Chapter 1: Building Your First Compose App

When Android was introduced more than 10 years ago, it quickly gained popularity among developers because it was incredibly easy to write apps. All you had to do was define the user interface (UI) in an XML file and connect it to your *activity*. This worked flawlessly because apps were small and developers needed to support just a handful of devices.

So much has changed since then.

With every new platform version, Android gained new features. Through the years, device manufacturers introduced thousands of devices with different screen sizes, pixel densities, and form factors. While Google did its best to keep the Android *view* system comprehendible, the complexity of apps increased significantly; basic tasks such as implementing scrolling lists or animations require lots of boilerplate code.

It turned out that these problems were not specific to Android. Other platforms and operating systems faced them as well. Most issues stem from how UI toolkits used to work; they follow a so-called **imperative approach** (which I will explain in *Chapter 2*, *Understanding the Declarative Paradigm*). The solution was a paradigm shift. The web framework React was the first to popularize a declarative approach. Other platforms and frameworks (for example, Flutter and SwiftUI) followed.

Jetpack Compose is Google's declarative UI framework for Android. It dramatically simplifies the creation of UIs. As you will surely agree after reading this book, using Jetpack Compose is both easy and fun. But before we dive in, please note that Jetpack Compose is Kotlin-only. This means that all your Compose code will have to be written in Kotlin. To follow this book, you should have a basic understanding of the Kotlin syntax and

the functional programming model. If you want to learn more about these topics, please refer to the *Further reading* section at the end of this chapter.

This chapter covers three main topics:

- Saying hello to composable functions
- Using the preview
- Running a Compose app

I will explain how to build a simple UI with Jetpack Compose. Next, you will learn to use the **preview** feature in Android Studio and how to run a Compose app. By the end of this chapter, you will have a basic understanding of how composable functions work, how they are integrated into your app, and how your project must be configured in order to use Jetpack Compose.

Technical requirements

All the code files for this chapter can be found on GitHub at https://github.com/PacktPublishing/Android-UI-Development-with-Jetpack-Compose/tree/main/chapter_01. Please download the zipped version or clone the repository to an arbitrary location on your computer. The projects require at least Android Studio Arctic Fox. You can download the latest version at https://developer.android.com/studio. Please follow the detailed installation instructions at https://developer.android.com/studio/install.

To open this book's project, launch Android Studio, click the **Open** button in the upper-right area of the **Welcome to Android Studio** window, and select the base directory of the project in the folder selection dialog. Please make sure to not open the base directory of the repository, because Android Studio would not recognize the projects. Instead, you must pick the directory that contains the project you want to work with.

To run a sample app, you need a real device or the Android Emulator. Please make sure that developer options and USB debugging are enabled on the real device, and that the device is connected to your development machine via USB or WLAN. Please follow the instructions at https://developer.android.com/studio/debug/dev-options. You can also set up the Android Emulator. You can find detailed instructions at https://developer.android.com/studio/run/emulator.

Saying hello to composable functions

As you will see shortly, composable functions are the essential building blocks of Compose apps; these elements make up the UI.

To take a first look at them, I will walk you through a simple app called **Hello** (*Figure 1.1*). If you have already cloned or downloaded the repository of this book, its project folder is located inside **`chapter_01`**. Otherwise, please do so now. To follow this section, open the project in Android Studio and open **`MainActivity.kt`**. The use case of our first Compose app is very simple. After you have entered your name and clicked on the **Done** button, you will see a greeting message:

