Android UI Development with **Jetpack Compose**

Bring declarative and native UIs to life quickly and easily on Android using Jetpack Compose

Thomas Künneth

Table of Contents

2
Understanding the Declarative Paradigm

Jetpack Compose marks a fundamental shift in Android UI development. While the traditional view-based approach is centered around components and classes, the new framework follows a declarative approach.

In *Chapter 1*, *Building Your First Compose App*, I introduced you to composable functions, the basic building blocks of a Compose-based UI. In this chapter, we will briefly review how Android UIs are implemented with traditional classes and techniques. You will learn about some issues of this approach, and how a declarative framework helps overcome them.

The main sections of this chapter are as follows:

- Looking at the Android view system
- Moving from components to composable functions
- Examining architectural concepts

We'll start by looking at my second sample app, *Hello View*. It is a re-implementation of the *Hello* app from *Chapter 1*, *Building Your First Compose App*. *Hello View* uses views, an XML **layout file**, and **view binding**.

Next, we will cover key aspects of **components**, which are UI building blocks in the view-based world. You will learn about the similarities and differences of composable functions, and we will find out how composable functions can overcome some of the limitations of component-centric frameworks.

Finally, we will look at the different layers of the Android framework and how they relate to building UIs. By the end of this chapter, you will have gathered enough background information to explore the key principles of Jetpack Compose, which is the topic of the next chapter.

Technical requirements

Please refer to the *Technical requirements* section in *Chapter 1*, *Building Your First Compose App*, for information about how to install and set up Android Studio and how to get the sample app. 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_02`.

Looking at the Android view system

The traditional approach to building Android UIs is to define component trees and modify them during runtime. While this can be done completely programmatically, the preferred way is to create layout files. They use XML tags and attributes to define which UI elements should be presented on screen. Let's take a look:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <TextView
    android:id="@+id/message"
    style="@style/TextAppearance.AppCompat.Medium"
    android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"
        android:textAlignment="center"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintBottom_toTopOf="@id/name"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.5"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.5"
        app:layout_constraintVertical_chainStyle="packed" />
    ...
</androidx.constraintlayout.widget.ConstraintLayout>
```

Layout files define a hierarchical structure (a tree). In the previous XML snippet, the root node (`ConstraintLayout`) contains only one child (`TextView`). The complete XML file of *Hello View* has two more children, an `EditText` component and a `Button` component. Layout files of real-world apps can be quite nested, containing dozens of children.

Generally speaking, `...Layout` elements are responsible for sizing and positioning their children. While they may have a visual representation (for example, background color or a border), they usually don't interact with the user. `ScrollView` is one of the exceptions to that rule. All other (non `...Layout`) elements such as buttons, checkboxes, and editable text fields not only allow for user interaction – it's their purpose.

Both layout and non-layout elements are called components. We will return to this term in the *Moving from components to composable functions* section. But before that, let's see how layout files are used in apps.

Inflating layout files

Activities are one of the core building blocks of an Android app. They implement a quite sophisticated lifecycle, which is reflected by a couple of methods we can override.

Typically, `onCreate()` is used to prepare the app and to show the UI by invoking `setContentView()`. This method can receive an ID representing a layout file, for example, `R.layout.main`. Because of this, you must define variables pointing to the UI elements you wish to access. This may look like the following:

```
private lateinit var doneButton: Button
...
val doneButton = findViewById(R.id.done)
```

It turned out that this doesn't scale well for bigger apps. There are two important issues to remember:

- You may face crashes during runtime if the variable is accessed before it has been initialized.
- The code quickly becomes lengthy if you have more than a few components.

Sometimes, you can prevent the first issue by using local variables, as follows:

```
val doneButton = findViewById<Button>(R.id.done)
```

This way, you can access the UI element immediately after the declaration. But the variable exists only in the scope in which it has been defined – a block or a function. This may be problematic because you often need to modify a component outside `onCreate()`. That's because in a component-based world, you modify the UI by modifying the properties of a component. It turned out that often the same set of changes are necessary for different parts of the app, so to avoid code duplication, they are refactored into methods, which need to know the component to change it.

To solve the second issue – that is, to spare the developer from the task of keeping references to components – Google introduced view binding. It belongs to Jetpack and debuted in Android Studio 3.6. Let's see how it is used:

```
class MainActivity : AppCompatActivity() {

  private lateinit var binding: MainBinding

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = MainBinding.inflate(layoutInflater)
    setContentView(binding.root)
    ...
    enableOrDisableButton()
  }
  ...
}
```

No matter how complex the UI of an activity is, we need to keep only one reference. This variable is usually called `binding`, which is initialized by invoking `inflate()` of a `...Binding` instance. The `MainBinding` class in my example is automatically generated and updated, when `main.xml` is modified. Every layout file gets a corresponding `...Binding` class. To enable this mechanism, the `viewBinding` build option must be set to `true` in the module-level `build.gradle` file:

```
android {
   ...
   buildFeatures {
     viewBinding true
   }
}
```

So, after you have inflated a layout file by invoking `...Binding.inflate()` and assigned it to an instance variable, you can access all of its components via their IDs using this variable. IDs are set using the XML attribute `android:id` (for example, `android:id="@+id/message"`).

Important Note

There is an important difference between the old-fashioned `findViewById()` and view binding. If you use the latter one, you must pass the root component (`binding.root`) to `setContentView()`, rather than an ID representing the layout file (`R.layout.main`).

In this section, I have shown you how to obtain references to UI elements. The next section, *Modifying the UI*, will explain how to make use of this.

Modifying the UI

In this section, we will see how to make changes to a View-based UI. Let's start by looking at the `enableOrDisableButton()` function, which is invoked in `onCreate()`. Its name gives you a clue regarding its purpose – enabling or disabling a button. But why do we need this? *Hello View* is a reimplementation of the *Hello* app from *Chapter 1*, *Building Your First Compose App*, but it has one additional feature. As long as the user has not entered at least one non-blank character, **Done** can't be clicked:

```
private fun enableOrDisableButton() {
   binding.done.isEnabled = binding.name.text.isNotBlank()
}
```

`binding.done` refers to the button during runtime. It can be clicked only if `isEnabled` is `true`. The text input field is denoted by `binding.name`. Its `text` property reflects what the user has already entered. `isNotBlank()` tells us if at least one non-whitespace character is present.

In the code I have shown you so far, `enableOrDisableButton()` is called only at the end of `onCreate()`. But we also need to invoke the function whenever the user has input something. Let's see how to do this (please note that the following code snippets belong inside `onCreate()` so that they are executed when the activity is created):

```
binding.name.run {
  setOnEditorActionListener { _, _, _ ->
    binding.done.performClick()
    true
  }
  doAfterTextChanged {
    enableOrDisableButton()
  }
  visibility = VISIBLE
}
```

Text input fields can modify certain aspects of the onscreen keyboard. For example, to have it show a **Done** key instead of the usual **Enter**, we add an `android:imeOptions="actionDone"` attribute to the layout file. To react to clicks on this key, we need to register code by invoking `setOnEditorActionListener()`. Then, `binding.done.performClick()` simulates clicks on the **Done** button. You will see shortly why I do this.

The lambda function we pass to `doAfterTextChanged()` is invoked every time the user enters or deletes something in the text input field. When this happens, `enableOrDisableButton()` is called, which makes the button clickable if the text currently present in the input field is not blank.

Finally, `visibility = VISIBLE` occurs inside `binding.name.run {`, so it makes the text input field visible. This is the desired state when the activity is created.

Now, let's turn to code related to the **Done** button:

