Chapter 5: Managing the State of Your Composable Functions

In *Chapter 4*, *Laying Out UI Elements*, I showed you how to set the red, green, and blue portions of a color by dragging sliders. We used **state** to share these values among composable functions. Quite a few other sample apps of the previous chapters dealt with state, too. In fact, reacting to state changes is critical to how modern mobile apps work.

So far, I have described state as data that can change over time. You learned about a few important functions, for example, `remember { }` and `mutableStateOf()`. I also briefly touched on a concept called **state hoisting**.

This chapter builds on these foundations. For example, you will understand the difference between stateless and stateful composables, and when to choose which. Also, I will show you how events should flow in a well-behaving Compose app.

The main sections of this chapter are the following:

- Understanding stateful and stateless composable functions
- Hoisting state and passing events
- Surviving configuration changes

We will start by exploring the differences between stateful and stateless composable functions. You will learn their typical use cases and understand why you should try to keep your composables stateless. Hoisting state is a tool to achieve that; we will cover this important topic in the second main section. Also, I will show you that you can make your composable functions reusable by passing logic as parameters, rather than implementing it inside the composable.

Finally, the *Surviving configuration changes* section will explore the integration of a Compose UI hierarchy in activities, concerning how to retain user input. If the user changes from portrait to landscape mode (or vice versa), activities are destroyed and recreated. Of course, input should not be lost. We look at several ways that a Compose app can achieve this.

Technical requirements

This chapter includes three sample apps. 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 them. **`StateDemo`** contains all examples from the *Understanding stateful and stateless composable functions* section. The *Hoisting state and passing events* section discusses the **`FlowOfEventsDemo`** sample. Finally, **`ViewModelDemo`** belongs to the *Surviving configuration changes* section.

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_05.

Understanding stateful and stateless composable functions

In this section, I will show you the difference between stateful and stateless composable functions. To understand why this is important, let's first focus on the **state** term. In previous chapters, I described state as *data that can change over time*. Where the data is held (an SQLite database, a file, or a value inside an object) does not matter. What is important is that the UI must always show the current data. Therefore, if a value changes, the UI must be notified. To achieve this, we use **observable** types. This is not specific to Jetpack Compose, but a common pattern in many frameworks, programming languages, and platforms. For example, Kotlin supports observables through property delegates:

```
var counter by observable(-1) { _, oldValue, newValue
->
  println("$oldValue -> $newValue")
}
for (i in 0..3) counter = i
```

`observable()` returns a delegate for a property that can be read and written to. In the previous code snippet, the initial value is set to **`-1`**. The property calls a specified function when its value is changed (**`counter = i`**). My example prints the old and new values. In an imperative UI framework, state changes require modifying the component tree. Such code could be put in the callback function. Fortunately, Jetpack Compose doesn't require this, because state changes automatically trigger a recomposition of the relevant UI elements. Let's see how this works.

The **`androidx.compose.runtime.State`** base interface defines a value holder, an object that stores a value of a particular type in a property named **`value`**. If this property is read during the execution of a composable function, the composable will be recomposed whenever **`value`** changes, because internally the current **`RecomposeScope`** interface will be subscribed to changes of that value. Please note that to be able to change the value, state must be an implementation of **`MutableState`**; unlike its immediate predecessor (**`State`**), this interface defines **`value`** using **`var`** instead of **`val`**.

The easiest way to create **`State`** instances is to invoke **`mutableStateOf()`**. This function returns a new **`MutableState`** instance, initialized with the value that was passed in. The next section explains how to use **`mutableStateOf()`** to create a stateful composable function.

Using state in a composable function

A composable function is said to be **stateful** if it maintains (remembers) some value. We achieve this by invoking **`remember {}`**. Let's take a look:

```
@Composable
```

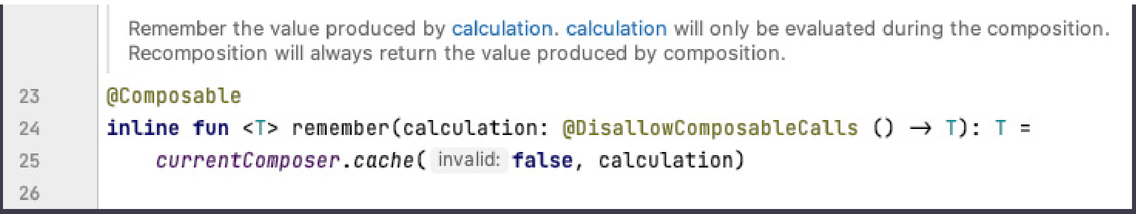
```
@Preview
fun SimpleStateDemo1() {
  val num = remember {
mutableStateOf(Random.nextInt(0,
    10)) }
  Text(text = num.value.toString())
}
```