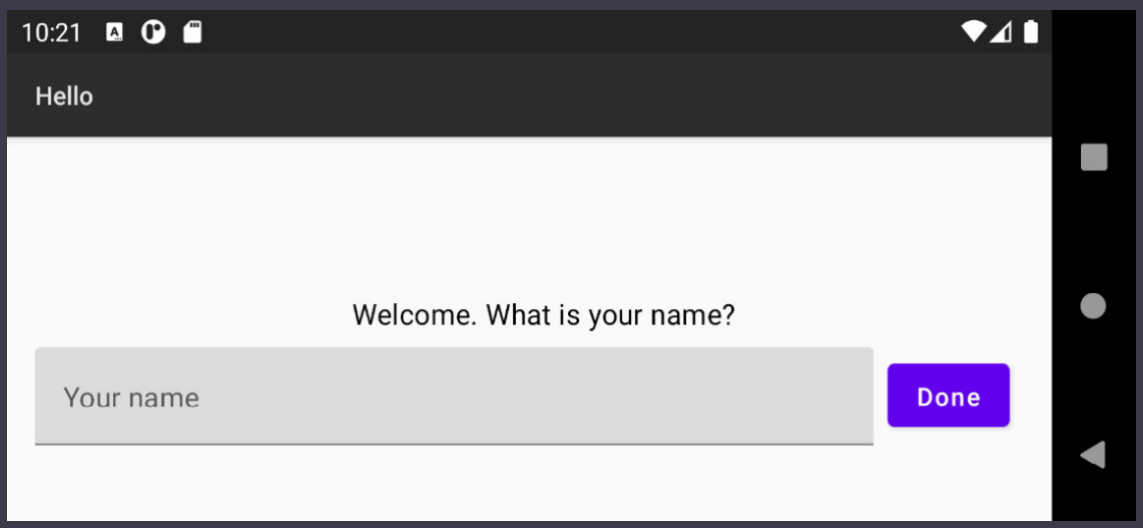

Figure 1.1 – The Hello app

Conceptually, the app consists of the following:

- The welcome text
- A row with an **`EditText`** equivalent and a button
- A greeting message

Let's take a look at how to create the app.

Showing a welcome text

Let's start with the welcome text, our first composable function:

```
@Composable
fun Welcome() {
  Text(
    text = stringResource(id = R.string.welcome),
    style = MaterialTheme.typography.subtitle1
  )
}
```

Composable functions can be easily identified by the **`@Composable`** annotation. They do not need to have a particular return type but instead emit UI elements. This is usually done by invoking other composables (for the sake of brevity, I will sometimes omit the word "function"). *Chapter 3*, *Exploring the Key Principles of Compose*, will cover this in greater detail.

In this example, **`Welcome()`** summons a text. **`Text()`** is a built-in composable function and belongs to the **`androidx.compose.material`** package.

To invoke **`Text()`** just by its name, you need to import it:

```
import androidx.compose.material.Text
```

Please note that you can save **`import`** lines by using the `*` wildcard.

To use **`Text()`** and other Material Design elements, your **`build.gradle`** file must include an implementation dependency to **`androidx.compose.material:material`**.

Looking back at the welcome text code, the **`Text()`** composable inside **`Welcome()`** is configured through two parameters, **`text`** and **`style`**.

The first, **`text`**, specifies what text will be displayed. **`R.string`** may look familiar; it refers to definitions inside the **`strings.xml`** files. Just like in view-based apps, you define text for UI elements there. **`stringResource()`** is a predefined composable function. It belongs to the **`androidx.compose.ui.res`** package.

The **`style`** parameter modifies the visual appearance of a text. In this case, the output will look like a subtitle. I will show you how to create your own themes in *Chapter 6*, *Putting Pieces Together*.

The next composable looks quite similar. Can you spot the differences?

```
@Composable
fun Greeting(name: String) {
  Text(
    text = stringResource(id = R.string.hello, name),
    textAlign = TextAlign.Center,
    style = MaterialTheme.typography.subtitle1
  )
}
```

Here, **`stringResource()`** receives an additional parameter. This is very convenient for replacing placeholders with actual texts. The string is defined in **`strings.xml`**, as follows:

```
<string name="hello">Hello, %1$s.\nNice to meet you.
</string>
```

The **`textAlign`** parameter specifies how text is positioned horizontally. Here, each line is centered.

