Chapter 10: Testing and Debugging Compose Apps

Programming is a very creative process. Implementing great-looking **user interfaces** (**UIs**) with slick animations is pure fun with Jetpack Compose. However, making an outstanding app involves more than just writing code. Testing and debugging are equally as important because no matter how carefully you design and implement your app, bugs and glitches are inevitable, at least in non-trivial programs. Yet there is nothing to fear, as there are powerful tools you can wield to check if your code is acting as intended.

This chapter introduces you to these tools. Its main sections are listed here:

- Setting up and writing tests
- Understanding semantics
- Debugging Compose apps

In the first main section, I will walk you through important terms and techniques regarding testing. We will set up the infrastructure, write a simple unit test, and then turn to Compose specifics—for example, `createComposeRule()` and `createAndroidComposeRule()`.

The *Understanding semantics* section builds on these foundations. We look at how composable functions are selected—or found—in a test, and why making your app accessible also helps to write better tests. You will also learn about actions and assertions.

Failing tests often hint at bugs unless, of course, the failure is intentional. If you suspect the code being checked by a test is buggy, a debugging session is due. The final main section, *Debugging Compose apps*, explains how to examine your Compose code. We will be revisiting the semantics tree, discussed in the *Understanding semantics* section. Finally, I will show you how to take advantage of **`InspectorInfo`** and **`InspectorValueInfo`**.

Technical requirements

This chapter is based on the **`TestingAndDebuggingDemo`** sample. Please refer to *Technical requirements* section of *Chapter 1*, *Building Your First Compose App*, for information about how to install and set up Android Studio and how to get the repository accompanying this book.

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_10.

Setting up and writing tests

As a software developer, you probably enjoy writing code. Seeing an app gain functionality feels very rewarding, probably more than writing tests—or worse, finding bugs—yet testing and debugging are essential. Eventually, your code will contain bugs, because all non-trivial programs do. To make your developer life easier, you need to familiarize yourself with writing tests and with debugging your own and others' code. Testing an app has various facets that correspond to different types of tests, as outlined here:

- **Unit test**: You need to make sure that the business logic works as expected. This, for example, means that formulae and calculations always produce correct results.
- **Integration tests**: Are all building blocks of the app properly integrated? Depending on what the app does, this may include accessing remote services, talking to a database, or reading and writing files on the device.

- **UI tests**: Is the UI accurate? Are all UI elements visible on all supported screen sizes? Do they always show the right values? Do interactions such as button clicks or slider movements trigger the intended function? And something very important: are all parts of the app accessible?

The number of tests varies among types. It has long been claimed that, ideally, most of your tests should be unit tests, followed by integration tests. This leads to the perception of a **test pyramid**, with unit tests being its foundation and UI tests the tip. As with all metaphors, the test pyramid has seen both support and harsh criticism. If you want to learn more about it, and testing strategies in general, please refer to the *Further reading* section at the end of this chapter. Jetpack Compose tests are UI tests. So, while you likely write many corresponding test cases, testing the underlying business logic using unit tests may well be even more important.

To make testing reliable, comprehensible, and reproducible, automation is used. In the next section, I will show you how to write unit tests using the *JUnit 4* testing framework.

Implementing unit tests

Units are small, isolated pieces of code—usually a function, method, subroutine, or property, depending on the programming language. Let's look at a simple Kotlin function in the following code snippet:

```
fun isEven(num: Int): Boolean {
  val div2 = num / 2
  return (div2 * 2) == num
}
```

`isEven()` determines if the passed `Int` value is even. If this is the case, the function returns `true`; otherwise, it returns `false`. The algorithm is based on the fact that only even `Int` values can be divided by `2` without a remainder. Assuming we use the function often, we certainly want to make sure that the result is always correct. But how do we do that (how do we test that)? To verify `isEven()` exhaustively, we would need to check every possible input value, ranging from `Int.MIN_VALUE` to `Int.MAX_VALUE`. Even on fast computers, this may take some time. Part of the art of writing good unit tests is to identify all the important boundaries and transitions. Regarding `isEven()`, these might be the following ones:

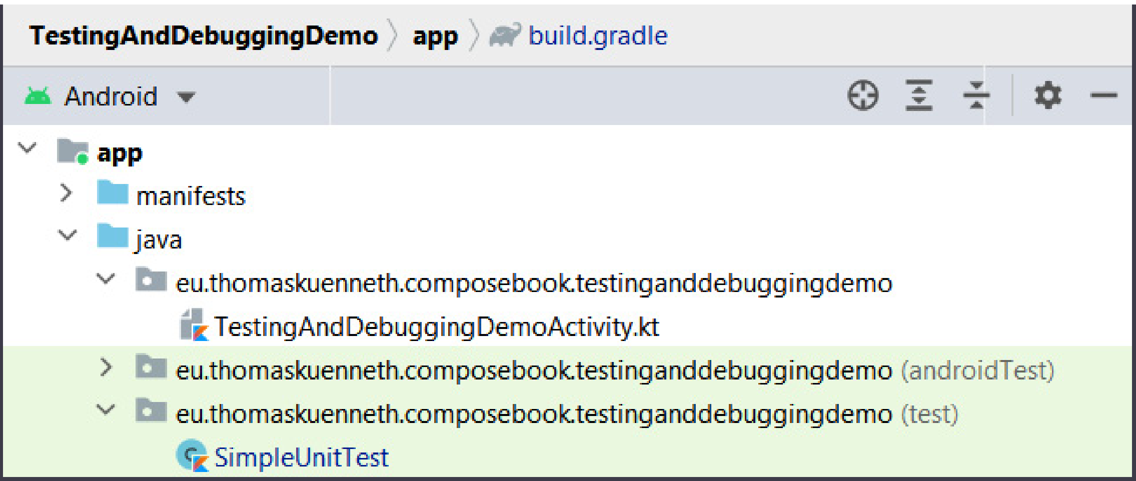
- `Int.MIN_VALUE` and `Int.MAX_VALUE`
- One negative even and one negative odd `Int` value
- One positive even and one positive odd `Int` value