`SimpleStateDemo1()` creates a mutable state holding a random integer. By invoking **`remember {}`**, we save the state, and in using **`=`**, we assign it to **`num`**. We get the random number through **`num.value`**. Please note that although we defined **`num`** with the **`val`** keyword, we could change the value with **`num.value = …`**, because **`num`** holds the reference to a mutable value holder (whose **`value`** property is writeable). Think of it as modifying an item in a list, not changing to another list. We can slightly alter the code, as shown in the following snippet. Can you spot the difference?

```
@Composable
@Preview
fun SimpleStateDemo2() {
  val num by remember {
mutableStateOf(Random.nextInt(0,
   10)) }
  Text(text = num.toString())
}
```

`SimpleStateDemo2()` creates a mutable state holding a random integer number, too. Using **`by`**, we do not assign the state itself to **`num`** but the value it stores (the random number). This spares us from using **`.value`**, which makes the code a little shorter and hopefully more understandable. However, if we want to change **`num`**, we must change **`val`** to **`var`**. Otherwise, we see a **`Val cannot be reassigned`** error message.

You may be wondering what **`remember {}`** does under the hood. Let's peek into its code and find out:



```
Remember the value produced by calculation. calculation will only be evaluated during the composition.
Recomposition will always return the value produced by composition.
@Composable
inline fun <T> remember(calculation: @DisallowComposableCalls () → T): T =
    currentComposer.cache( invalid: false, calculation)
```

Figure 5.1 – The source code of remember {}

The read-only, top-level **`currentComposer`** property belongs to the **`androidx.compose.runtime`** package. It references an instance of **`Composer`**. This interface is targeted by the Compose Kotlin compiler plugin and used by code generation helpers. You should not call it directly, because the runtime assumes that calls are generated by the compiler and therefore do not contain much validation logic. **`Cache()`** is an extension function of **`Composer`**. It stores a value in the composition data of a composition. So, **`remember {}`** creates internal state. Therefore, composable functions that contain **`remember {}`** are stateful.

`calculation` represents a lambda expression that creates the value to be remembered. It is evaluated only once, during the composition. Subsequent calls to **`remember {}`** (during recompositions) always return this value. The expression is not evaluated again. But what if we need to reevaluate the calculation, that is, remember a new value? After all, isn't state data that can change over time? Here's how you can do this:

```
@Composable
@Preview
fun RememberWithKeyDemo() {
  var key by remember { mutableStateOf(false) }
  val date by remember(key) { mutableStateOf(Date())
}
  Column(horizontalAlignment =
         Alignment.CenterHorizontally) {
    Text(date.toString())
    Button(onClick = { key = !key }) {
      Text(text = stringResource(id =
R.string.click))
```

```
        }
      }
    }
```

The preview of **`RememberWithKeyDemo()`** is shown in *Figure 5.2*:

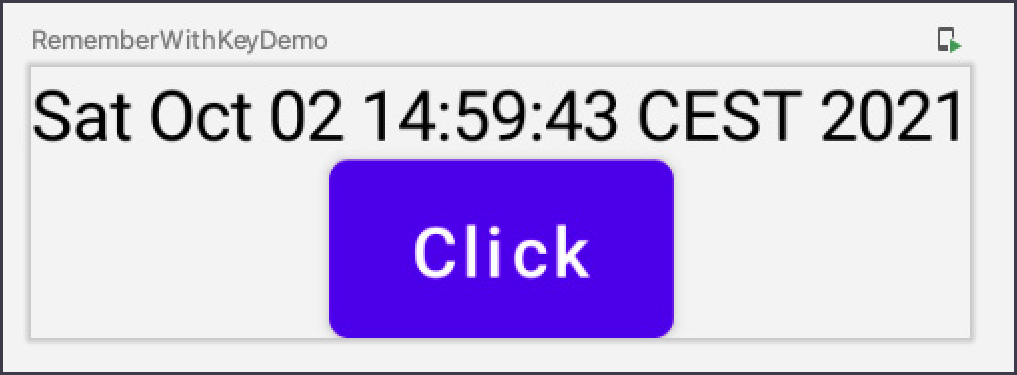


Figure 5.2 – Preview of RememberWithKeyDemo()

`RememberWithKeyDemo()` emits **`Column()`** with two horizontally centered children:

- **`Text()`** shows the string representation of a remembered **`Date`** instance.
- **`Button()`** toggles a Boolean value (**`key`**).

