Chapter 3: Exploring the Key Principles of Compose

In the first chapter of this book, we built and run our first Jetpack Compose app. Then, in *Chapter 2*, *Understanding the Declarative Paradigm,* we explained the imperative nature of Android's traditional UI toolkit, illustrated some of its weaknesses, and saw how a declarative approach can overcome them.

In this chapter, we build upon these foundations by examining a few key principles Jetpack Compose relies on. This knowledge is essential for writing well-behaving Compose apps. This chapter introduces these key principles.

In this chapter, we will cover the following topics:

- Looking closer at composable functions
- Composing and recomposing the **user interface (UI)**
- Modifying the behavior of composable functions

We will start by revisiting composable functions, the building blocks of a composable UI. This time, we will dig much deeper into their underlying ideas and concepts. By the end of the first main section, you will have established a thorough understanding of what composable functions are, how they are written, and how they are used.

The following section focuses on creating and updating the UI. You will learn how Jetpack Compose achieves what other UI frameworks call repainting. This mechanism, which is called **recomposition** in Compose, takes place automatically whenever something relevant to the UI changes. To keep this process fluent, your composable functions must adhere to a few best practices. I will explain them to you in this section.

We will close this chapter by expanding our knowledge of the concept of modifiers. We will take a close look at how modifier chains work and what you need to keep in mind to always get the intended results. You will also learn how to implement custom modifiers. They allow you to amend any composable function to look or behave in precisely the way you want them to.

Now, let's get started!

Technical requirements

Please refer to the *Technical requirements* section of *Chapter 1*, *Building Your First Compose App*, for information on how to install and set up Android Studio, as well as how to get the sample apps. If you want to try the **`ShortColoredTextDemo()`** and **`ColoredTextDemo()`** composables from the *Looking closer at composable functions* section, you can use the **`Sandbox`** app project in the top-level directory of this book's GitHub repository at https://github.com/PacktPublishing/Android-UI-Development-with-Jetpack-Compose. Open **`SandboxActivity`** and copy the composable functions from **`code_snippets.txt`**, which will be located in the **`/chapter_03`** folder.

Looking closer at composable functions

The UI of a Compose app is built by writing and calling composable functions. We have already done both in the previous chapters, but my explanations regarding the structure of a composable, as well as its internals, have been quite basic – it's time to fix that.

Building blocks of composable functions

A **composable function** is a Kotlin function that has been annotated with `@Composable`.All composables *must* be marked this way because the annotation informs the Compose compiler that the function converts data into UI elements.

The signature of a Kotlin function consists of the following parts or building blocks:

- An optional visibility modifier (`private`, `protected`, `internal`, or `public`)
- The `fun` keyword
- A name
- A list of parameters (can be empty) or, optionally, a default value
- An optional return type
- A block of code

Let's explore these parts in greater detail.

The default visibility (if you omit the modifier) is `public`. This means that the (composable) function can be called from anywhere. If a function is meant to be reused (for example, a text styled to match your brand), it should be publicly available. On the other hand, if a function is tied to a particular **context** (the region of code, such as a class), it may make sense to restrict its access. There is an open debate on how rigid the visibility of functions should be restrained. In the end, you and your team need to agree on a point of view and stick to it. For the sake of simplicity, my examples are usually public.

The name of a composable function uses the *PascalCase* notation: it starts with an uppercase letter, whereas the remaining characters are lowercase. If the name consists of more than one word, each word follows this rule. The name should be a noun (`Demo`), or a noun that has been prefixed with a descriptive adjective (`FancyDemo`). Unlike other (ordinary) Kotlin functions, it should *not* be a verb or a verb phrase (`getDataFromServer`). The *API Guidelines for Jetpack Compose* file, which is available at https://github.com/androidx/androidx/blob/androidx-

main/compose/docs/compose-api-guidelines.md, details these naming conventions.

All the data you want to pass to a composable function is provided through a comma-separated list, which is enclosed in parenthesis. If a composable does not require values, the list remains empty. Here's a composable function that can receive two parameters:

```
@Composable
fun ColoredTextDemo(
  text: String = "",
  color: Color = Color.Black
) {
  Text(
    text = text,
    style = TextStyle(color = color)
  )
}
```

In Kotlin, function parameters are defined as **`name: type`**. Parameters are separated by a comma. You can specify a default value by adding **`= ...`**. This is used if no value is provided for a particular parameter when the function is being invocated.

The return type of a function is optional. In this case, the function returns **`Unit`**. **`Unit`** is a type with only one value: **`Unit`**. If, like in this example, it is omitted, the function body follows immediately after the list of arguments. Most composable functions you will be writing do not need to return anything, so do not need a return type. Situations that require it will be covered in the *Returning values* section.

If the code of a function contains more than one statement or expression, it will be enclosed in curly braces. Kotlin offers a nice abbreviation for if just one expression needs to be executed – Jetpack Compose itself uses this quite frequently.

```
@Composable
fun ShortColoredTextDemo(
    text: String = "",
    color: Color = Color.Black
) = Text(
    text = text,
    style = TextStyle(color = color)
)
```

As you can see, the expression follows an equals sign. This means that **`ShortColoredTextDemo()`** returns whatever **`Text()`** is returning.

Unlike Java, Kotlin does not know about the **`void`** keyword, so all the functions must return *something*. By omitting the return type, we implicitly tell Kotlin that the return type of a function is **`kotlin.Unit`**. This type has only one value: the **`Unit`** object. So, **`Unit`** corresponds to **`void`** in Java.