To write and execute unit tests, you should add the following dependencies to your module-level `build.gradle` properties file:

```
androidTestImplementation
"androidx.test.ext:junit:1.1.3"
androidTestImplementation "androidx.compose.ui:ui-
test-
  junit4:$compose_version"
debugImplementation "androidx.compose.ui:ui-test-
  manifest:$compose_version"
testImplementation 'junit:junit:4.13.2'
androidTestImplementation
"androidx.test.espresso:espresso-
  core:3.4.0"
```

Depending on which types of tests you will be adding to your app project, some of the preceding dependencies will be optional. For example, `androidx.test.espresso` is needed only if your app also contains old-fashioned Views you wish to test (such as in interoperability scenarios).

Unit tests are executed on your development machine. Test classes are placed inside the `app/src/test/java` directory and are available through the **Project** tool window, as illustrated in the following screenshot. The Android Studio project assistant configures your projects accordingly and creates a test class, which I have renamed `SimpleUnitTest`:



```
app
  manifests
  java
    eu.thomaskuenneth.composebook.testinganddebuggingdemo
      TestingAndDebuggingDemoActivity.kt
    eu.thomaskuenneth.composebook.testinganddebuggingdemo (androidTest)
    eu.thomaskuenneth.composebook.testinganddebuggingdemo (test)
      SimpleUnitTest
```

Figure 10.1 – Unit tests in the Android Studio Project tool window

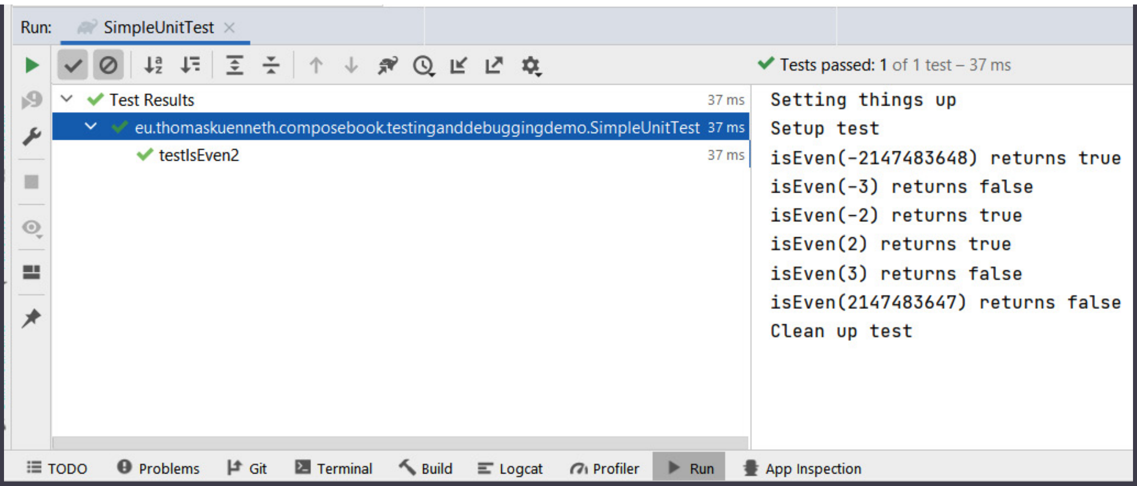
Let's look at the class in the following code snippet:

```
Package
  eu.thomaskuenneth.composebook.testinganddebuggingde
mo
import org.junit.*
import org.junit.Assert.assertEquals
class SimpleUnitTest {
  companion object {
    @BeforeClass
    @JvmStatic
    fun setupAll() {
      println("Setting things up")
    }
  }
  @Before
  fun setup() {
    println("Setup test")
  }
  @After
  fun teardown() {
    println("Clean up test")
  }
  @Test
  fun testListOfInts() {
    val nums = listOf(Int.MIN_VALUE, -3, -2, 2, 3,
                       Int.MAX_VALUE)
    val results = listOf(true, false, true, true,
false,
                          false)
    nums.forEachIndexed { index, num ->
      val result = isEven(num)
      println("isEven($num) returns $result")
      assertEquals(result, results[index])
    }
  }
}
```

A test class contains one or more tests. A **test** (also called a **test case**) is an ordinary Kotlin function, annotated with **`@Test`**. It checks certain well-defined situations, conditions, or criteria. Tests should be isolated, which means they should not rely on previous ones. My example tests if **`isEven()`** returns correct results for six input values. Such checks are based on **assertions**. An assertion formulates expected behavior. If an assertion is not met, the test fails.