Have you noticed that I pass **`key`** to **`remember { mutableStateOf(Date()) }`**? Here's what happens – when **`remember {}`** is invoked for the first time, the result of the calculation (**`mutableStateOf(Date())`**) is remembered and returned. During recompositions, the calculation is not reevaluated unless **`key`** is *not* equal to the previous composition. In this case, a new value is calculated, remembered, and returned.

TIP

You can pass any number of keys to **`remember {}`**. *If one of them has changed since the previous composition, the calculation is reevaluated, and the new value is remembered and returned.*

Passing keys to **`remember {}`** allows you to change remembered values. Please keep in mind, though, that this makes the composable function less predictable. Therefore, you should consider whether such logic needs to be composable or whether you could pass all state to it.

In the next section, we turn to stateless composables.

Writing stateless composable functions

`remember {}` makes a composable function stateful. A stateless composable, on the other hand, doesn't hold any state. Here's an example:

```
@Composable
@Preview
fun SimpleStatelessComposable1() {
  Text(text = "Hello Compose")
}
```

`SimpleStatelessComposable1()` doesn't receive parameters and it always calls **`Text()`**with the same parameters. Clearly, it doesn't hold any state. But how about the following one?

```
@Composable
fun SimpleStatelessComposable2(text: State<String>) {
  Text(text = text.value)
}
```

While it receives state through the **`text`** parameter, it doesn't store it, and it doesn't remember other state. Consequently, **`SimpleStatelessComposable2()`** is stateless, too. It behaves the same way when called with the same argument multiple times. Such functions are said to be **idempotent**. This makes **`SimpleStatelessComposable2()`** a good blueprint for your own composable functions. They should be as follows:

- **Fast**: Your composable must not do heavy (that is, time-consuming) computations. Never invoke a web service or do any other I/O. Data that is used by a composable should be passed to it.
- **Free of side-effects**: Do not modify global properties or produce unintended observable effects (modifying state that has been passed to a

composable is certainly intentional).

- **Idempotent**: Do not use **`remember {}`**, do not access global properties, and do not call unpredictable code. For example, **`SimpleStateDemo1()`** and **`SimpleStateDemo2()`** use **`Random.nextInt()`**, which, by definition, is (practically) not predictable.

Such composable functions are both easy to reuse and test because they don't rely on anything that isn't passed in as parameters.

When developing reusable composables, you may want to expose both a stateful and a stateless version. Let's see how this looks:

```
@Composable
fun TextFieldDemo(state:
MutableState<TextFieldValue>) {
  TextField(
    value = state.value,
    onValueChange = {
      state.value = it
    },
    placeholder = { Text("Hello") },
    modifier = Modifier.fillMaxWidth()
  )
}
```

This version is stateless because it receives state and does not remember anything. Stateless versions are necessary for callers that need to control the state or hoist it themselves:

```
@Composable
@Preview
fun TextFieldDemo() {
  val state = remember {
mutableStateOf(TextFieldValue("")) }
  TextFieldDemo(state)
}
```

This version is stateful because it remembers the state it creates. Stateful versions are convenient for callers that don't care about the state.

To conclude, try to make your composables stateless by not relying on **`remember {}`** or other functions that remember state (for example, **`rememberLazyListState()`** or **`rememberSaveable()`**). Instead, pass state to the composable. You will see more use cases in the next section.

Hoisting state and passing events

So, state is any value that can change over time. As Jetpack Compose is a declarative UI framework, the only way to update a composable is to call it with new arguments. This happens automatically when state a composable is using changes. State hoisting is a pattern of moving state up to make a composable stateless.

Besides making a composable more easily reusable and testable, moving state up is necessary to use it in more than one composable function. You have already seen this in quite a few of my sample apps. For example, in the *Composing and recomposing the UI* section of *Chapter 3*, *Exploring the Key Principles of Compose*, we used three sliders to create and display a color.