Using rows, text fields, and buttons

Next, let's turn to the text input field (**Your name**) and the **Done** button, which both appear on the same row. This is a very common pattern, therefore Jetpack Compose provides a composable named **`Row()`**, which belongs to the **`androidx.compose.foundation.layout`** package. Just like all composable functions, **`Row()`** can receive a comma-separated list of parameters inside **`( )`** and its children are put inside curly braces:

```
@Composable
fun TextAndButton(name: MutableState<String>,
                  nameEntered: MutableState<Boolean>)
{
  Row(modifier = Modifier.padding(top = 8.dp)) {
    ...
  }
}
```

`TextAndButton()` requires two parameters, **`name`** and **`nameEntered`**. You will see what they are used for in the *Showing a greeting message* section. For now, please ignore their **`MutableState`** type.

`Row()` receives a parameter called **`modifier`**. Modifiers are a key technique in Jetpack Compose to influence both the look and behavior of composable functions. I will explain them in greater detail in *Chapter 3*, *Exploring the Key Principles of Compose*.

`padding(top = 8.dp)` means that the row will have a padding of eight density-independent pixels (**`.dp`**) at its upper side, thus separating itself from the welcome message above it.

Now, we will look at the text input field, which allows the user to enter a name:

```
TextField(
  value = name.value,
  onValueChange = {
    name.value = it
```

```
    },
    placeholder = {
      Text(text = stringResource(id = R.string.hint))
    },
    modifier = Modifier
      .alignByBaseline()
      .weight(1.0F),
    singleLine = true,
    keyboardOptions = KeyboardOptions(
      autoCorrect = false,
      capitalization = KeyboardCapitalization.Words,
    ),
    keyboardActions = KeyboardActions(onAny = {
      nameEntered.value = true
    })
  )
```

`TextField()` belongs to the **`androidx.compose.material`** package. The composable can receive quite a few arguments; most of them are optional, though. Please note that the previous code fragment uses both the **`name`** and **`nameEntered`** parameters, which are passed to **`TextAndButton()`**. Their type is **`MutableState`**. **`MutableState`** objects carry changeable values, which you access as **`name.value`** or **`nameEntered.value`**.

The **`value`** parameter of a **`TextField()`** composable receives the current value of the text input field, for example, text that has already been input. **`onValueChange`** is invoked when changes to the text occur (if the user enters or deletes something). But why is **`name.value`** used in both places? I will answer this question in the *Showing a greeting message* section.

RECOMPOSITION

Certain types trigger a so-called recomposition. For now, think of this as repainting an associated composable. **`MutableState`** *is such a type. If we change its value, the* **`TextField()`** *composable is redrawn or repainted.*

Please note that both terms are not entirely correct. We will cover recomposition in Chapter 3, Exploring the Key Principles of Compose.

Let's briefly look at the remaining code. With **`alignByBaseline()`**, we can nicely align the baselines of other composable functions in a particular **`Row()`**. **`placeholder`** contains the text that is shown until the user has entered something. **`singleLine`** controls whether the user can enter multiple lines of text. Finally, **`keyboardOptions`** and **`keyboardActions`** describe the behavior of the onscreen keyboard. For example, certain actions will set **`nameEntered.value`** to **`true`**. I will show you soon why we do this.

However, we need to take a look at the **`Button()`** composable first. It also belongs to the **`androidx.compose.material`** package:

```
Button(modifier = Modifier
  .alignByBaseline()
  .padding(8.dp),
  onClick = {
    nameEntered.value = true
  }) {
  Text(text = stringResource(id = R.string.done))
}
```

Some things will already look familiar. For example, we call **`alignByBaseline()`** to align the baseline of the button with the text input field, and we apply a padding of eight density-independent pixels to all sides of the button using **`padding()`**. Now, **`onClick()`** specifies what to do when the button is clicked. Here, too, we set **`nameEntered.value`** to **`true`**. The next composable function, **`Hello()`**, finally shows you why this is done.