```
binding.done.run {
  setOnClickListener {
    val name = binding.name.text
    if (name.isNotBlank()) {
        binding.message.text = getString(R.string.hello,
                                          name)
        binding.name.visibility = GONE
        it.visibility = GONE
      }
    }
    visibility = VISIBLE
  }
```

When **Done** is clicked, we test whether the text input field contains at least one character besides whitespace. If this is the case, the greeting message will be constructed and displayed. Also, both the button and the text input field are hidden; they need to disappear after the user has entered a name, because then only the greeting message should be visible. Making a component visible or invisible is done by modifying the `visibility` property: `visibility = VISIBLE` makes the **Done** button visible. This is the desired state when the activity is created.

Do you remember that I promised to explain why I invoke `performClick()` inside the lambda function for `setOnEditorActionListener`? This way, I can reuse the code inside the button listener without refactoring it into a separate function and calling it instead, which certainly is a viable alternative.

Before we move on, let's recap what have we seen so far:

- The UI is defined in an XML file.
- At runtime, it is inflated to a component tree.
- To change the UI, attributes of all related components must be modified.
- Even if a UI element is not visible, it remains part of the component tree.

This is why common UI frameworks are called **imperative**. Any change to the UI is done by deliberately modifying the attributes of all components involved. As you can see in my example, this works quite well for small apps. But the more UI elements an app has, the more demanding it will be to keep track of such changes. Let me explain. Changes in domain data (adding an item to a list, deleting text, or loading an image from a remote service) require changes in the UI. The developer needs to know which portion of domain data relates to which UI element and must then modify the component tree accordingly. The bigger an app becomes, the more difficult this is.

Also, without clear architectural guidance, the code for changing the component tree almost always eventually mixes with code that modifies data the app is using. This makes it even more demanding and error-prone to maintain and further develop the app. In the next section, we will turn to composable functions. You will learn how they differ from components and why this helps overcome weaknesses in the imperative approach.

Moving from components to composable functions

So far, I explained the word *component* by saying that it refers to UI elements. In fact, the term is used in quite a few other areas. Generally speaking, components structure systems by separating distinct portions or parts of them. The inner workings of a component are typically hidden from the outside (known as the **black box principle**).

> **Tip**
>
> To learn more about the black box principle, please refer to `https://en.wikipedia.org/wiki/Black_box`.

Components communicate with other parts of the system by sending and receiving messages. The appearance or behavior of a component is controlled through a set of attributes, or **properties**.

Consider `TextView`. We set text by modifying the `text` property and we control its visibility through `visibility`. What about sending and receiving messages? Let's look at `Button`. We can react to clicks (receive a message) by registering (sending a message) an `OnClickListener` instance. The same principle applies to `EditText`. We configure its appearance through setting properties (`text`), send a message by invoking `setOnEditorActionListener()`, and receive one through the lambda expression we passed as a parameter.