While state controls the visual representation of a composable function (that is, how it looks on screen), **events** notify a part of a program that something has happened. Let's focus a little more on this. My sample **`FlowOfEventsDemo`** app is a simple temperature converter. The user enters a value, specifies whether it represents degrees Celsius or Fahrenheit, and then hits the **Convert** button:

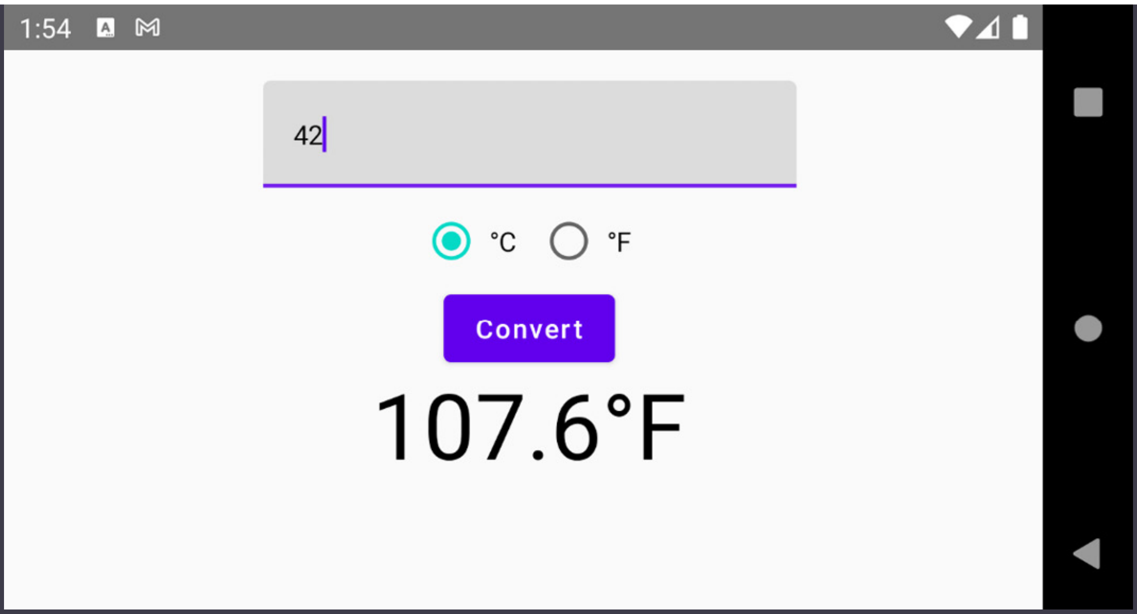


Figure 5.3 – Sample FlowOfEventsDemo app

The UI consists of `Column()` with four children: a text input field, a group of radio buttons with text, a button, and some result text. Let's look at the text input field first:

```
@Composable
fun TemperatureTextField(
  temperature: MutableState<String>,
  modifier: Modifier = Modifier,
  callback: () -> Unit
) {
  TextField(
    value = temperature.value,
    onValueChange = {
      temperature.value = it
    },
    …
    modifier = modifier,
    keyboardActions = KeyboardActions(onAny = {
      callback()
    }),
    keyboardOptions = KeyboardOptions(
      keyboardType = KeyboardType.Number,
      imeAction = ImeAction.Done
    ),
    singleLine = true
  )
}
```

It receives **`MutableState<String>`**, to which it pushes changes to the text in **`onValueChange {}`**. The virtual keyboard is configured to show a *Done* button. If it is invoked, code passed to the composable through **`callback`** is executed. As you will see a little later, that same code will run if the user clicks on the *Convert* button.

In the next section, I'll show you how to create radio buttons and put them in groups so that only one button is selected at a time. The section also covers the button and the result text, which you can see in *Fig. 5.3*.