Showing a greeting message

`Hello()` emits **`Box()`**, which (depending on **`nameEntered.value`**) contains either the **`Greeting()`** or a **`Column()`** composable that, in turn, includes **`Welcome()`** and **`TextAndButton()`**. The **`Column()`** composable is quite similar to **`Row()`** but arranges its siblings vertically. Like the latter one and **`Box()`**,

it belongs to the **`androidx.compose.foundation.layout`** package. **`Box()`** can contain one or more children. They are positioned inside the box according to the **`contentAlignment`** parameter. We will be exploring this in greater detail in the *Combining basic building blocks* section of *Chapter 4*, *Laying Out UI Elements*:

```
@Composable
fun Hello() {
  val name = remember { mutableStateOf("") }
  val nameEntered = remember { mutableStateOf(false)
}
  Box(
    modifier = Modifier
      .fillMaxSize()
      .padding(16.dp),
    contentAlignment = Alignment.Center
  ) {
    if (nameEntered.value) {
      Greeting(name.value)
    } else {
      Column(horizontalAlignment =
             Alignment.CenterHorizontally) {
        Welcome()
        TextAndButton(name, nameEntered)
      }
    }
  }
}
```

Have you noticed **`remember`** and **`mutableStateOf`**? Both are very important for creating and maintaining state. Generally speaking, state in an app refers to a value that can change over time. While this also applies to domain data (for example, the result of a web service call), state usually refers to something being displayed or used by a UI element. If a composable function has (or relies on) state, it is recomposed (for now, repainted

or redrawn) when that state changes. To get an idea of what this means, recall this composable:

```
@Composable
fun Welcome() {
    Text(
        text = stringResource(id = R.string.welcome),
        style = MaterialTheme.typography.subtitle1
    )
}
```

`Welcome()` is said to be stateless; all values that might trigger a recomposition remain the same for the time being. **`Hello()`**, on the other hand, is stateful, because it uses the **`name`** and **`nameEntered`** variables. They change over time. This may not be obvious if you look at the source code of **`Hello()`**. Please recall that both **`name`** and **`nameEntered`** are passed to **`TextAndButton()`** and modified there.

Do you recall that in the previous section I promised to explain why **`name.value`** is used in two places, providing the text to display and receiving changes after the user has entered something? This is a common pattern often used with states; **`Hello()`** creates and remembers state by invoking **`mutableStateOf()`** and **`remember`**. And it passes state to another composable (**`TextAndButton()`**), which is called **state hoisting**. You will learn more about this in *Chapter 5*, *Managing the State of Your Composable Functions*.

So far, you have seen the source code of quite a few composable functions but not their output. Android Studio has a very important feature called **Compose preview**. It allows you to view a composable function without running the app. In the next section, I will show you how to use this feature.

Using the preview

The upper-right corner of the Android Studio code editor contains three buttons, **Code**, **Split**, and **Design** (*Figure 1.2*):

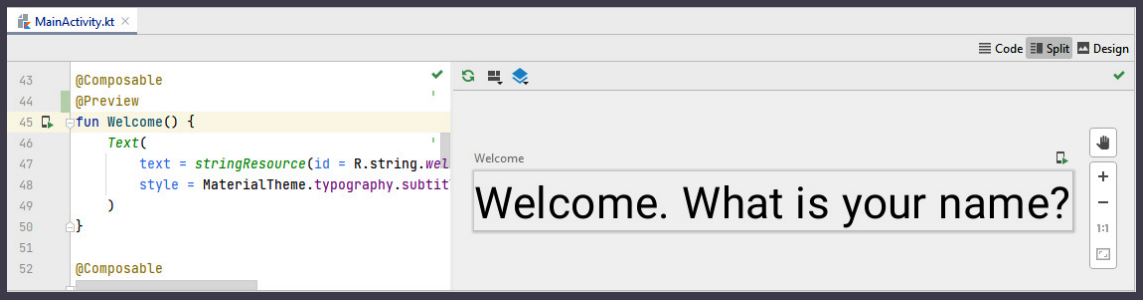

Figure 1.2 – Compose preview (Split mode)

They switch between the following different display modes:

- Code only
- Code and preview
- Preview only

To use the Compose preview, your composable functions must contain an additional annotation, **`@Preview`**, which belongs to the **`androidx.compose.ui.tooling.preview`** package. This requires an implementation dependency to **`androidx.compose.ui:ui-tooling-preview`** in your **`build.gradle`** file.