Message-based communication and configuration via properties make components very tool-friendly. In fact, most component-based UI frameworks work well with drawing board-like editors. The developer defines a UI using drag and drop. Components are configured using property sheets. *Figure 2.1* shows the Layout Editor in Android Studio. You can switch between a **Design** view, browse **Code** (an XML file), or a combination of both (**Split**):

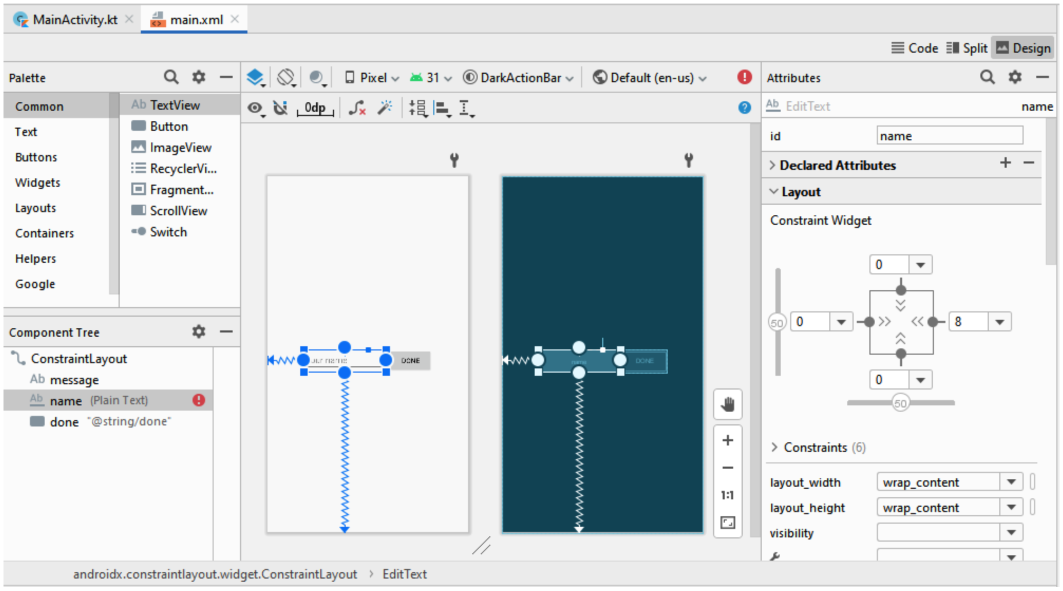

Figure 2.1 – The Layout Editor in Android Studio

We now have a more precise understanding of how the *component* term is used in the context of UIs. Building on this foundation, we will now look at component hierarchies.

Component hierarchies

If you compare the XML attributes of `ConstraintLayout`, `TextView`, and `EditText`, you will find unique attributes per tag, one example being `android:inputType`. On the other hand, `android:layout_width` and `android:layout_height` are present in all three tags, defining the size of the corresponding element. Size and position are relevant for all components.

Yet, specific attributes influence visual appearance or behavior; this is *not* relevant for all kinds of UI elements, only a subset. Here's an example: text fields and buttons will want to show or receive text. A `FrameLayout` UI element won't. Think of it this way: the *more specialized* an attribute is, the *less likely is its reuse* in another component. However, general ones (such as `width`, `height`, `location`, or `color`) will be needed in most UI elements.

Based on its attributes, each component has a level of specialization. For example, `EditText` is more specific than `TextView` because it can handle text input. `Button` is a general-purpose button; clicking on it triggers some action. On the other hand, a `CheckBox` component can be either checked or unchecked. This type of button can represent two states. A `Switch` component has two states, too. It's a toggle switch widget that can select between two options.

The degree of specialization can be modeled easily in object-oriented programming languages through inheritance. A more specialized UI element (class) extends a general element. Therefore, many often-used UI frameworks have been implemented in Java, C++, or C# (object-oriented languages). It is important to note, though, that component-like concepts can be achieved with other types of programming languages too. So, object orientation may be considered a benefit, but it's not a necessity.

At this point, you may be thinking, *Didn't he mix two different things? How are tags and attributes of Android layout files related to classes?* Allow me to explain. Earlier, I said that an XML file is **inflated** to a component tree. To be more precise – it becomes an *object* tree. The tags in the XML file represent class names and its attributes correspond to members of that class. `inflate()` creates a tree of objects based on this information.

So, Android layout files describe component trees outside of Java or Kotlin files using a different syntax (an XML syntax). But they are not declarative in the same way Jetpack Compose is because layout files define a UI regardless of the current state. For example, they do not take into account that a button should be disabled because a text field is empty. A Compose UI, on the other hand, is declared *based* on that.

The remaining part of this section will look closer at some of Android's UI components and how they are related. Before that, let's recap what we have learned so far:

- All Android views are classes.
- Tags in layout files represent classes and attributes are their members.
- `inflate()` creates an object tree.
- Changes to the UI are achieved by modifying this tree.

Some of Android's UI elements are quite specific. `RatingBar`, for example, allows the user to rate something by selecting a certain number of stars. Others are way more general; for example, `ImageView` just displays image resources, and `FrameLayout` blocks out an area on the screen to display a stack of children.

To understand how Android's UI elements are related, let's look at the ones used in *Hello View* in a little more detail. We'll start with `ConstraintLayout`:

```
java.lang.Object
  ↳  android.view.View
      ↳  android.view.ViewGroup
          ↳  androidx.constraintlayout.widget.ConstraintLayout
```

The root of *all* classes in Java is `java.lang.Object`. Significant parts of the Android framework are based on Java and its class library. So, all views directly or indirectly extend `java.lang.Object`. The immediate parent of `ConstraintLayout` is `android.view.ViewGroup`, which in turn is a sibling of `android.view.View`.

Now, let's look at `android.widget.Button`.

```
java.lang.Object
  ↳  android.view.View
      ↳  android.widget.TextView
          ↳  android.widget.Button
```

Its direct ancestor is `android.widget.TextView`, which extends `android.view.View`. Are we seeing a pattern here? `android.view.View` seems to be the root of all Android UI elements. Let's check our hypothesis by examining another component:

```
java.lang.Object
  ↳  android.view.View
      ↳  android.widget.TextView
          ↳  android.widget.EditText
```

As you can see, components showing or receiving text usually extend `android.widget.TextView`, whose parent is `android.view.View`.

Important Note

`android.view.View` is the root of all Android UI elements. All components that position and size their children extend `android.view.ViewGroup`.

So far, structuring UI elements in a hierarchy based on specialization seems to work well. Unfortunately, this approach does have limitations. We'll turn to them in the following section.

Limitations of component hierarchies

Buttons usually show text. Therefore, it seems natural to extend a more general text component. As we have seen in the previous section, Android does just that. What if your app requires a button that has no text and shows an image instead? In such scenarios, you can use `ImageButton`:

```
java.lang.Object
   ↳  android.view.View
      ↳  android.widget.ImageView
         ↳  android.widget.ImageButton
```

The class extends `android.widget.ImageView`. This makes sense, as the purpose of this component is to show just an image, quite like `Button` and text. But what if we want to show a button that contains both text and image? The closest common ancestor of `ImageButton` and an ordinary text button is `android.view.View`, the root of the Android UI element hierarchy. Therefore, everything `Button` inherits from `TextView` is not immediately available to `ImageButton` (and vice versa).

The reason is that Java is based upon **single inheritance**: a class extends exactly one other class. If `Button` wanted to take advantage of the features of `TextView` and `ImageView`, it would need to extend both, which it can't. Does this mean that things would be different if Java supported **multiple inheritance**? We could combine the behavior of several components, but we still wouldn't be able to reuse functionality tied to *individual* attributes, methods, or sets of them. Let's see why this is important.