Creating radio button groups

The app converts between degrees Celsius and Fahrenheit. Therefore, the user must choose the target scale. Such selections can be implemented easily in Jetpack Compose using **`androidx.compose.material.RadioButton()`**. This composable doesn't show some descriptive text, but it is easy to add some. Here's how:

```
  @Composable
  fun TemperatureRadioButton(
    selected: Boolean,
    resId: Int,
    onClick: (Int) -> Unit,
    modifier: Modifier = Modifier
  ) {
    Row(
      verticalAlignment = Alignment.CenterVertically,
      modifier = modifier
    ) {
      RadioButton(
        selected = selected,
        onClick = {
          onClick(resId)
        }
      )
      Text(
        text = stringResource(resId),
        modifier = Modifier
          .padding(start = 8.dp)
      )
    }
  }
```

RadioButton() and **Text()** are simply added to **Row()** and vertically centered. **TemperatureRadioButton()** receives a lambda expression with the **onClick** parameter. That code is executed when the radio button is clicked. My implementation passes the **resId** parameter to the lambda expression, which will be used to determine the button in a group. Here's how:

```
@Composable
fun TemperatureScaleButtonGroup(
  selected: MutableState<Int>,
  modifier: Modifier = Modifier
) {
  val sel = selected.value
  val onClick = { resId: Int -> selected.value =
resId }
  Row(modifier = modifier) {
    TemperatureRadioButton(
      selected = sel == R.string.celsius,
      resId = R.string.celsius,
      onClick = onClick
    )
    TemperatureRadioButton(
      selected = sel == R.string.fahrenheit,
      resId = R.string.fahrenheit,
      onClick = onClick,
      modifier = Modifier.padding(start = 16.dp)
    )
  }
}
```

Two **TemperatureRadioButton()** are put in a **Row()**. The first one is configured to represent degrees Celsius, the second one degrees Fahrenheit. Both receive the same **onClick** lambda. It sets the **resId** parameter it received from **TemperatureRadioButton()** as the new value of the **selected** parameter, a mutable state. So, what is happening here? Clicks on a radio button are not handled inside **TemperatureRadioButton()** but passed to the parent, **TemperatureScaleButtonGroup()**. The event, a button click, is said to **bubble up**. This way, the parent can orchestrate its children and notify its parent. In my example, this means changing some state.

Next, let's see what happens when the user clicks the **Convert** button. This happens inside **FlowOfEventsDemo()**. Here's the overall structure of this composable function:

```
@Composable
@Preview
fun FlowOfEventsDemo() {
  ...
  val calc = {
    val temp = temperature.value.toFloat()
    convertedTemperature = if (scale.value ==
                               R.string.celsius)
      (temp * 1.8F) + 32F
    else
      (temp - 32F) / 1.8F
  }
  val result = remember(convertedTemperature) {
    if (convertedTemperature.isNaN())
      ""
    else
      "${convertedTemperature}${
        if (scale.value == R.string.celsius)
          strFahrenheit
        else strCelsius
      }"
  }
  val enabled = temperature.value.isNotBlank()
  Column( ... ) {
    TemperatureTextField(
      temperature = temperature,
      modifier = Modifier.padding(bottom = 16.dp),
      callback = calc
    )
    TemperatureScaleButtonGroup(
      selected = scale,
      modifier = Modifier.padding(bottom = 16.dp)
    )
    Button(
      onClick = calc,
      enabled = enabled
    ) {
      Text( ... )
    }
    if (result.isNotEmpty()) {
      Text(text = result, …
      )
    }
  }
}
```

The conversion logic is assigned to a read-only variable called **`calc`**. It is passed to **`TemperatureTextField()`** and **`Button()`**. Passing the code that is going to be executed in response to an event to a composable function rather than hard coding it inside makes the composable more easily reusable and testable.

The text that is displayed after conversion is remembered and assigned to **`result`**. It is re-evaluated when **`convertedTemperature`** changes. This happens inside the **`calc`** lambda expression. Please note that I need to pass a key to **`remember {}`**; otherwise, the result would be changed also if the user picks another scale.

In the next section, we will look at how state can be persisted. To be more precise, we turn to configuration changes. If the user rotates a device, the UI should not be reset. Unfortunately, this is what happens with all sample apps I have shown you so far. It's time to fix this.

Surviving configuration changes

Please recall that our definition of state as data that may change over time is quite broad. For example, we do not specify where the data is stored. If it resides in a database, a file, or some backend in the cloud, the app should include a dedicated persistence layer. However, until Google introduced the Android Architecture Components back in 2017, there had been practically no guidance for developers on how to structure their apps. Consequently, persistence code, UI logic, and domain logic were often crammed into one activity. Such code was difficult to maintain and often prone to errors. To make matters a little more complicated, there are situations when an activity is destroyed and recreated shortly after. For example, this happens when a user rotates a device. Certainly, data should then be remembered.