Unfortunately, if you try to add **`@Preview`** to **`Greeting()`**, you will see an error message like this:

```
Composable functions with non-default parameters are
not supported in Preview unless they are annotated
with @PreviewParameter.
```

So, how can you preview composables that take parameters?

Preview parameters

The most obvious solution is a wrapper composable:

```
@Composable
```

```
@Preview
fun GreetingWrapper() {
  Greeting("Jetpack Compose")
}
```

This means that you write another composable function that takes no parameters but invokes your existing one and provides the required parameter (in my example, a text). Depending on how many composable functions your source file contains, you might be creating quite a lot of boilerplate code. The wrappers don't add value besides enabling the preview.

Fortunately, there are other options. You can, for example, add default values to your composable:

```
@Composable
fun AltGreeting(name: String = "Jetpack Compose") {
```

While this looks less hacky, it alters how your composable functions can be invoked (that is, without passing a parameter). This may not be desirable if you had a reason for not defining a default value in the first place.

With **`@PreviewParameter`**, you can pass values to a composable that affect only the preview. Unfortunately, this is a little verbose, though, because you need to write a new class:

```
class HelloProvider :
PreviewParameterProvider<String> {
  override val values: Sequence<String>
    get() =
listOf("PreviewParameterProvider").asSequence()
}
```

The class must extend **`androidx.compose.ui.tooling.preview.PreviewParameterProvider`** because it will provide a parameter for the preview. Now, you can annotate the parameter of the composable with **`@PreviewParameter`** and pass your new class:

```
@Composable
@Preview
fun
AltGreeting2(@PreviewParameter(HelloProvider::class)
        name: String) {
```

In a way, you are creating boilerplate code, too. So, which method you choose in the end is a matter of personal taste. The **`@Preview`** annotation can receive quite a few parameters. They modify the visual appearance of the preview. Let's explore some of them.

Configuring previews

You can set a background color for a preview using **`backgroundColor =`**. The value is a **`Long`** type and represents an ARGB color. Please make sure to also set **`showBackground`** to **`true`**. The following snippet will produce a solid red background:

```
@Preview(showBackground = true, backgroundColor =
         0xffff0000)
```

By default, preview dimensions are chosen automatically. If you want to set them explicitly, you can pass **`heightDp`** and **`widthDp`**:

```
@Composable
@Preview(widthDp = 100, heightDp = 100)
fun Welcome() {
  Text(
    text = stringResource(id = R.string.welcome),
    style = MaterialTheme.typography.subtitle1
  )
}
```

Figure 1.3 shows the result. Both values are interpreted as density-independent pixels, so you don't need to add **`.dp`** as you would do inside your composable function.

Figure 1.3 – Setting the width and height of a preview

To test different user locales, you can add the **`locale`** parameter. If, for example, your app contains German strings inside **`values-de-rDE`**, you can use them by adding the following:

```
  @Preview(locale = "de-rDE")
```

The string matches the directory name after **`values-`**. Please recall that the directory is created by Android Studio if you add a language in the Translations Editor.

If you want to display the status and action bars, you can achieve this with **`showSystemUi`**:

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

To get an idea of how your composables react to different form factors, aspect ratios, and pixel densities, you can utilize the **`device`** parameter. It takes a string. Pass one of the values from **`Devices`**, for example, **`Devices.PIXEL_C`** or **`Devices.AUTOMOTIVE_1024p`**.

In this section, you have seen how to configure a preview. Next, I will introduce you to preview groups. They are very handy if your source code file contains more than a few composable functions that you want to preview.