Let's test this by printing the result of invoking a composable function:

```
class SandboxActivity : ComponentActivity() {
  override fun onCreate(savedInstanceState: Bundle?)
{
    super.onCreate(savedInstanceState)
    setContent {
      println(ColoredTextDemo(
        text = "Hello Compose",
        color = Color.Blue
      ))
    }
  }
}
```

If you run the app, the following line will be printed:

```
I/System.out: kotlin.Unit
```

While this may not look too exciting, its implications are profound. Think of it: although the **`ColoredTextDemo()`** composable function returns noth-

ing interesting, some text is shown on the screen. This happens because it invokes another composable, called **`Text()`**. So, whatever may be needed to show text must happen inside **`Text()`**, and it cannot have anything to do with the return value of a composable.

In the previous chapter, I said that composable functions *emit* UI elements. I will explain what this means in the next section.

Emitting UI elements

A Compose UI is created by nesting calls to composable functions, which can be provided by the Jetpack Compose libraries, code of other developers, or your app.

Let's find out what happens once **`ColoredTextDemo()`** has called **`androidx.compose.material.Text()`**. To see the source code of (among others) composable functions in Android Studio, you can click on their names while holding down the *Ctrl* key (on a Mac, it's the *cmd* key).

PLEASE NOTE

I will only show you the important steps because otherwise, I would need to copy too much code. To get the best learning experience, please follow the call chain directly in your IDE.

`Text()` defines two variables, **`textColor`** and **`mergedStyle`**, and passes them to **`androidx.compose.foundation.text.BasicText()`**. Although you can use **`BasicText()`** in your code, you should choose **`androidx.compose.material.Text()`** if possible, because it consumes style information from a theme. Please refer to *Chapter 6, Putting Pieces Together*, for more information about themes.

`BasicText()` immediately delegates to **`CoreText()`**, which belongs to the **`androidx.compose.foundation.text`** package too. It is an internal composable function, meaning you can't use it in your apps.

CoreText() initializes and remembers quite a few variables. There is no need to explain them all here, but the most important piece is the invocation of another composable function: **Layout()**.

Layout() belongs to the **androidx.compose.ui.layout** package. It is the core composable function for the layout, with its purpose being to size and position children. *Chapter 4*, *Laying Out UI Elements*, covers this in great detail. Right now, we still need to find out what *emitting UI elements* means. So, let's see what **Layout()** does:

```
66  @Suppress( ...names: "ComposableLambdaParameterPosition")
67  @Composable inline fun Layout(
68      content: @Composable () -> Unit,
69      modifier: Modifier = Modifier,
70      measurePolicy: MeasurePolicy
71  ) {
72      val density = LocalDensity.current
73      val layoutDirection = LocalLayoutDirection.current
74      ReusableComposeNode<ComposeUiNode, Applier<Any>>(
75          factory = ComposeUiNode.Constructor,
76          update = { this: Updater<ComposeUiNode>
77              set(measurePolicy, ComposeUiNode.SetMeasurePolicy)
78              set(density, ComposeUiNode.SetDensity)
79              set(layoutDirection, ComposeUiNode.SetLayoutDirection)
80          },
81          skippableUpdate = materializerOf(modifier),
82          content = content
83      )
84  }
```

Figure 3.1 – Source code of Layout()

Layout() invokes **ReusableComposeNode()**, which belongs to the **androidx.compose.runtime** package. This composable function *emits* a so-called **node**, a UI element hierarchy. Nodes are created using a factory, which is passed through the **factory** argument. The **update** and **skippableUpdate** parameters receive code that performs updates on the node, with the latter one handling modifiers (we will be taking a closer look at them at the end of this chapter). Finally, **content** contains composable functions that become the children of the node.

PLEASE NOTE

When we speak of composable functions emitting UI elements, we mean that **nodes** *are added to data structures that are internal to Jetpack Compose. This will eventually lead to UI elements being visible.*

To complete the call chain, let's briefly look at **`ReusableComposeNode()`**:

```
411  @Composable @ExplicitGroupsComposable
412  inline fun <T, reified E : Applier<*>> ReusableComposeNode(
413      noinline factory: () → T,
414      update: @DisallowComposableCalls Updater<T>.() → Unit,
415      noinline skippableUpdate: @Composable SkippableUpdater<T>.() → Unit,
416      content: @Composable () → Unit
417  ) {
418      if (currentComposer.applier !is E) invalidApplier()
419      currentComposer.startReusableNode()
420      if (currentComposer.inserting) {
421          currentComposer.createNode(factory)
422      } else {
423          currentComposer.useNode()
424      }
425      currentComposer.disableReusing()
426      Updater<T>(currentComposer).update()
427      currentComposer.enableReusing()
428      SkippableUpdater<T>(currentComposer).skippableUpdate()
429      currentComposer.startReplaceableGroup( key: 0x7ab4aae9)
430      content()
431      currentComposer.endReplaceableGroup()
432      currentComposer.endNode()
433  }
```

Figure 3.2 – Source code of ReusableComposeNode()

`currentComposer` is a top-level variable inside **`androidx.compose.runtime.Composables.kt`**. Its type is **`Composer`**, which is an interface. **`Composer`** is targeted by the Jetpack Compose Kotlin compiler plugin and used by code generation helpers; your code should not call it directly. **`ReusableComposeNode`** determines if a new node should be created or whether an existing one should be reused. It then performs updates and finally emits the content to the node by invoking **`content()`**.