The `View` class knows about padding (providing space to the inside of its bounds) but not about margins (space to the outside of its bounds). Margins are defined in `ViewGroup`. Hence, if a component wants to use them, it must extend `ViewGroup`. But in doing so, it inevitably inherits all other features of this class (for example, the ability to layout children), regardless of needing them or not. The underlying issue is that in a component-centric framework, the combination of *individual features* of one or more components to create a more specialized UI element is not possible because you cannot cut out these features. The reason for this is that reuse happens at a component level.

To make individual features reusable, we need to put aside the notion of components. That's what, for example, Flutter (the very successful cross-platform alternative to Jetpack Compose) does. Its UI framework is fully declarative, still class-based. Flutter relies on a simple principle called **composition over inheritance**. It means the look and the behavior of a UI element (and the complete UI) are defined by combining simple building blocks, such as `Container`, `Padding`, `Align`, or `GestureDetector`, rather than modifying a parent.

In Jetpack Compose, we combine simple building blocks too. Instead of classes, we use composable functions. Before we turn to them, I would like to briefly show you another potential issue of components.

As you have seen, in class-based UI component frameworks, specialization is modeled through inheritance. The specialized version of a class (which may have new features, a new look, or behave slightly different than the ancestor) extends a more general version of the class. However, most object-oriented programming languages provide means to prohibit this; for example, if a Java class is marked final or a Kotlin class is not open, they cannot be extended.

So, the framework developer can make a deliberate decision to prevent further inheritance. `android.widget.Space`, a lightweight `View` subclass to create gaps between UI elements, is final. The same applies to `android.view.ViewStub`. It's an invisible, zero-sized `View` used to lazily inflate layout resources at runtime. Fortunately, most of Android's UI elements can be extended. And for both examples, it seems unlikely that we would want to extend them. Hence, you may not face this potential issue at all. The point is that in a framework based upon composition rather than inheritance, it doesn't matter.

Composing UIs with functions

Now it's time to return to composable functions. In this section, we will look at my sample app *Factorial* (*Figure 2.2*). When the user picks a number between 0 and 9, its factorial (the product of it and all the integers below it greater than 0) is computed and output, like so:

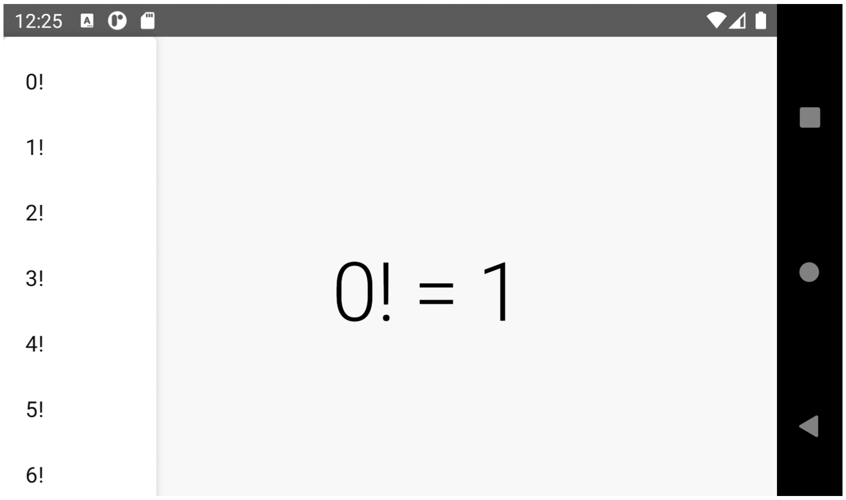

Figure 2.2 – The Factorial app

Here is the simple function that creates the output text:

```
fun factorialAsString(n: Int): String {
  var result = 1L
  for (i in 1..n) {
    result *= i
  }
  return "$n! = $result"
}
```

The factorial of an `n` non-negative integer value is the product of all positive integers less than or equal to `n`. So, the result can be computed easily by multiplying all integers between `1` and `n`. Please note that the maximum value of a Kotlin `Long` type is 9,223,372,036,854,775,807. Hence, my implementation does not work if `result` would need to be bigger than that.

Next, I'll show you how the UI is composed:

```
@Composable
fun Factorial() {
  var expanded by remember { mutableStateOf(false) }
  var text by remember {
    mutableStateOf(factorialAsString(0)) }
  Box(
    modifier = Modifier.fillMaxSize(),
    contentAlignment = Alignment.Center
  ) {
    Text(
      modifier = Modifier.clickable {
        expanded = true
      },
      text = text,
      style = MaterialTheme.typography.h2
    )
    DropdownMenu(
      expanded = expanded,
      onDismissRequest = {
        expanded = false
      }) {
        for (n in 0 until 10) {
          DropdownMenuItem(onClick = {
            expanded = false
            text = factorialAsString(n)
          }) {
            Text("${n.toString()}!")
          }
        }
      }
    }
  }
```

The `Factorial()` composable function contains one predefined composable, `Box()`, which in turn has two children, `Text()` and `DropdownMenu()`. I briefly introduced you to `Text()` and `Box()` in *Chapter 1*, *Building Your First Compose App*. So let's concentrate on `DropdownMenu()`.

A drop-down menu (the equivalent to a `Spinner`) displays a list of entries in a compact way. It appears upon interaction with an element, such as the following:

- An icon or a button
- When the user performs a specific action

In my example, the `Text()` composable must be clicked.

The content of a menu can either be provided by a `for` loop statement or by adding it one by one. Often, but not necessarily, `DropdownMenuItem()` is used. If the menu is expanded (that is, open or visible), it is controlled by the `expanded` parameter. `onDismissRequest` is used to react to closing the menu without selecting something. `DropdownMenuItem()` receives a click handler via the `onClick` parameter. That code is executed when the item is clicked.

So far, I have presented quite a lot of information about composable functions to you. Before we move on, let's recap what we know so far:

- The entry point of a Compose UI is a composable function.
- From there, other composable functions are called.
- Often, composable functions receive *content* that is other composables.
- The order of invocation controls where a UI element will be in relation to other UI elements.

Let's continue with how `Factorial()` works. It defines two variables, `expanded` and `text`. But how are they used? While an Android layout file defines a component tree in its initial state, a composable UI is always declared using actual data. This means that there is no need to set up or prepare the UI before it can be displayed for the first time. Whenever it is displayed, it looks the way you want. Let's see how this works.

Most composable functions are configured by a set of parameters. Some of them are mandatory; others can be omitted. The important thing is that the composable is always called with actual values. On the other hand, components (that is, views) are initialized when they are created. And they remain this way until they are deliberately changed by altering the value of properties. That's why an app needs to keep references to all components (UI elements) it wishes to modify. But how is a Compose UI updated?

The process of updating a Compose UI is called **recomposition**. It takes place automatically whenever a composable function being part of the UI needs to be updated. This is the case when some of the values that influence its look or behavior (parameters) change. If you always pass the same text to `Text()`, there is no need to recompose it. If, on the other hand, you pass something Jetpack Compose knows it can change, the Compose runtime will initiate an update, a recomposition, when that change happens. Values that change over time are called **state**. You can create state, for example, using `mutableStateOf()`. To refer to state in a composable, you need to `remember` it in that composable function.

Both `expanded` and `text` contain state. When these variables are used as parameters for composable functions, those composables will be recomposed whenever the value of these variables changes. Setting `expanded` to `true` brings the drop-down menu on screen immediately. This is done inside a lambda function passed to `clickable {}`. I will be discussing this in the next section. Giving `text` a new value changes the display of `Text()` because we pass the variable `text` as the value of the equally named parameter. This happens, for example, inside the code block passed to `onClick`.

Getting rid of a component tree (that needs to be updated deliberately) in favor of declaring a UI based on state and thus getting updates upon state changes for free is possibly one of the most exciting advantages of the declarative approach. In the next section, I will explain a few more architectural principles of component-based and declarative UI frameworks.

Examining architectural aspects

In the *Component hierarchies* section, I showed you that component-based UI frameworks rely on specialization. General features and concepts are implemented in the root component or one of its immediate successors. Such general features include the following:

- Location and size on screen
- Basic visual aspects like background (color)
- Simple user interactions (reacting to clicks)

Any component will provide these features, either in a specialized way or in its basic implementation. Android's view system is class-based, so changing functionality is done by overriding the methods of the parent.

Composable functions, on the other hand, do not have a shared set of properties. By annotating a function with `@Composable`, we make certain parts of Jetpack Compose aware of it. But besides not specifying a return type, composables seem to have few things in common. However, this would have been a pretty short-sighted architectural decision. In fact, Jetpack Compose makes providing a simple, predictable API really easy. The remaining part of this section illustrates this by showing you how to react to clicks, and how to size and position UI elements.

Reacting to clicks

Android's `View` class contains a method called `setOnClickListener()`. It receives a `View.OnClickListener` instance. This interface contains one method, `onClick(View v)`. The implementation of this method provides the code that should be executed when the view is clicked. Additionally, there is a view property called `clickable`. It is accessed through `setClickable()` and `isClickable()`. If `clickable` is set to `false` after the listener has been set, the click event will not be delivered (`onClick()` is not called).

Jetpack Compose can provide click handling in two ways. Firstly, composable functions that require it (because it is a core feature for them) have a dedicated `onClick` parameter. Secondly, composables that usually do not require click handling can be amended with a modifier. Let's start with the first one.