Grouping previews

Android Studio shows composable functions with a **`@Preview`** annotation in the order of their appearance in the source code. You can choose between **Vertical Layout** and **Grid Layout** (*Figure 1.4*):

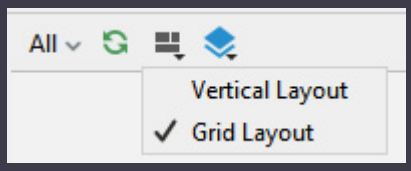

Figure 1.4 – Switching between Vertical Layout and Grid Layout

Depending on the number of your composables, the preview pane may at some point feel crowded. If this is the case, just put your composables into different groups by adding a **`group`** parameter:

```
  @Preview(group = "my-group-1")
```

You can then show either all composable functions or just those that belong to a particular group (*Figure 1.5*):

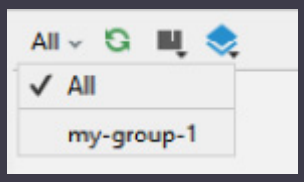

Figure 1.5 – Switching between groups

So far, I have shown you what the source code of composable functions looks like and how you can preview them inside Android Studio. In the next section, we will execute a composable on the Android Emulator or a real device, and you will learn how to connect composable functions to the other parts of an app. But before that, here is one more tip:

EXPORT A PREVIEW AS AN IMAGE

If you click on a Compose preview with the secondary mouse button, you will see a small pop-up menu. Select **Copy Image** *to put a bitmap of the pre-*

view on the system clipboard. Most graphics applications allow you to paste it into a new document.

Running a Compose app

If you want to see how a composable function looks and feels on the Android Emulator or a real device, you have two options:

- Deploying a composable function
- Running the app

The first option is useful if you want to focus on a particular composable rather than the whole app. Also, the time needed to deploy a composable may be significantly shorter than deploying a complete app (depending on the app size). So, let's start with this one.

Deploying a composable function

To deploy a composable function to a real device or the Android Emulator, click on the **Deploy Preview** button, which is a small image in the upper-right corner of a preview (*Figure 1.6*):

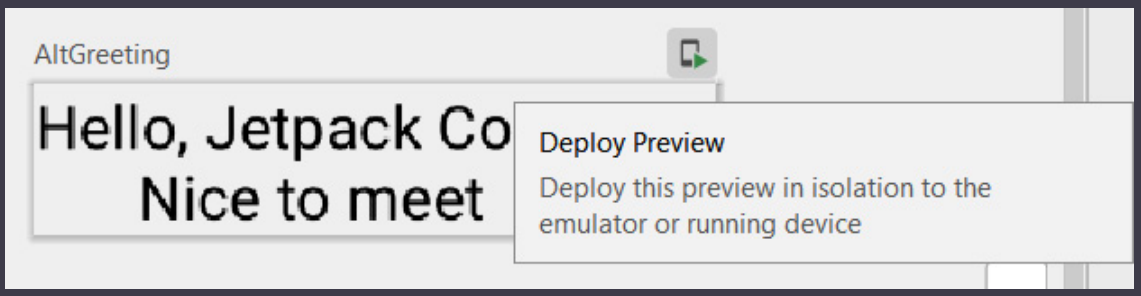

Figure 1.6 – Deploying a composable function

This will automatically create new launch configurations (*Figure 1.7*):