If you need something to be done before or after each test, you can implement functions and annotate them with **`@Before`** or **`@After`**. You can achieve something similar using **`@Rule`**. We will be looking at this in the following section. To run code before all tests, you need to implement a companion object with a function annotated with **`@BeforeClass`** and **`@JvmStatic`**. **`@AfterClass`** is useful for cleanup purposes after all tests have been run.

You can run a unit test by right-clicking on the test class in the **Project** tool window and choosing **Run '...'**. Once a launch configuration for the test class has been created, you can also run the tests using the menu bar and the toolbar. Test results are presented in the **Run** tool window, as illustrated in the following screenshot:



```
Setting things up
Setup test
isEven(-2147483648) returns true
isEven(-3) returns false
isEven(-2) returns true
isEven(2) returns true
isEven(3) returns false
isEven(2147483647) returns false
Clean up test
```

Figure 10.2 – Test results in the Android Studio Run tool window

Although the test passes, my implementation of **`isEven()`** may still not be flawless. While the test checks the upper and lower bounds, it leaves the transition between negative and positive numbers untested. Let's correct this and add another test, as follows:

```
@Test
fun testIsEvenZero() {
  assertEquals(true, isEven(0))
}
```

Fortunately, this test passes, too.

IMPORTANT NOTE

Pay close attention to the parameters a unit receives and the result it produces. Always test boundaries and transitions. Make sure to cover all code paths (if possible) and watch for pitfalls such as exceptions due to invalid arguments (for example, division by zero or wrong number formats).

Please remember that composable functions are top-level Kotlin functions, so they are prime candidates for unit tests. Let's see how this works. In the next section, you will learn to test a simple Compose UI.

Testing composable functions

The **`SimpleButtonDemo()`** composable (which belongs to the **`TestingAndDebuggingDemo`** sample) shows a box with a button centered inside. Clicking the button for the first time changes its text from **A** to **B**. Subsequent clicks toggle between **B** and **A**. The code is illustrated in the following snippet:

```
@Composable
fun SimpleButtonDemo() {
  val a = stringResource(id = R.string.a)
  val b = stringResource(id = R.string.b)
  var text by remember { mutableStateOf(a) }
  Box(
    modifier = Modifier.fillMaxSize(),
    contentAlignment = Alignment.Center
  ) {
    Button(onClick = {
      text = if (text == a) b else a
    }) {
      Text(text = text)
    }
  }
}
```

The text is stored as a mutable **`String`** state. It is changed inside the **`onClick`** block and used as a parameter for the **`Text()`** composable. If we want to test **`SimpleButtonDemo()`**, some aspects we likely need to check are these:

- **Initial state of the UI**: Is the initial button text **A**?
- **Behavior**: Does the first button click change the text to **B**? Do subsequent clicks toggle between **B** and **A**?

Here's what a simple test class looks like:

```
@RunWith(AndroidJUnit4::class)
class SimpleInstrumentedTest {
  @get:Rule
  val rule = createComposeRule()
  @Before
  fun setup() {
    rule.setContent {
      SimpleButtonDemo()
    }
  }
  @Test
  fun testInitialLetterIsA() {
    rule.onNodeWithText("A").assertExists()
  }
}
```

Unlike the **`SimpleUnitTest`** class from the *Implementing unit tests* section, its source code is stored inside the **`app/src/androidTest/java`** directory (contrary to **`…/test/…`** for ordinary unit tests). **`SimpleInstrumentedTest`** is an **instrumented test**. Contrary to plain unit tests, they are not executed locally on the development machine, but on the Android Emulator or a real device, because they need Android-specific functionality to run. Instrumented tests are available through the **Project** tool window, as illustrated in the following screenshot:

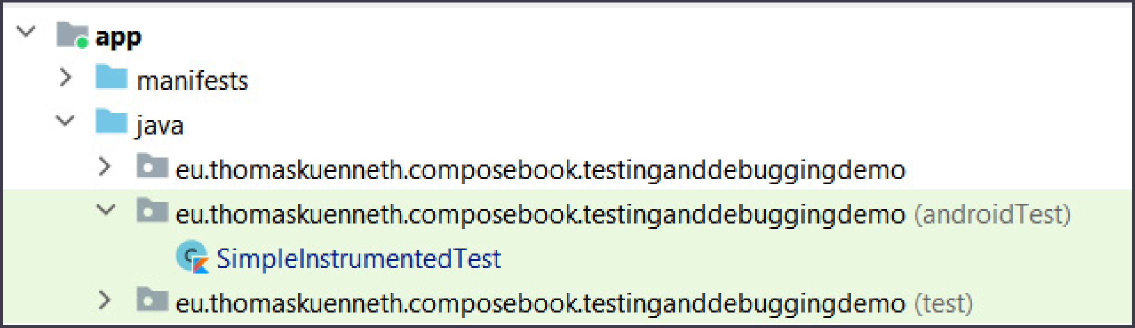
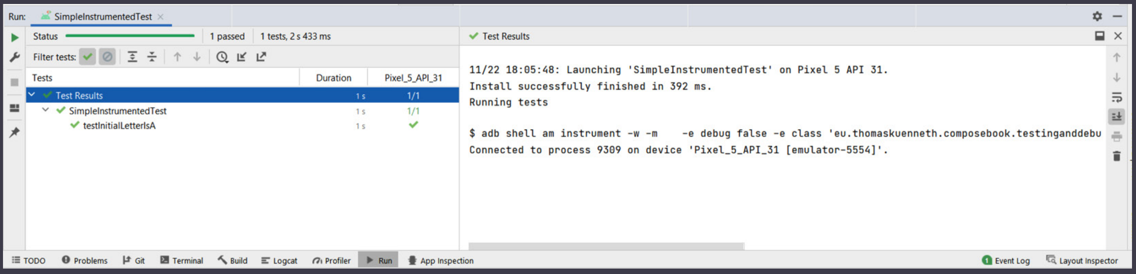

