure

How it works...

In Android Studio, it can be overwhelming for first-time users not to know where files go and what is essential. Hence, having a step-by-step guide on where to add your tests or code and understanding the Android project structure is vital. In addition, in complex projects, you might find different modules; hence, understanding the project structure is helpful. A module in Android Studio is a collection of source files and build settings that allow you to divide your project into distinct entities with specific purposes.

Debugging and logging in Android Studio

Debugging and logging are crucial in Android development, and you can write log messages that appear in Logcat to help you find issues in your code or verify a piece of code executes when it should.

We will introduce this topic here, but it is unfair to say we will cover it all in just one recipe; for that reason, we will cover more about debugging and logging in *Chapter 12, Android Studio Tips and Tricks to Help You during Development*.

Getting ready

Let us use an example to understand logging. The following log methods are listed from the highest to lowest priority. They are proper when logging network errors, success calls, and other errors:

- **`Log.e()`**: A log error
- **`Log.w()`**: A log warning
- **`Log.i()`**: Log information
- **`Log.d()`**: Debugging shows critical messages to developers, the most used log
- **`Log.v()`**: Verbose

A good practice is associating every log with a **`TAG`** to identify the error message in Logcat quickly. A "**`TAG"`** refers to a text label that can be assigned to a View or other UI element in an Android application. The primary purpose of using **tags** in Android is to provide a way to associate additional information or metadata with a UI element.

How to do it...

Let's go ahead and add a log message to our small project:

1. We will go ahead and create a debug log, then run the application:

```
Log.d(TAG, "asdf Testing call")
```

In the **Logcat** section, in the search box, enter **`asdf`** and see whether you can find the message.

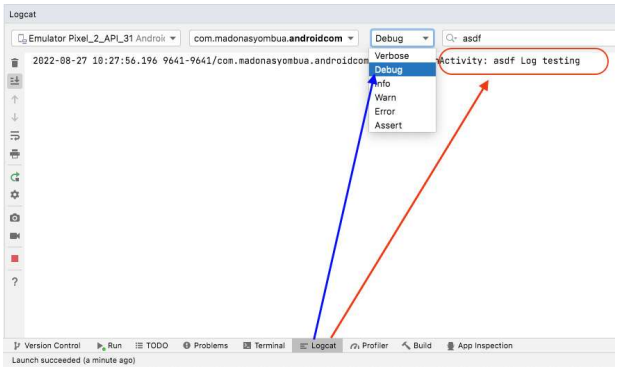

Figure 1.19 – Android Logcat

You will notice that the log has a class name, our **`TAG`** (**`MainActivity`**), and the log message displayed; see the right arrow in *Figure 1.19*.

2. The left arrow shows the mentioned log types and using the dropdown, you can quickly view your message based on the specification.

How it works...

Debugging is when you put breakpoints in your classes, slow down your emulator, and try to find issues in your code. Debugging is very powerful if, for instance, you encounter a race condition in your code or if your code works on some devices and does not work on others.

In addition, to take advantage of debugging, you need first to attach a debugger to the emulator, and then run in **Debug** mode. **Logging**, on the other hand, helps you log information that might be helpful to you when you encounter issues. Sometimes, debugging can be challenging, but placing logs where needed in your code might be very helpful.

A practical case is when you are loading data from an API; you might want to log it when there is a network error to inform you what happens if the network call fails. Hence, debugging using breakpoints might help slow down the process as you evaluate the values, and since we did not build a lot in this chapter, we can revisit this topic in a different recipe in later chapters.

See also

Timber is a logger with a small, extensible API that provides utility on top of Android's standard **Log** class, and many developers use it for logging. For more information about Timber, see **https://github.com/JakeWharton/timber**.