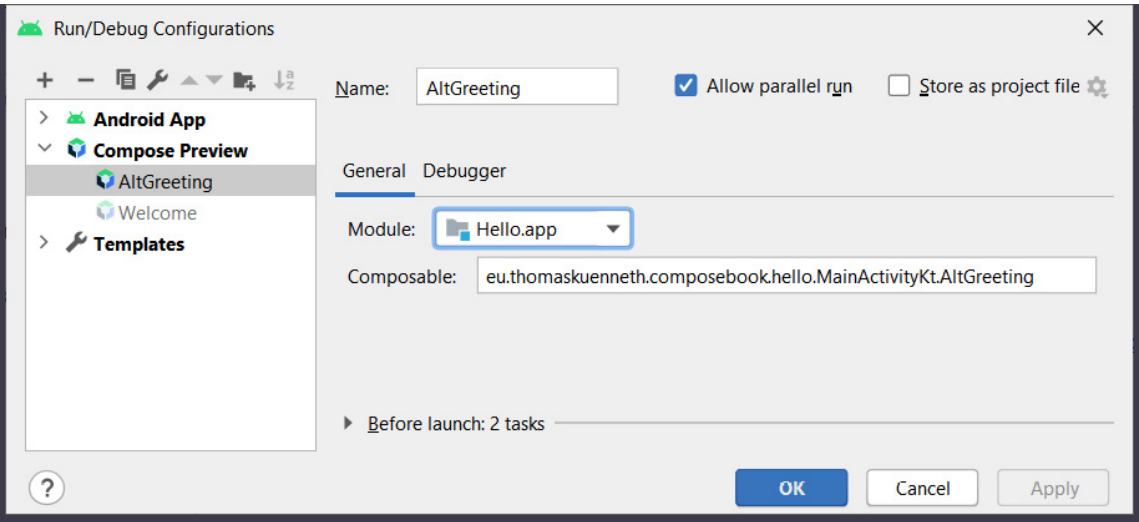

Figure 1.7 – Launch configurations representing Compose previews

You can modify or delete Compose preview configurations in the **Run/Debug Configurations** dialog. To access them, open the **Compose Preview** node. Then you can, for example, change its name or deny parallel runs by unchecking **Allow parallel run**.

The goal of this chapter is to deploy and run your first Compose app on a real device or the Android Emulator. You are almost there; in the next section, I will show you how to embed composable functions in an activity, which is a prerequisite. You will finally be running the app in the *Pressing the play button* section.

Using composable functions in activities

Activities have been one of the basic building blocks of Android apps since the first platform version. Practically every app has at least one activity. They are configured in the manifest file. To launch an activity from the home screen, the corresponding entry looks like this:

```
...
<activity
  android:name=".MainActivity"
  android:exported="true"
  android:label="@string/app_name">
  <intent-filter>
```

```
    <action android:name="android.intent.action.MAIN"
/>
    <category
     android:name="android.intent.category.LAUNCHER"
/>
  </intent-filter>
</activity>
...
```

This is still true for Compose apps. An activity that wishes to show composable functions is set up just like one that inflates a traditional layout file. But what does its source code look like? The main activity of the **`Hello`** app is called **`MainActivity`**, shown in the next code block:

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

As you can see, it is very short. The UI (the **`Hello()`** composable function) is displayed by invoking a function called **`setContent`**, which is an extension function to **`androidx.activity.ComponentActivity`** and belongs to the **`androidx.activity.compose`** package.

To render composables, your activity must extend either **`ComponentActivity`** or another class that has **`ComponentActivity`** as its direct or indirect ancestor. This is the case for **`androidx.fragment.app.FragmentActivity`** and **`androidx.appcompat.app.AppCompatActivity`**.

This is an important difference; while Compose apps invoke **`setContent()`**, View-based apps call **`setContentView()`** and pass either the ID of a layout

(**`R.layout.activity_main`**) or the root view itself (which is usually obtained through some binding mechanism). Let's see how the older mechanism works. The following code snippet is taken from one of my open source apps (you can find it on GitHub at https://github.com/MATHEMA-GmbH/TKWeek but it won't be discussed any further in this book):

```
class TKWeekActivity : TKWeekBaseActivity() {
  private var backing: TkweekBinding? = null
  private val binding get() = backing!!
  override fun onCreate(savedInstanceState: Bundle?)
{
    super.onCreate(savedInstanceState)
    backing = TkweekBinding.inflate(layoutInflater,
null,
              false)
    setContentView(binding.root)
    ...
```

If you compare both approaches, a striking difference is that with Jetpack Compose, there is no need for maintaining references to the UI component tree or individual elements of it. I will explain in *Chapter 2*, *Understanding the Declarative Paradigm*, why this leads to code that is easily maintainable and less error-prone.

Let's now return to **`setContent()`**. It receives two parameters, a **`parent`** (which can be **`null`**) and the **`content`** (the UI). The **`parent`** is an instance of **`androidx.compose.runtime.CompositionContext`**. It is used to logically link together two compositions. This is an advanced topic that I will be discussing in *Chapter 3*, *Exploring the Key Principles of Compose*.