Based on what you know by now, let me elaborate a little more on nodes. **`Layout()`** passes **`ComposeUiNode.Constructor`** to **`ReusableComposeNode`** as the **`factory`** argument, which is used to create a node

(`currentComposer.createNode(factory)`). So, the features of a node are defined by the `ComposeUiNode` interface:

```
    Interface extracted from LayoutNode to not mark the whole LayoutNode class as @PublishedApi.
27  @PublishedApi
28  internal interface ComposeUiNode {
29      var measurePolicy: MeasurePolicy
30      var layoutDirection: LayoutDirection
31      var density: Density
32      var modifier: Modifier
33
        Object of pre-allocated lambdas used to make use with ComposeNode allocation-less.
37      companion object {
38          val Constructor: () -> ComposeUiNode = LayoutNode.Constructor
39          val SetModifier: ComposeUiNode.(Modifier) -> Unit = { this.modifier = it }
40          val SetDensity: ComposeUiNode.(Density) -> Unit = { this.density = it }
41          val SetMeasurePolicy: ComposeUiNode.(MeasurePolicy) -> Unit =
42              { this.measurePolicy = it }
43          val SetLayoutDirection: ComposeUiNode.(LayoutDirection) -> Unit =
44              { this.layoutDirection = it }
45      }
46  }
```

Figure 3.3 – Source code of ComposeUiNode

A node has four properties, as defined by the following classes or interfaces:

- `MeasurePolicy`
- `LayoutDirection`
- `Density`
- `Modifier`

In essence, a node is an element in a Compose hierarchy. You will not be dealing with them in your code because nodes are part of the inner workings of Jetpack Compose that are not exposed to apps. However, you will see `MeasurePolicy`, `LayoutDirection`, `Density`, and `Modifier` throughout this book. They represent important data structures and concepts that are relevant to apps.

This concludes our investigation of how UI elements are emitted (nodes are added to data structures that are internal to Jetpack Compose). In the next section, we will look at composable functions that return values.

Returning values

Most of your composable functions will not need to return something, so they will not specify a return type. This is because the main purpose of a composable is to compose the UI. As you saw in the previous section, this is done by emitting UI elements or element hierarchies. But when do we need to return something different than **`Unit`**?

Some of my examples invoke **`remember {}`** to retain state for future use and **`stringResource()`** to access strings that are stored in the **`strings.xml`** file. To be able to perform their tasks, both must be composable functions.

Let's look at **`stringResource()`** to see why. Remember that you can press Ctrl + click on a name to see its source code. The function is pretty short; it does just two things:

```
  val resources = resources()
  return resources.getString(id)
```

`resources()` is a composable too. It returns **`LocalContext.current.resources`**. **`LocalContext`** is a top-level variable in **`AndroidCompositionLocals.android.kt`** that belongs to the **`androidx.compose.ui.platform`** package. It returns an instance of **`StaticProvidableCompositionLocal`**, which holds **`android.content.Context`**. This object provides access to resources.

Even though the returned data has nothing to do with Jetpack Compose, the code that obtains it must conform to Jetpack Compose mechanics because, in the end, it will be called from a composable function. The important thing to remember is that if you need to return something that is part of the composition and recomposition mechanic, you must make your function composable by annotating it with **`@Composable`**. Also, such functions do not follow the naming conventions for composable functions but follow a *camelCase* style (they begin with a small letter, with the subse-

quent word starting in uppercase) and consist of verb phrases (`rememberScrollState`).

In the next section, we will return to composing UIs at the app level. You will learn more about the terms **composition** and **recomposition**.

Composing and recomposing the UI

Unlike imperative UI frameworks, Jetpack Compose does not depend on the developer proactively modifying a component tree when changes in the app data require changes to be made to the UI. Instead, Jetpack Compose detects such changes on its own and updates only the affected parts.

As you know by now, a Compose UI is declared *based on* the current app data. In my previous examples, you have seen quite a few conditional expressions (such as `if` or `when`) that determine which composable function is called or which parameters it receives. So, we are describing the *complete* UI in our code. The branch that will be executed depends on the app data (state) during runtime. The Web framework that React has a similar concept called Virtual DOM. But doesn't this contradict with me saying *Compose detects such changes on its own and updates only the affected parts*?

Conceptually, Jetpack Compose regenerates the entire UI when changes need to be applied. This, of course, would waste time, battery, and processing power. And it might be noticeable by the user as screen flickering. Therefore, the framework puts a lot of effort into making sure only those parts of the UI element tree requiring an update are regenerated.

You saw some of these efforts in the previous section, where I briefly mentioned `update` and `skippableUpdate`. To ensure fast and reliable **recompositions** (the Jetpack Compose term for updating, regenerating, or

repainting), you need to make sure your composable functions follow a few simple rules. I will introduce them to you by walking you through the code of an app called **`ColorPickerDemo`**:

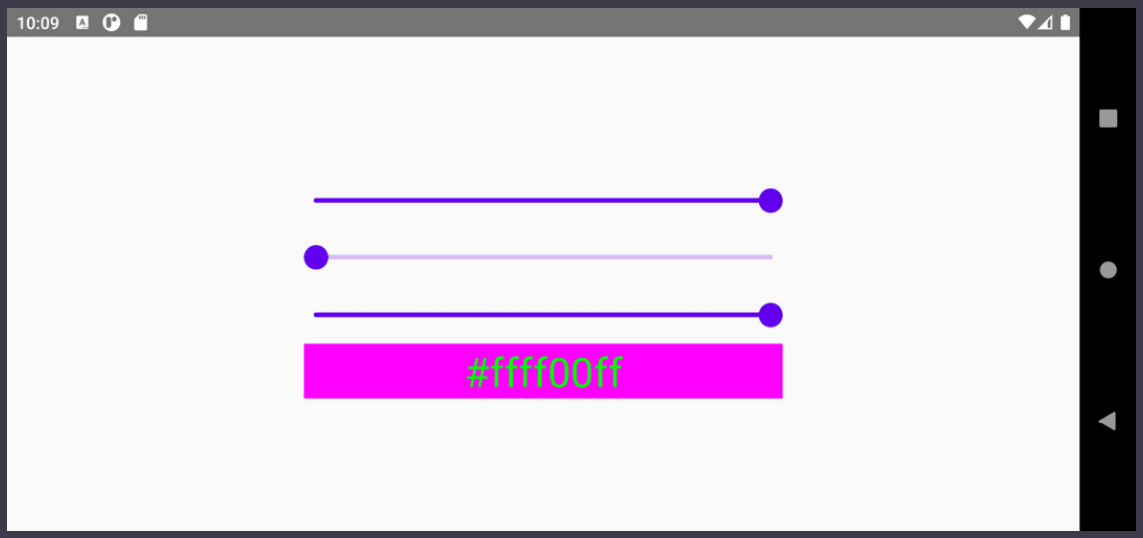

Figure 3.4 – The ColorPickerDemo app

The app aims to set a color by specifying its **red, green, and blue** (**RGB**) portions. This color is used as the background color of a text (which displays the value of the color as a hexadecimal string). The foreground color is complementary to the selected one.

In the next few sections, we look at its code. You will learn how sliders communicate changes in their values.

Sharing state among composable functions

Sometimes, you may want to use a state in more than one composable function. For example, you may wish to use the color portion that's been set by one slider to create the full color, which, in turn, becomes the background color of a text. So, how can you share state? Let's start by looking at **`ColorPicker()`** – it groups three sliders vertically in a **`Column()`**:

```
@Composable
fun ColorPicker(color: MutableState<Color>) {
  val red = color.value.red
  val green = color.value.green
```

```
    val blue = color.value.blue
    Column {
      Slider(
        value = red,
        onValueChange = { color.value = Color(it,
green,
                                              blue)
})
      Slider(
        value = green,
        onValueChange = { color.value = Color(red, it,
blue) })
      Slider(
        value = blue,
        onValueChange = { color.value = Color(red,
green, it) })
    }
}
```

The composable receives one parameter: **`MutableState<Color>`**. The **`value`** property of **`color`** contains an instance of **`androidx.compose.ui.graphics.Color`**. Its **`red`**, **`green`**, and **`blue`** properties return a **`Float`** based on the so-called **color space**, which is used to identify a specific organization of colors. Each color space is characterized by a color model, which, in turn, defines how a color value is represented. If not specified otherwise, this is **`ColorSpaces.Srgb`**.

My code does not set a particular color space, so it defaults to **`ColorSpaces.Srgb`**. This causes the value being returned to be between **`0F`** and **`1F`**. The first three lines assign the red, green, and blue portions of the color to local variables named **`red`**, **`green`**, and **`blue`**. They are used for the **`Slider()`** functions; let's see how.

Each slider in my example receives two parameters: **`value`** and **`onValueChange`**. The first specifies the value that the slider will display. It must be between **`0F`** and **`1F`** (which fits nicely with **`red`**, **`green`**, and **`blue`**). If

needed, you can supply an alternative range through the optional **`valueRange`** parameter. **`onValueChange`** is invoked when the user drags the slider handle or clicks on the thin line underneath. The code of the three lambda expressions is quite similar: a new **`Color`** object is created and assigned to **`color.value`**. Color portions that are being controlled by other sliders are taken from the corresponding local variables. They have not been changed. The new color portion of the current slider can be obtained from **`it`** because it is the new slider value, which is passed to **`onValueChange`**.

By now, you may be wondering why **`ColorPicker()`** receives the color wrapped inside a **`MutableState<Color>`**. Wouldn't it suffice to pass it directly, using **`color: Color`**? As shown in *Figure 3.4*, the app shows the selected color as a text with complementary background and foreground colors. But **`ColorPicker()`** does not emit text. This happens somewhere else (as you will see shortly, inside a **`Column()`**). To show the correct color, the text must receive it too. As the color change takes place inside **`ColorPicker()`**, we must inform the caller about it. An ordinary **`Color`** instance being passed as a parameter can't do that because Kotlin function parameters are immutable.

We can achieve changeability using global properties. But this is not recommended for Jetpack Compose. Composables should not use global variables at all. It is a best practice to pass all the data that influences the look or behavior of a composable function as parameters. If that data is modified inside the composable, you should use **`MutableState`**. Moving state to a composable's caller by receiving a state is called **state hoisting**. A good alternative to passing **`MutableState`** and applying changes inside a composable is to pass the change logic as a lambda expression. In my example, **`onValueChange`** would just provide the new slider value to the lambda expression.

IMPORTANT

Try to make your composables side effect-free. Having no side effects means calling a function repeatedly with the same set of arguments that will always produce the same result. Besides getting all the relevant data from the caller, being free of side effects also requires not relying on global properties or calling functions that return unpredictable values. There are a few scenarios where you want side effects. I will cover these in Chapter 7, Tips, Tricks, and Best Practices.