app
manifests
java
eu.thomaskuenneth.composebook.testinganddebuggingdemo
eu.thomaskuenneth.composebook.testinganddebuggingdemo (androidTest)
SimpleInstrumentedTest
eu.thomaskuenneth.composebook.testinganddebuggingdemo (test)

Figure 10.3 – Instrumented tests in the Android Studio Project tool window

You can run an instrumented test by right-clicking on the test class in the **Project** tool window and choosing **Run '…'**. Once a launch configuration for the test class has been created, you can also run the tests using the menu bar and the toolbar. Test results are presented in the **Run** tool window, as illustrated in the following screenshot:



Tests	Duration	Pixel_5_API_31
Test Results	1 s	1/1
SimpleInstrumentedTest	1 s	1/1
testInitialLetterIsA	1 s	✓

```
11/22 18:05:48: Launching 'SimpleInstrumentedTest' on Pixel 5 API 31.
Install successfully finished in 392 ms.
Running tests

$ adb shell am instrument -w -m    -e debug false -e class 'eu.thomaskuenneth.composebook.testinganddebu
Connected to process 9309 on device 'Pixel_5_API_31 [emulator-5554]'.
```

Figure 10.4 – Instrumented test results in the Android Studio Run tool window

JUnit **rules** allow you to run some code alongside a test case. In a way, this is like having `@Before` and `@After` annotations in your test class. There are several predefined rules—for example, the `TestName` rule can provide the current test name inside a test method, as follows:

```
@get:Rule
var name = TestName()
...
@Test
fun testPrintMethodName() {
  println(name.methodName)
}
```

When the `testPrintMethodName()` function runs, it prints its name. You can see the output in **Logcat**. Please remember—you need to apply the `@Rule` annotation to the property getter by adding `get:`. Failing to do so will result in a `ValidationError` (`The @Rule '…' must be public`) message during execution.

Compose tests are based on rules. `createComposeRule()` returns an implementation of the `ComposeContentTestRule` interface, which extends `ComposeTestRule`. This interface in turn extends `org.junit.rules.TestRule`. Each `TestRule` instance implements `apply()`. This method receives `Statement` and returns the same, a modified, or completely new `Statement`. However, writing your own test rules is beyond the scope of this book. To learn more, please refer to the *Further reading* section at the end of this chapter.

Which implementation of the **`ComposeContentTestRule`** interface **`createComposeRule()`** returns depends on the platform. It is **`AndroidComposeTestRule<ComponentActivity>`** on Android. That is why you should add a dependency to **`androidx.compose.ui:ui-test-manifest`** in the module-level **`build.gradle`** file. Otherwise, you may need to manually add a reference to **`ComponentActivity`** in the manifest file.

`createAndroidComposeRule()` allows you to create **`AndroidComposeTestRule`** for activity classes other than **`ComponentActivity`**. This is useful if you require the functionality of this activity in a test. On Compose for Desktop or Web, **`createComposeRule()`** may return different implementations of **`ComposeContentTestRule`**, depending on where the Compose UI is hosted. To help make your tests platform-independent, use **`createComposeRule()`** whenever possible.

Your test cases use (among others) methods provided by **`ComposeContentTestRule`** implementations. For example, **`setContent()`** sets the composable function to act as the content of the current screen—that is, the UI to be tested. **`setContent()`** should be called exactly once per test. To achieve this, just invoke it in a function annotated with **`@Before`**.

IMPORTANT NOTE

If you want to reuse your tests among platforms, try to rely only on methods defined in the **`ComposeContentTestRule`**, **`ComposeContentTestRule`**, *and* **`TestRule`** *interfaces. Avoid calling functions from the implementation.*

Next, let's look at **`testInitialLetterIsA()`**. This test case checks if the initial button text is **A**. To do so, the test must find the button, get its text, and compare it to **`"A"`**. This comparison is done with an **`assertExists()`** **assertion**. **`onNodeWithText()`** is called a **finder**. Finders work on **semantics nodes**, and you will learn more about these in the *Understanding se-*

mantics section. But first: why do we need to *find* the composable to be tested anyway?

Unlike the traditional View system, Jetpack Compose does not use references to identify individual UI elements. Please remember that such references are needed in an imperative approach to modify the component tree during runtime. But this is not how Compose works—instead, we declare how the UI should look based on state. Yet, to test if a particular composable looks and behaves as expected, we need to find it among all other children of a Compose hierarchy.

This is where the **semantics tree** comes into play. As the name implies, *semantics* give meaning to a UI element or element hierarchies. The semantics tree is generated alongside the UI hierarchy, which it describes using attributes such as **`Role`**, **`Text`**, and **`Actions`**. It is used for accessibility and testing.