```
@Composable
@Preview
fun ButtonDemo() {
  Box {
    Button(onClick = {
        println("clicked")
      }) {
        Text("Click me!")
      }
    }
}
```

Please note that `onClick` is mandatory; you must provide it.

If you want to show the button but the user should not be able to click it, the code looks like this:

```
Button(
  onClick = {
    println("clicked")
  },
  enabled = false
) {
  Text("Click me!")
}
```

Figure 2.3 shows what the button looks like when `enabled` is either `true` or `false`:

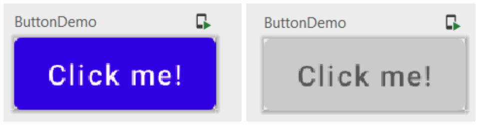

Figure 2.3 – A button with enabled = true or false

`Text()` doesn't have an `onClick` property. If you want to make it clickable (like I do in the *Factorial* app), you pass `clickable { ... }` to the `modifier` parameter:

```
modifier = Modifier.clickable { ...
```

Modifiers, as their name suggests, provide an infrastructure for influencing both the visual appearance and behavior of composable functions. I will show you another example for modifiers in the next section. *Chapter 3*, *Exploring the Key Principles of Compose*, covers them in much greater detail.

Sizing and positioning UI elements

In component-centric UI frameworks, size and location onscreen (or relative to another component) are core properties. They are defined in the root component (on Android, the `View` class). Descendants of `ViewGroup` size and position their children by changing their corresponding properties. For example, `RelativeLayout` is based upon instructions such as `toStartOf`, `toEndOf`, or `below`. `FrameLayout` draws its children in a stack. And `LinearLayout` lays out children horizontally or vertically. So, `...Layouts` are containers with the ability to size and position their children.

Jetpack Compose has a very similar concept. You have already learned about `Row()` and `Column()`, which lay out their content horizontally or vertically. `Box()` is similar to `FrameLayout`. It organizes its content in the order it appears in code. The position inside the box is controlled by `contentAlignment`:

```
@Composable
@Preview
fun BoxDemo() {
  Box(contentAlignment = Alignment.Center) {
    Box(
      modifier = Modifier
        .size(width = 100.dp, height = 100.dp)
        .background(Color.Green)
    )
    Box(
      modifier = Modifier
        .size(width = 80.dp, height = 80.dp)
        .background(Color.Yellow)
    )
    Text(
      text = "Hello",
      color = Color.Black,
      modifier = Modifier.align(Alignment.TopStart)
    )
  }
}
```

The content may override this by using `modifier = Modifier.align()`, the result of which we can see in *Figure 2.4*:

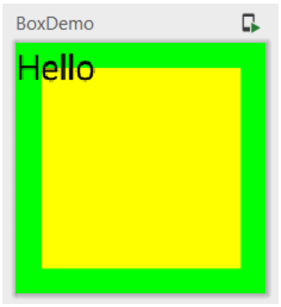

Figure 2.4 – An invisible box containing two colored boxes and text

Modifiers can also be used to request a size. In some of my examples, you may have spotted `Modifier.fillMaxSize()`, which makes the composable as big as possible. `Modifier.size()` requests a particular size. Modifiers can be chained. The root of such a chain is the `Modifier` companion object. Subsequent modifiers are added using a dot.

Before closing this chapter, I would like to emphasize the benefits of the modifier concept with one more example. Did you notice the `background()` modifiers of the first and second content box? This modifier allows you to set a background color for any composable function. When you need something a composable function does not offer out of the box, you can add it with a modifier. As you can write custom modifiers, the possibilities to adjust a composable to your needs are almost endless. I will elaborate on this in the next chapter.

Summary

In this chapter, you have learned about key elements of component-centric UI frameworks. We saw some of the limitations of this approach and how the declarative paradigm can overcome them. For example, specialization takes place on a component level. If the framework is based upon inheritance, the distribution of features to children may be too broad. Jetpack Compose tackles this with the modifier mechanism, which allows us to amend functionality at a very fine-grained level; this means that composables only get the functionality they need (for example, a background color).

The remaining chapters of this book are solely based on the declarative approach. In *Chapter 3*, *Exploring the Key Principles of Compose*, we will take an even closer look at composable functions and examine the concepts of composition and recomposition. And, as promised, we will also dive deep into modifiers.