Now, let's learn how the color is passed to the text:

```
Column(
  modifier = Modifier.width(min(400.dp, maxWidth)),
  horizontalAlignment = Alignment.CenterHorizontally
) {
  val color = remember {
mutableStateOf(Color.Magenta) }
  ColorPicker(color)
  Text(
    modifier = Modifier
      .fillMaxWidth()
      .background(color.value),
    text =
    "#${color.value.toArgb().toUInt().toString(16)}",
    textAlign = TextAlign.Center,
    style = MaterialTheme.typography.h4.merge(
      TextStyle(
        color = color.value.complementary()
      )
    )
  )
}
```

`ColorPicker()` and **`Text()`** are laid out vertically (being centered horizontally) inside a **`Column()`**. The width of the column is either **`400`** density-independent pixels or **`maxWidth`**, depending on which value is smaller. **`maxWidth`** is defined by the predefined **`BoxWithConstraints()`** composable

(you will learn more about it in the *Controlling size* section). The color for both `ColorPicker()` and `Text()` is defined like this:

```
val color = remember { mutableStateOf(Color.Magenta)
}
```

When `Column()` is composed for the first time, `mutableStateOf(Color.Magenta)` is executed. This creates **state**. State represents app data (in this case, a color) that changes over time. You will learn more about state in *Chapter 5*, *Managing the State of Your Composable Functions*. For now, it suffices to understand that the state is *remembered* and assigned to `color`.

But what does `remember` mean? Any subsequent composition, which is called **recomposition**, will lead to `color` receiving the value created by `mutableStateOf` – that is, a reference to a `MutableState<Color>` (state hoisting). The lambda expression that's passed to `remember` is called a **calculation**. It will only be evaluated once. Recompositions always return the same value.

If the reference remains the same, how can the color be changed? The actual color is accessed through the `value` property. You saw this in the code of `ColorPicker()`. `Text()` does not modify the color – it only works with it. Therefore, we pass `color.value` (which is the color), not the mutable state (`color`), to some of its parameters, such as `background`. Note that this is a modifier. You will learn more about them in the *Modifying behavior* section. It sets the background color of a UI element that's emitted by a composable function.

Also, have you noticed the call of `complementary()` inside `TextStyle()`? Here's what it does:

```
fun Color.complementary() = Color(
  red = 1F - red,
  green = 1F - green,
  blue = 1F - blue
```

```
)
```

`complementary()` is an extension function of `Color`. It computes the complementary color to the one it receives. This is done to make the text (the hexadecimal RGB value of the color that was selected using the three sliders) readable, regardless of the currently selected color (which is used as the background of the text).

In this section, I talked about some very important Jetpack Compose concepts. Let's recap what we've learned so far:

- A compose UI is defined by nesting calls to composable functions
- Composable functions emit UI elements or UI element hierarchies
- Building the UI for the first time is called **composition**
- Rebuilding the UI upon changes being made to app data is called **recomposition**
- Recomposition happens automatically
  IMPORTANT
  There is no way for your app to predict when or how often recomposition will take place. If animations are involved, this may happen each frame. Therefore, it is of utmost importance to make your composables as fast as possible. You may never do time-consuming calculations, load or save data, or access the network. Any such code must be executed outside of composable functions. They only receive ready data. Also, please note that the order of recomposition is unspecified. This means that the first child of, say, a `Column()`*, might be recomposed later than a sibling that appears after it in the source code. Recomposition can occur in parallel and it may be skipped. Therefore, never rely on a particular order of recomposition, and never compute something in a composable that is needed somewhere else.*

In the next section, we will finish our walkthrough of the `ColorPickerDemo` app. I will show you how to specify and limit the dimensions of composable functions.

Controlling size

Most of my examples contain code such as **`fillMaxSize()`** or **`fillMaxWidth()`**. Both modifiers control the size of a composable. **`fillMaxSize()`** uses all the available horizontal and vertical space, while **`fillMaxWidth()`** maximizes only the horizontal expansion.

However, **`fillMaxWidth()`** may not be the right choice for sliders. In my opinion, large sliders are awkward to use due to the distance you would need to drag their handles to reach the minimum or maximum value. So, the question is, how can we limit its width? The most straightforward solution is to use the **`width()`** modifier. It sets the preferred width of a composable to a particular size. I want sliders to be 400 density-independent pixels wide at most. If the screen is smaller, its width should be used instead. Here's how you achieve this:

```
  modifier = Modifier.width(min(400.dp, maxWidth)),
```

The modifier belongs to the **`Column()`** property that contains both **`ColorPicker()`** and **`Text()`**.

`maxWidth` is provided by the **`BoxWithConstraints()`** composable:

```
  BoxWithConstraints(
    contentAlignment = Alignment.Center,
    modifier = Modifier.fillMaxSize()
  ) {
    Column ...
  }
```

Its content receives an instance of a **`BoxWithConstraintsScope`** scope, which provides access to **`constraints`**, **`minWidth`**, **`minHeight`**, **`maxWidth`**, and **`maxHeight`**. **`BoxWithConstraints()`** defines its content according to the available space, based on incoming constraints. You will learn more about this in *Chapter 4*, *Laying Out UI Elements*.

This concludes our walkthrough of the **`ColorPickerDemo`** app. In the next section, we take a closer look at how a composable hierarchy is displayed in an **`Activity`**.