Before we move on, let's briefly recap: **`onNodeWithText()`** tries to find a composable (to be, more precisely, a semantics node) with a given text. **`assertExists()`** checks if a matching node is present in the current UI. If so, the test passes. Otherwise, it fails.

Understanding semantics

In the previous section, I showed you a simple test case that checks if a button text matches a given string. Here is another test case. It performs a click on the button to see if the button text changes as expected:

```
@Test
fun testLetterAfterButtonClickIsB() {
  rule.onNodeWithText("A")
    .performClick()
    .assert(hasText("B"))
}
```

Again, we start by finding the button. **`performClick()`** (this is called an **action**) clicks it. **`Assert(hasText("B"))`** checks if the button text is **B** afterward. Assertions determine if a test passes or fails.

`onNodeWithText()` (an extension function of **`SemanticsNodeInteractionsProvider`**) returns a **`SemanticsNodeInteraction`** semantics node. The **`SemanticsNodeInteractionsProvider`** interface is the main entry point into testing and is typically implemented by a test rule. It defines two methods, as follows:

- **`onNode()`** finds and returns a semantics node (**`SemanticsNodeInteraction`**) that matches the given condition.
- **`onAllNodes()`** finds all semantics nodes that match the given condition. It returnsa **`SemanticsNodeInteractionCollection`** instance.

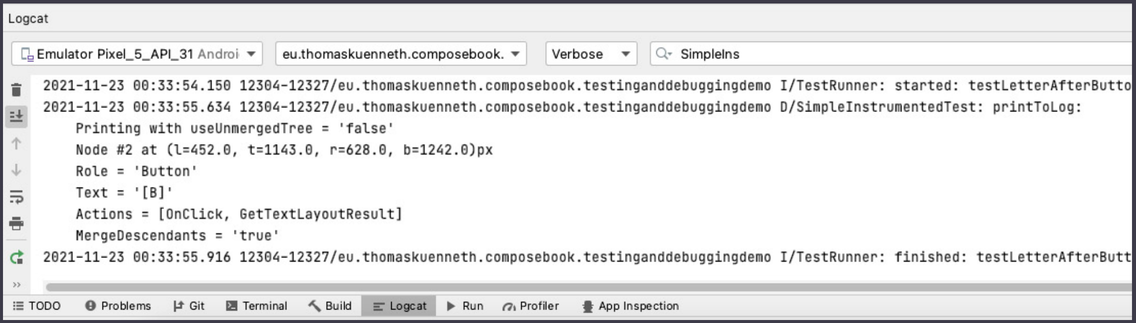
Both are called **finders** because they return (*find*) semantics nodes matching certain conditions.

Working with semantics nodes

To see what the semantics node we tested with **`testLetterAfterButtonClickIsB()`** from the previous section looks like, you can add the following expression after **`.assert(…)`**:

```
  .printToLog("SimpleInstrumentedTest")
```

The result is visible in **Logcat**, as illustrated in the following screenshot:



```
2021-11-23 00:33:54.150 12304-12327/eu.thomaskuenneth.composebook.testinganddebuggingdemo I/TestRunner: started: testLetterAfterButto
2021-11-23 00:33:55.634 12304-12327/eu.thomaskuenneth.composebook.testinganddebuggingdemo D/SimpleInstrumentedTest: printToLog:
    Printing with useUnmergedTree = 'false'
    Node #2 at (l=452.0, t=1143.0, r=628.0, b=1242.0)px
    Role = 'Button'
    Text = '[B]'
    Actions = [OnClick, GetTextLayoutResult]
    MergeDescendants = 'true'
2021-11-23 00:33:55.916 12304-12327/eu.thomaskuenneth.composebook.testinganddebuggingdemo I/TestRunner: finished: testLetterAfterButt
```

Figure 10.5 – A semantics node in Logcat

`SemanticsNodeInteraction` represents a semantics node. You can interact with a node by performing actions such as **`performClick()`** or assertions such as **`assertHasClickAction()`**, or you can navigate to other nodes such as **`onChildren()`**. Such functions are extension functions of **`SemanticsNodeInteraction`**. **`SemanticsNodeInteractionCollection`** is a collection of semantics nodes.

Let's look at another finder function, **`onNodeWithContentDescription()`**. We will be using it to test if **`Image()`** is part of the current UI. The code is illustrated in the following snippet:

```
@Composable
fun ImageDemo() {
  Image(
    painter = painterResource(id =
        R.drawable.ic_baseline_airport_shuttle_24),
    contentDescription = stringResource(id =
        R.string.airport_shuttle),
    contentScale = ContentScale.FillBounds,
    modifier = Modifier
      .size(width = 128.dp, height = 128.dp)
      .background(Color.Blue)
  )
}
```

If the UI of your app contains images, you should add content descriptions for them in most cases. Content descriptions are used, for example, by accessibility software to describe to visually impaired people what is currently presented on screen. So, by adding them, you greatly enhance the usability. Additionally, content descriptions help in finding composables. You can see these being used in the following code snippet:

```
@RunWith(AndroidJUnit4::class)
class AnotherInstrumentedTest {
  @get:Rule
  val rule = createComposeRule()
  @Test
  fun testImage() {
    var contentDescription = ""
    rule.setContent {
      ImageDemo()
      contentDescription = stringResource(id =
          R.string.airport_shuttle)
    }
    rule.onNodeWithContentDescription(contentDescript
ion)
      .assertWidthIsEqualTo(128.dp)
  }
}
```

`testImage()` first sets the content (**`ImageDemo()`**). It then finds a semantics node with the given content description. Finally, **`assertWidthIsEqualTo()`** checks if the width of the UI element represented by this node is 128 density-independent pixels wide.