The **`Activity`** class has a few methods to handle this. For example, **`onSaveInstanceState()`** is invoked when the activity is (temporarily) destroyed. Its counterpart **`onRestoreInstanceState()`** method is called only when such an instance state has been saved before. Both methods receive an instance of **`Bundle`**, which has getters and setters for various data types. However, the concept of instance state has been designed for the traditional view system. Most activities held references to UI elements and therefore could be accessed easily inside **`onSaveInstanceState()`** and **`onRestoreInstanceState()`**.

Composables, on the other hand, are usually implemented as top-level functions. So, how can their state be set or queried from inside an activity? To temporarily save state in a Compose app, you can use **`rememberSaveable {}`**. This composable function remembers the value produced by a factory function. It behaves similarly to **`remember {}`**. The stored value will survive the activity or process recreation. Internally, the **`savedInstanceState`** mechanism is used. The sample **`ViewModelDemo`** app shows how to use **`rememberSaveable {}`**. Here's what the main activity looks like:

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

We don't need to override **`onSaveInstanceState()`** to temporarily save our state used with composables:

```
@Composable
@Preview
fun ViewModelDemo() {
  ...
  val state1 = remember {
    mutableStateOf("Hello #1")
  }
  val state2 = rememberSaveable {
    mutableStateOf("Hello #2")
  }
  ...
```

```
    state3.value?.let {
      Column(modifier = Modifier.fillMaxWidth()) {
        MyTextField(state1) { state1.value = it }
        MyTextField(state2) { state2.value = it }
        ...
      }
    }
  }
```

The app shows three text input fields that receive their values from states assigned to **`state1`**, **`state2`**, and **`state3`**. For now, we will focus on the first two. **`state3`** will be the subject of the *Using ViewModel* section. **`state1`** invokes **`remember {}`**, whereas **`state2`** uses **`rememberSaveable {}`**. If you ran **`ViewModelDemo`**, changed the content of the text input fields, and rotated the device, the first one would be reset to the original text, whereas the second one would keep your changes.

`MyTextField` is a very simple composable. It looks like this:

```
@Composable
fun MyTextField(
  value: State<String?>,
  onValueChange: (String) -> Unit
) {
  value.value?.let {
    TextField(
      value = it,
      onValueChange = onValueChange,
      modifier = Modifier.fillMaxWidth()
    )
  }
}
```

Have you noticed that **`value`** is of **`State<String?>`**? Why would I need a value holder whose value can be **`null`**, and therefore need to check with **`value.value?.let {}`** that it isn't? We will be reusing the composable in the following section, and you will find the answer to this question there.

Please note, though, that for both **state1** and **state2**, this would not have been necessary.

Using ViewModel

While temporarily storing state with **rememberSaveable {}** works great, an app still must get data that is persisted for a longer time (for example, in a database or file) and make it available as state that can be used in composables. The Android Architecture Components include **ViewModel** and **LiveData**. Both can be used seamlessly with Jetpack Compose.

First, you need to add a few implementation dependencies to the module-level **build.gradle** file:

```
implementation "androidx.compose.runtime:runtime-
  livedata:$compose_version"
implementation 'androidx.lifecycle:lifecycle-runtime-
  ktx:2.4.0'
implementation 'androidx.lifecycle:lifecycle-
viewmodel-
  compose:2.4.0'
```

The next step is to define a **ViewModel** class. It extends **androidx.lifecycle.ViewModel**. A **ViewModel** class stores and manages UI-related data in a lifecycle-conscious way. This means that data will survive configuration changes, such as screen rotations. **MyViewModel** exposes one property called **text** and a method named **setText()** to set it:

```
class MyViewModel : ViewModel() {
    private val _text: MutableLiveData<String> =
        MutableLiveData<String>("Hello #3")
    val text: LiveData<String>
        get() = _text
    fun setText(value: String) {
        _text.value = value
    }
```

```
  }
```

My example shows a **ViewModel** class using **LiveData**. Depending on the architecture of an app, you can utilize other mechanisms for working with observable data. Going into more detail is, however, beyond the scope of this book. You can find additional information in *Guide to app architecture* at https://developer.android.com/jetpack/guide.