Displaying a composable hierarchy inside an Activity

In the previous section, we built a UI element hierarchy consisting of three sliders and some text. We embedded it in an **`Activity`** using **`setContent`**, an extension function of **`androidx.activity.ComponentActivity`**. This implies that you cannot invoke **`setContent`** on *any* activity, but only ones that extend **`ComponentActivty`**. This is the case for **`androidx.appcompat.app.AppCompatActivity`**.

However, this class inherits quite a lot of functionality that is relevant for the old View-based world, such as support for toolbars and the options menu. Jetpack Compose handles these differently. You will learn more about this in *Chapter 6*, *Putting Pieces Together*. Therefore, you should avoid using **`AppCompatActivity`**, and instead extend **`ComponentActivity`** if possible. For combining View-based and Compose UIs, please refer to *Chapter 9*, *Exploring Interoperability APIs*.

Let's return to **`setContent`**. It expects two parameters:

- **`parent`**, an optional **`CompositionContext`**
- **`content`**, a composable function for declaring the UI

You will likely omit **`parent`** most of the time. **`CompositionContext`** is an abstract class that belongs to the **`androidx.compose.runtime`** package. It is used to logically connect two compositions. This refers to the inner workings of Jetpack Compose that you do not need to worry about in your app code. Yet, to get an idea of what this means, let's look at the source code of **`setContent`**:

```
public fun ComponentActivity.setContent(
    parent: CompositionContext? = null,
    content: @Composable () → Unit
) {
    val existingComposeView = window.decorView
        .findViewById<ViewGroup>(android.R.id.content)
        .getChildAt( index: 0) as? ComposeView

    if (existingComposeView ≠ null) with(existingComposeView) {  this: ComposeView
        setParentCompositionContext(parent)
        setContent(content)
    } else ComposeView( context: this).apply {  this: ComposeView
        // Set content and parent **before** setContentView
        // to have ComposeView create the composition on attach
        setParentCompositionContext(parent)
        setContent(content)
        // Set the view tree owners before setting the content view so that the inflation process
        // and attach listeners will see them already present
        setOwners()
        setContentView( view: this, DefaultActivityContentLayoutParams)
    }
}
```

Figure 3.5 – The source code of setContent

First, **findViewById()** is used to find out if the activity already contains content that is an instance of **androidx.compose.ui.platform.ComposeView**. If so, the **setParentCompositionContext()** and **setContent()** methods of this view will be invoked.

Let's look at **setParentCompositionContext()** first. It belongs to **AbstractComposeView**, the immediate parent of **ComposeView**. It sets a **CompositionContext** that should be the parent of the view's composition. If that context is **null**, it will be determined automatically: **AbstractComposeView** contains a private function called **ensureCompositionCreated()**. It invokes another implementation of **setContent** (an internal extension function of **ViewGroup** that's defined in **Wrapper.android.kt**) and passes the result of a call to **resolveParentCompositionContext()** as a **parent**.

Now, let's return to the version of **setContent()** that's shown in the preceding screenshot. Once **setParentCompositionContext()** has been called, it invokes yet another version of **setContent()**. This implementation belongs to **ComposeView**. It sets the content of the view.

If **findViewById()** does not return a **ComposeView**, a new instance is created and passed to **setContentView**, after **setParentCompositionContext()** and

`setContent()` have been invoked.

In this section, we continued looking at some of the inner workings of Jetpack Compose. You now know that `ComposeView` is the missing link to the old-fashioned View-based world. We will revisit this class in *Chapter 9, Exploring Interoperability APIs*.

In the next section, we will return modifiers; you will learn how they work under the hood and how you can write your own.

Modifying the behavior of composable functions

Unlike components in traditional imperative UI frameworks, composable functions do not share a basic set of properties. They also do not automatically (in the sense of inheriting) reuse functionality. This must be done explicitly by calling other composables. Their visual appearance and behavior can be controlled through parameters, modifiers, or both. In a way, modifiers pick up the idea of properties in a component but enhance it – unlike properties of components, modifiers can be used completely at the discretion of the developer.

You have already seen quite a few modifiers in my examples, such as the following:

- `width()`
- `fillMaxWidth()`
- `fillMaxSize()`

These control the width and size of the corresponding UI element; `background()` can set a background color and shape, while `clickable {}` allows the user to interact with the composable function by clicking on the UI element. Jetpack Compose provides an extensive list of modifiers, so it may take some time to make yourself familiar with most of them.

Conceptually, these modifiers can be assigned to one of several categories, such as *Actions* (`draggable()`), *Alignment* (`alignByBaseline()`), or *Drawing* (`paint()`). You can find a list of modifiers grouped by category at https://developer.android.com/jetpack/compose/modifiers-list.

To further familiarize yourself with modifiers, let's look at the `ModifierDemo` example. It contains several composable functions. The following screenshot shows the app running `OrderDemo()`:

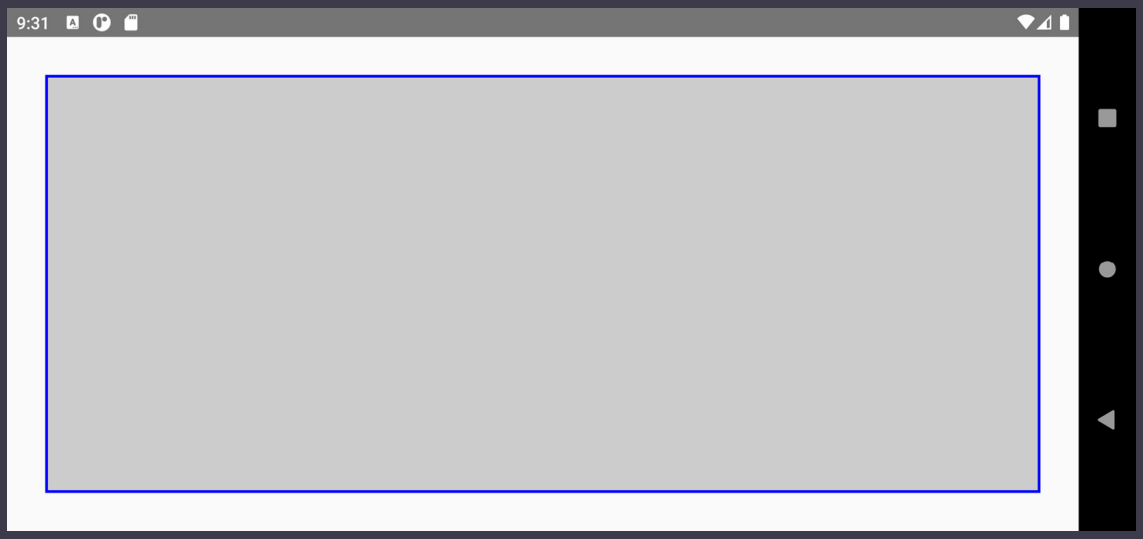

Figure 3.6 – The ModifierDemo app

The composable produces a gap of 32 density-independent pixels on all its sides, followed by a 2 density-independent pixels wide blue border. The inner rectangle is painted in light gray.

Here's what the code looks like:

```
@Composable
fun OrderDemo() {
  var color by remember { mutableStateOf(Color.Blue)
}
  Box(
    modifier = Modifier
      .fillMaxSize()
      .padding(32.dp)
```

```
        .border(BorderStroke(width = 2.dp, color =
  color))
        .background(Color.LightGray)
        .clickable {
          color = if (color == Color.Blue)
            Color.Red
          else
            Color.Blue
        }
    )
  }
```

`Box()` is clickable – doing so changes the border color from blue to red and back. If you click inside the gaps, nothing will happen. If, however, you move **`.clickable { }`** before **`.padding(32.dp)`**, clicks work inside the gaps too. This is intentional. Here's what happens: you define a modifier chain by combining several modifiers with `.`. In doing so, you specify the order in which the modifiers are used. The location of a modifier in the chain determines when it is executed. As **`clickable {}`** only reacts to clicks inside the bounds of a composable, the padding is not considered for clicks when it occurs before **`clickable {}`**.

In the next section, I will show you how Jetpack Compose handles modifiers and modifier chains internally.

Understanding how modifiers work

Composable functions that accept modifiers should receive them via the **`modifier`** parameter and assign it a default value of **`Modifier`**. **`modifier`** should be the first optional parameter and thus appear after all the required ones, except for trailing lambda parameters.

Let's see how a composable can receive a **`modifier`** parameter:

```
@Composable
fun TextWithYellowBackground(
```

```
    text: String,
    modifier: Modifier = Modifier
  ) {
    Text(
      text = text,
      modifier = modifier.background(Color.Yellow)
    )
  }
```

This way, the composable can receive a modifier chain from the caller. If none are provided, **`Modifier`** acts as a new, empty chain. In both cases, the composable can add additional modifiers, such as **`background()`** in the previous code snippet.

If a composable function accepts a modifier that will be applied to a specific part or child of its corresponding UI element, the name of this part or child should be used as a prefix, such as **`titleModifier`**. Such modifiers follow the rules I mentioned previously. They should be grouped and appear after the parent's modifier. Please refer to https://developer.android.com/reference/kotlin/androidx/compose/ui/Modifier for additional information regarding the definition of modifier parameters.

Now that you know how to define a **`modifier`** parameter in a composable function, let's focus a little more on the idea of chaining. **`Modifier`** is both an interface and a companion object. The interface belongs to the **`androidx.compose.ui`** package. It defines several functions, such as **`foldIn()`** and **`foldOut()`**. You won't need them, though. The important one is **`then()`**. It concatenates two modifiers. As you will see shortly, you need to invoke it in your modifiers. The **`Element`** interface extends **`Modifier`**. It defines a single element contained within a **`Modifier`** chain. Finally, the **`Modifier`** companion object is the empty, default modifier, which contains no elements.

TO SUMMARIZE

A modifier is an ordered, immutable collection of modifier elements.

Next, let's see how the **`background()`** modifier is implemented:

```
Draws shape with a solid color behind the content.
Params: color - color to paint background with
        shape - desired shape of the background
Samples: androidx.compose.foundation.samples.DrawBackgroundColor
        // Unresolved
fun Modifier.background(
    color: Color,
    shape: Shape = RectangleShape
) = this.then(
    Background(
        color = color,
        shape = shape,
        inspectorInfo = debugInspectorInfo { this: InspectorInfo
            name = "background"
            value = color
            properties["color"] = color
            properties["shape"] = shape
        }
    )
)
```

Figure 3.7 – Source code of the background() modifier

`background()` is an extension function of **`Modifier`**. It receives a **`Modifier`** instance. It invokes **`then()`** and returns the result (a concatenated modifier). **`then()`** expects just one parameter: the *other* modifier that should be concatenated with the current one. In the case of **`background()`**, *other* is an instance of **`Background`**. This class extends **`InspectorValueInfo`** and implements the **`DrawModifier`** interface, which, in turn, extends **`Modifier.Element`**. As **`InspectorValueInfo`** is primarily used for debugging purposes, I will not elaborate on it any further. **`DrawModifier`**, on the other hand, is very interesting. Implementations can draw into the space of a UI element. We will make use of this in the final section.