TIP

Have you noticed that I used **`stringResource()`** *to obtain the content description? Hardcoded values can lead to subtle bugs in tests (for example, spelling errors or typos). To avoid them, try to write your tests in a way that they access the same values as the code being tested. But please keep in mind that under the hood,* **`stringResource()`** *relies on Android resources. So, the test case is platform-specific.*

Using **`onNodeWithText()`** and **`onNodeWithContentDescription()`** makes it easy to find composable functions that contain texts and images. But what if you need to find the semantics node for something else—for example, a **`Box()`**? The following example, **`BoxButtonDemo()`**, shows a **`Box()`** with a **`Button()`** centered inside. Clicking the button toggles the background color of the box from white to light gray and back:

```
val COLOR1 = Color.White
val COLOR2 = Color.LightGray
@Composable
fun BoxButtonDemo() {
  var color by remember { mutableStateOf(COLOR1) }
  Box(
    modifier = Modifier
      .fillMaxSize()
      .background(color = color),
    contentAlignment = Alignment.Center
  ) {
    Button(onClick = {
      color = if (color == COLOR1)
        COLOR2
      else
        COLOR1
    }) {
      Text(text = stringResource(id =
R.string.toggle))
    }
  }
}
```

Testing **`BoxButtonDemo()`** means finding the box, checking its initial background color, clicking the button, and checking the color again. To be able to find the box, we tag it using the **`testTag()`** modifier, as illustrated in the following code snippet. Applying a tag allows us to find the modified element in tests:

```
val TAG1 = "BoxButtonDemo"
Box(
  modifier = ...
    .testTag(TAG1)
    ...
```

We can check if the box is present, as follows:

```
@Test
```

```
fun testBoxInitialBackgroundColorIsColor1() {
  rule.setContent {
    BoxButtonDemo()
  }
  rule.onNode(hasTestTag(TAG1)).assertExists()
}
```

The **`onNode()`** finder receives a **`hasTestTag()`** **matcher**. Matchers find nodes that meet certain criteria. **`hasTestTag()`** finds a node with the given test tag. There are several predefined matchers. For example, **`isEnabled()`** returns whether the node is enabled, and **`isToggleable()`** returns **`true`** if the node can be toggled.

TIP

Google provides a testing cheat sheet at https://developer.android.com/jetpack/compose/testing-cheatsheet. It nicely groups finders, matchers, actions, and assertions.

To complete the code for the test, we need to check the background color of the box. But how do we do that? Following previous examples, you may expect a **`hasBackgroundColor()`** matcher. Unfortunately, there currently is none. Tests can rely only on what is available through the semantics tree, yet if it does not contain the information we need, we can easily add it. I will show you how in the following section.

Adding custom semantics properties

If you want to expose additional information to tests, you can create custom semantics properties. This requires the following:

- Defining **`SemanticsPropertyKey`**
- Making it available using **`SemanticsPropertyReceiver`**

You can see these in use in the following code snippet:

```
val BackgroundColorKey =
        SemanticsPropertyKey<Color>
("BackgroundColor")
var SemanticsPropertyReceiver.backgroundColor by
BackgroundColorKey
@Composable
fun BoxButtonDemo() {
  ...
  Box(
    modifier = ...
      .semantics { backgroundColor = color }
      .background(color = color),
      ...
```

With **`SemanticsPropertyKey`**, you can set key-value pairs in semantics blocks in a type-safe way. Each key has one statically defined value type—in my example, this is **`Color`**. **`SemanticsPropertyReceiver`** is the scope provided by **`semantics {}`** blocks. It is intended for setting key-value pairs via extension functions.

Here's how to access the custom semantic property in a test case:

```
@Test
fun testBoxInitialBackgroundColorIsColor1() {
  rule.setContent {
    BoxButtonDemo()
  }
  rule.onNode(SemanticsMatcher.expectValue
            (BackgroundColorKey,COLOR1))
     .assertExists()
}
```

`expectValue()` checks whether the value of the given key is equal to the expected value.

Adding custom values to the semantics tree can be of great help when writing tests. However, please carefully consider if you really need to rely on **`SemanticsPropertyKey`**. The semantics tree is also used by the accessibility framework and tools, so it is vital to not pollute the semantics tree with irrelevant information. A solution is to rethink the testing strategy. Instead of testing *if the initial background color of the box is white*, we may just test if the value we pass to the **`background()`** function represents white.

This concludes the sections on testing composable functions. In the following section, we look at debugging Compose apps.