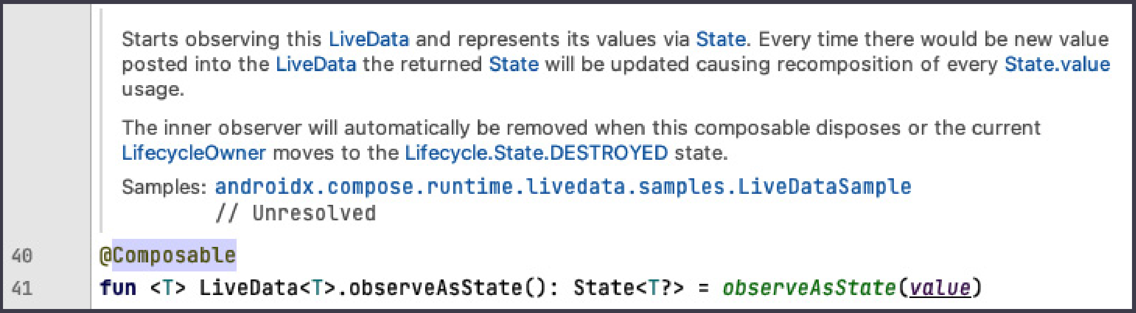
To access the **ViewModel** class from inside a composable function, we invoke the composable **viewModel()**. It belongs to the **androidx.lifecycle.viewmodel.compose** package:

```
  val viewModel: MyViewModel = viewModel()
```

LiveData is made available as state like this:

```
  val state3 = viewModel.text.observeAsState()
```

Let's take a quick look at its source code:



Starts observing this LiveData and represents its values via State. Every time there would be new value posted into the LiveData the returned State will be updated causing recomposition of every State.value usage.

The inner observer will automatically be removed when this composable disposes or the current LifecycleOwner moves to the Lifecycle.State.DESTROYED state.

Samples: androidx.compose.runtime.livedata.samples.LiveDataSample
// Unresolved

```
@Composable
fun <T> LiveData<T>.observeAsState(): State<T?> = observeAsState(value)
```

Figure 5.4 – Source code of the observeAsState() extension function

observeAsState() is an extension function of **LiveData**. It passes the **value** property of its **LiveData** instance to a variant of **observeAsState()** that takes parameters. Have you noticed that the return type is **State<T?>**? That is why I defined **MyTextField()** in the previous section to receive **State<String?>**. To be able to use **State<String>** as with **remember {}** and **rememberSaveable {}**, we would need to define **state3** like this:

```
  val state3 =
     viewModel.text.observeAsState(viewModel.text.value
  ) as
     State<String>
```

In my opinion, this is less favorable than using **`State<String?>`** because we use an unchecked cast.

To reflect changes in state in the **`ViewModel`** class, we need code like this:

```
MyTextField(state3) {
  viewModel.setText(it)
}
```

Unlike using **`MutableState`**, we must explicitly invoke the **`setText()`** method of **`MyViewModel`** and pass the changed text.

To conclude, **`rememberSaveable {}`** is simple and easy to use. For more complex scenarios than presented in this chapter, you can provide **`androidx.compose.runtime.saveable.Saver`** implementations, which make your data objects simpler and convert them to something saveable. Bigger apps should use **`ViewModel`** classes, as recommended for quite a while now by Google. The combination of **`ViewModel`** and **`LiveData`** classes can be integrated nicely into composable apps using **`observerAsState()`**.

Summary

This chapter aimed to give a more detailed look at state in Compose apps. We started by exploring the differences between stateful and stateless composable functions. You learned their typical use cases and why you should try to keep your composables stateless. Hoisting state is a tool to achieve that. We covered this important topic in the second main section. I also showed you that you can make your composable functions more re-usable by passing logic as parameters, rather than implementing it inside the composable. The previous section explored the integration of a Compose UI hierarchy in activities concerning how to retain user input. We looked at the differences between **`remember {}`** and **`rememberSaveable {}`**, and I gave you a glimpse of how bigger Compose apps can benefit from **`ViewModel`** classes.

Chapters 1 to 5 introduced you to various aspects of Jetpack Compose, such as composable functions, state, and layout. Chapter 6, *Putting Pieces Together*, focuses on one app, providing you with a bigger picture of how these pieces work together to form a real-world app. We will implement a simple unit converter app, focusing on app architecture and UI, including theming and navigation.

Support | Sign Out

©2022 O'REILLY MEDIA, INC. TERMS OF SERVICE | PRIVACY POLICY