Implementing custom modifiers

Although Jetpack Compose contains an extensive list of modifiers, you may want to implement your own. Let me show you how to do this. My example, **`drawYellowCross()`**, draws two thick yellow lines behind the content, which is some **`Text()`** here:

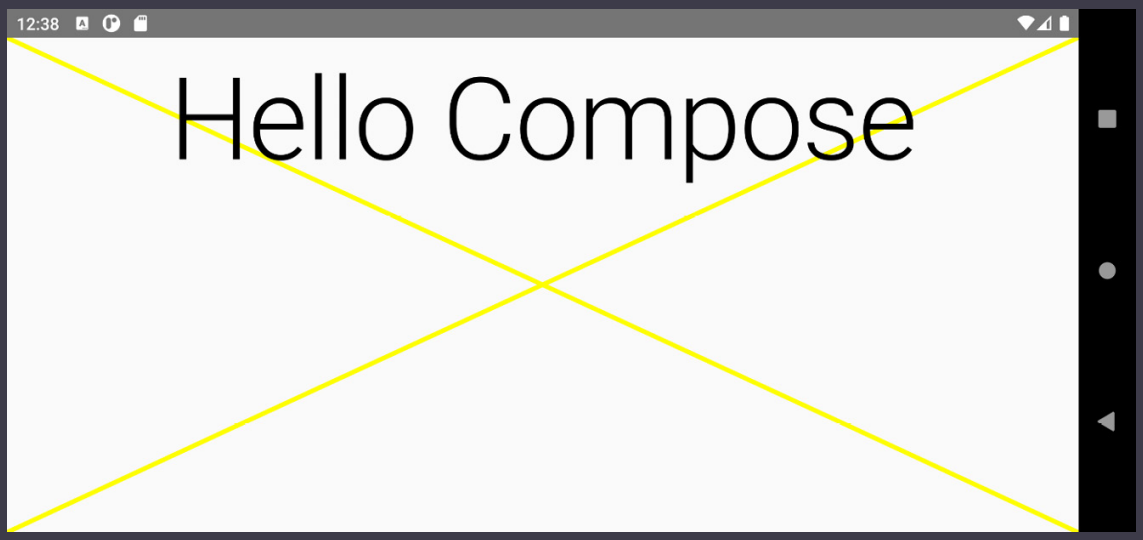

Figure 3.8 – A custom modifier

The modifier is invoked like this:

```
Text(
  text = "Hello Compose",
  modifier = Modifier
    .fillMaxSize()
    .drawYellowCross(),
  textAlign = TextAlign.Center,
  style = MaterialTheme.typography.h1
)
```

As you can see, the modifier integrates nicely into an existing modifier chain. Now, let's look at the source code:

```
fun Modifier.drawYellowCross() = then(
  object : DrawModifier {
    override fun ContentDrawScope.draw() {
      drawLine(
        color = Color.Yellow,
```

```
            start = Offset(0F, 0F),
            end = Offset(size.width - 1, size.height -
  1),
            strokeWidth = 10F
          )
          drawLine(
            color = Color.Yellow,
            start = Offset(0F, size.height - 1),
            end = Offset(size.width - 1, 0F),
            strokeWidth = 10F
          )
          drawContent()
        }
      }
  )
```

`drawYellowCross()` is an extension function of **`Modifier`**. This means we can invoke **`then()`** and simply return the result. **`then()`** receives an instance of **`DrawModifier`**. After that, we need to implement only one function, called **`draw()`**, which is an extension function of **`ContentDrawScope`**. This interface defines one function (**`drawContent()`**) and extends **`DrawScope`**; this way, we gain access to quite a few drawing primitives, such as **`drawLine()`**, **`drawRect()`**, and **`drawImage()`**. **`drawContent()`** draws the UI element, so depending on when it is invoked, the element appears in front of, or behind, the other drawing primitives. In my example, it is the last instruction, so the UI element (for example, **`Text()`**) is the topmost one.

Jetpack Compose also includes a modifier called **`drawBehind {}`**. It receives a lambda expression that can contain drawing primitives, just like in my example. To learn even more about the internals of Jetpack Compose, you may want to take a look at its source code. To see it, just click on **`drawBehind()`** in your code while pressing the *Ctrl* key.

This concludes my explanations of modifiers. As you have seen, they are a very elegant way to control both the visual appearance and behavior of

composable functions.

Summary

This chapter introduced you to the key principles of Jetpack Compose. We closely looked at the underlying ideas and concepts of composable functions, and you now know how they are written and used. We also focused on how to create and update the UI, as well as how Jetpack Compose achieves what other frameworks call repainting or updating the screen. When relevant app data changes, the UI changes, or so-called recomposition takes place automatically, this is one of the advantages over the traditional View-based approach, where the developer must imperatively change the component tree.

We then expanded our knowledge of the concept of modifiers. We looked at how modifier chains work and what you need to keep in mind to always get the intended results. For example, to receive clicks inside padding, `padding {}` must occur after `clickable {}` in the `modifier` chain. Finally, you learned how to implement custom modifiers.

In *Chapter 4*, *Laying Out UI Elements*, we will examine how to lay out UI elements and introduce you to the **single measure pass**. We will explore built-in layouts, but also write a custom Compose layout.