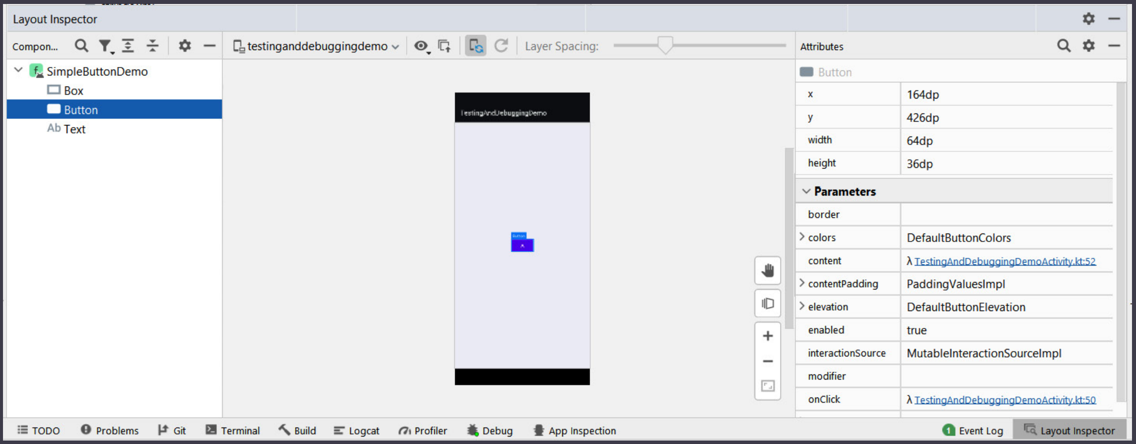
Debugging Compose apps

The title of this section, *Debugging Compose apps*, may indicate major differences to debugging traditional View-based apps. Fortunately, this is not the case. On Android, all composable hierarchies are wrapped inside **`androidx.compose.ui.platform.ComposeView`**. This happens indirectly if you invoke the **`setContent {}`** extension function of **`ComponentActivity`**, or if you deliberately include a composable hierarchy inside a layout (see *Chapter 9*, *Exploring Interoperability APIs*). Either way, in the end, **`ComposeView`** is displayed on screen—for example, inside an Activity or a Fragment. Therefore, all aspects regarding the basic building blocks of an Android app (Activities, Fragments, Services, Broadcast Receiver, Intents, and Content Provider) remain the same.

Of course, any UI framework advocates specific debugging habits. For example, the View system requires watching for **`null`** references. Also, you need to make sure that changes in state reliably trigger updates of the component tree. Fortunately, neither is relevant for Jetpack Compose. As composables are Kotlin functions, you can follow the creation of the composable hierarchy by stepping through the code and examining **`State`** when needed.

To closely examine the visual representation of your composable functions during runtime, you can use the **Layout Inspector** of Android

Studio, as illustrated in the following screenshot. Once you have deployed your app on the Emulator or a real device, open the tool with **Layout Inspector** in the **Tools** menu:



SimpleButtonDemo
Box
Button
Ab Text

Button

x	164dp
y	426dp
width	64dp
height	36dp
Parameters	
border	
colors	DefaultButtonColors
content	λ TestingAndDebuggingDemoActivity.kt:52
contentPadding	PaddingValuesImpl
elevation	DefaultButtonElevation
enabled	true
interactionSource	MutableInteractionSourceImpl
modifier	
onClick	λ TestingAndDebuggingDemoActivity.kt:50

Figure 10.6 – The Layout Inspector in Android Studio

You can select the composable to inspect using the tree on the left-hand side of the Android Studio main window. Important attributes are presented on the right. The center of the tool window contains a configurable, zoomable preview. You can also enable a **three-dimensional (3D)** mode. This allows you to visually inspect the hierarchy by clicking and dragging to rotate the layout.

If you want to log important values of a composable for debugging purposes, you can easily achieve this with modifiers. The following section shows you how to do this.

Using custom modifiers for logging and debugging

As I explained in the *Modifying behavior* section of Chapter 3, *Exploring the Key Principles of Compose*, a modifier is an ordered, immutable collection of modifier elements. Modifiers can change the look and behavior of Compose UI elements. You create custom modifiers by implementing an extension function of **`Modifier`**. The following code snippet uses the **`DrawScope`** interface to print the size of a composable:

```
fun Modifier.simpleDebug() = then(object :
DrawModifier {
  override fun ContentDrawScope.draw() {
    println("width=${size.width},
height=${size.height}")
    drawContent()
  }
})
```

Depending on which interface you choose, you can log different aspects. Using **`LayoutModifier`** you could, for example, access layout-related information.

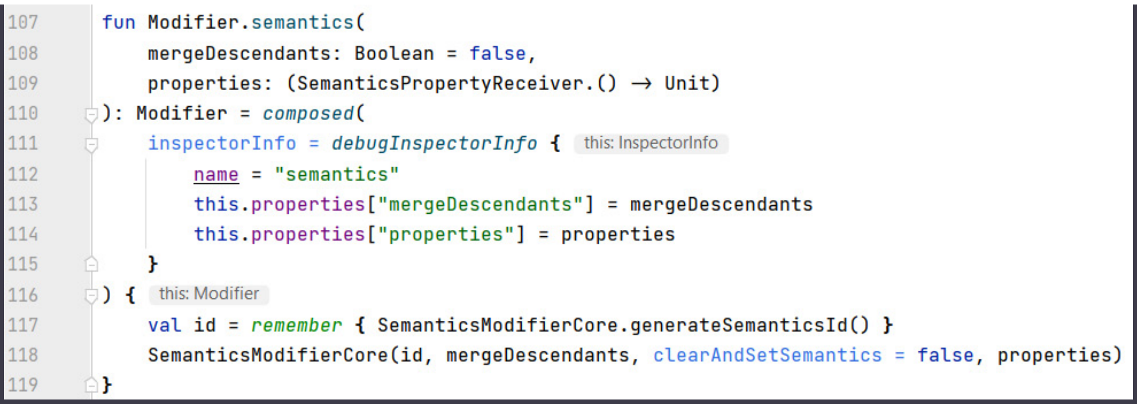
IMPORTANT NOTE

While this may be a clever trick, it is certainly not the primary use case for modifiers. Therefore, if you implement a custom modifier merely for debugging purposes, you should add it to the modifier chain only when debugging.