IMPORTANT NOTE

Have you noticed that **`MainActivity`** *does not contain any composable functions? They do not need to be part of a class. In fact, you should implement them as top-level functions whenever possible. Jetpack Compose provides alternative means to access* **`android.content.Context`**. *You have already*

seen the **`stringResource()`** *composable function, which is a replacement for* **`getString()`**.

Now that you have seen how to embed composable functions in activities, it is time to look at the structure of Jetpack Compose-based projects. While Android Studio sets everything up for you if you create a Compose app using the project wizard, it is important to know which files are involved under the hood.

Looking under the hood

Jetpack Compose heavily relies on Kotlin. This means that your app project must be configured to use Kotlin. It does not imply, though, that you cannot use Java at all. In fact, you can easily mix Kotlin and Java in your project, as long as your composable functions are written in Kotlin. You can also combine traditional views and composables. I will be discussing this topic in *Chapter 9*, *Exploring Interoperability APIs*.

First, make sure to configure the Android Gradle plugin that corresponds to your version of Android Studio in the project-level build.gradle file:

```
buildscript {
  ...
  dependencies {
    classpath "com.android.tools.build:gradle:7.0.4"
    classpath "org.jetbrains.kotlin:kotlin-
gradle-              plugin:1.5.31"
    ...
  }
}
```

The following code snippets belong in the module-level build.gradle file:

```
plugins {
    id 'com.android.application'
    id 'kotlin-android'
```

```
  }
```

Next, please make sure that your app's minimum API level is set to 21 or higher and that Jetpack Compose is enabled. The following code snippet also sets the version for the Kotlin compiler plugin:

```
  android {
    defaultConfig {
      ...
      minSdkVersion 21
    }
    buildFeatures {
      compose true
    }
    ...
    compileOptions {
      sourceCompatibility JavaVersion.VERSION_11
      targetCompatibility JavaVersion.VERSION_11
    }
    kotlinOptions {
      jvmTarget = "11"
    }
    composeOptions {
      kotlinCompilerExtensionVersion compose_version
    }
  }
```

Finally, declare dependencies. The following code snippet acts as a good starting point. Depending on which packages your app uses, you may need additional ones:

```
  dependencies {
      implementation 'androidx.core:core-ktx:1.7.0'
      implementation
  'androidx.appcompat:appcompat:1.4.0'
      Implementation
        "androidx.compose.ui:ui:$compose_version"
```

```
    implementation
      "androidx.compose.material:material:$compose_ve
rsion"
    implementation
      "androidx.compose.ui:ui-tooling-
       preview:$compose_version"
    implementation
      'androidx.lifecycle:lifecycle-runtime-
ktx:2.4.0'
    implementation
      'androidx.activity:activity-compose:1.4.0'
    debugImplementation
      "androidx.compose.ui:ui-
tooling:$compose_version"
}
```

Once you have configured your project, building and running a Compose app works just like traditional view-based apps.

Pressing the play button

To run your Compose app, select your target device, make sure that the **app** module is selected, and press the green *play* button (*Figure 1.8*):

Figure 1.8 – Android Studio toolbar elements to launch an app

Congratulations! Well done. You have now launched your first Compose app, and you have achieved quite a lot. Let's recap.

Summary

In this chapter, we learned how to write our first composables: Kotlin functions that have been annotated with **`@Composable`**. Composable functions are the core building blocks of Jetpack Compose-based UIs. You com-

bined existing library composables with your own to create beautiful app screens. To see a preview, we can add the **`@Preview`** annotation. To use Jetpack Compose in a project, both **`build.gradle`** files must be configured accordingly.

In *Chapter 2*, *Understanding the Declarative Paradigm*, we will take a closer look at the differences between the declarative approach of Jetpack Compose and the imperative nature of traditional UI frameworks such as Android's view-based component library.

Further reading

This book assumes you have a basic understanding of the syntax of Kotlin and Android development in general. If you would like to learn more about this, I suggest looking at *Android Programming with Kotlin for Beginners*, *John Horton*, *Packt Publishing*, *2019*, *ISBN 9781789615401*.