There is also a built-in feature to provide additional information for debugging purposes. Several modifiers can receive an **`inspectorInfo`** parameter, which is an extension function of **`InspectorInfo`**. This class is a builder for an **`InspectableValue`** interface (this interface defines a value that is inspectable by tools, giving access to private parts of a value). **`InspectorInfo`** has three properties, as follows:

- **`name`** (provides **`nameFallback`** for **`InspectableValue`**)
- **`value`** (provides **`valueOverride`** for **`InspectableValue`**)
- **`properties`** (provides **`inspectableElements`** for **`InspectableValue`**)

To understand how **`inspectorInfo`** is used, let's look in the following screenshot at the implementation of the **`semantics {}`** modifier, which adds semantics key-value pairs for testing and accessibility. Please refer to the *Adding custom semantics properties* section for details:



```
fun Modifier.semantics(
    mergeDescendants: Boolean = false,
    properties: (SemanticsPropertyReceiver.() → Unit)
): Modifier = composed(
    inspectorInfo = debugInspectorInfo {  this: InspectorInfo
        name = "semantics"
        this.properties["mergeDescendants"] = mergeDescendants
        this.properties["properties"] = properties
    }
) {  this: Modifier
    val id = remember { SemanticsModifierCore.generateSemanticsId() }
    SemanticsModifierCore(id, mergeDescendants, clearAndSetSemantics = false, properties)
}
```

Figure 10.7 – Source code of the semantics {} modifier

`semantics {}` invokes the **`composed {}`** modifier, which receives two parameters, **`inspectorInfo`** and **`factory`** (the modifier to be composed). The **`inspectorInfo`** parameter gets the result of the **`debugInspectorInfo {}`** factory method (which receives a **`name`** instance and two elements for **`properties`** as parameters).

`composed {}` adds a **`ComposedModifier`** class to the modifier chain. This private class implements the **`Modifier.Element`** interface and extends **`InspectorValueInfo`**, which in turn implements **`InspectorValueInfo`**. The **`inspectableElements`** property keeps **`Sequence`** of **`ValueElements`**.

To turn on debug inspector information, you must set the **`isDebugInspectorInfoEnabled`** global top-level variable in the **`androidx.compose.ui.platform`** package to **`true`**. Then, you can access and print debug inspector information using reflection. Here's the code you'll need:

```
.semantics { backgroundColor = color }.also {
  (it as CombinedModifier).run {
    val inner =
this.javaClass.getDeclaredField("inner")
    inner.isAccessible = true
    val value = inner.get(this) as InspectorValueInfo
    value.inspectableElements.forEach {
      println(it)
    }
    }
}
```

The modifier returned by **`semantics {}`** is of type **`CombinedModifier`** because **`composed {}`** invokes **`then()`**, which uses **`CombinedModifier`** under the hood. Instead of just printing the raw inspectable element, you can customize the output to your needs.

Summary

In this chapter, we looked at important terms and techniques regarding testing. In the first main section, we set up the infrastructure, wrote and ran a simple unit test locally on the development machine, and then turned to Compose specifics. I introduced you to **`createComposeRule()`** and **`createAndroidComposeRule()`**.

Next, we looked at how composable functions are found in a Compose hierarchy, and why making your app accessible also helps in writing better tests. You also learned about actions and assertions. Finally, we added custom entries to the semantics tree.

The final main section explained how to debug a Compose app. We revisited the semantics tree, and I showed you how to take advantage of **`InspectorInfo`** and **`InspectorValueInfo`** to debug custom modifiers.

Chapter 11, *Conclusion and Next Steps,* concludes this book. We look in the crystal ball to see what future versions of Jetpack Compose may add. For example, we preview Material 3 for Compose, which brings *Material You* design concepts to Compose apps. And we look beyond Android and examine Compose on other platforms.

Further reading

- This book assumes a basic understanding of how to test Android apps. To learn more, please refer to *Test apps on Android* at https://developer.android.com/training/testing.

- *JUnit in Action* by *Catalin Tudose* (*Manning Publications, 2020, ISBN 978-1617297045*) is a thorough introduction to the latest version of the JUnit testing framework.
- If you want to learn more about test automation, you may want to look at *Complete Guide to Test Automation: Techniques, Practices, and Patterns for Building and Maintaining Effective Software Projects* by *Arnon Axelrod* (*Apress, 2018, ISBN 978-1484238318*).
- To get an insight into the test pyramid metaphor, you may want to refer to *The Practical Test Pyramid* by *Ham Vocke*, available at https://martinfowler.com/articles/practical-test-pyramid.html.