# Chapter 7: Tips, Tricks, and Best Practices

In *Chapter 6*, *Putting Pieces Together*, we combined several key techniques of Jetpack Compose such as state hoisting, app theming, and navigation in a real-world example. **`ComposeUnitConverter`** stores state in a **`ViewModel`** and eventually persists it using the *Repository* pattern. In this chapter, I show you how to pass objects to a **`ViewModel`** upon instantiation and use these objects to load and save data. In *Chapter 3*, *Exploring the Key Principles of Compose*, we examined features of well-behaved composable functions. Composables should be free of side effects to make them reusable and easy to test. However, there are situations when you need to either react to or initiate state changes that happen outside the scope of a composable function. We will cover this at the end of this chapter.

These are the main sections of this chapter:

- Persisting and retrieving state
- Keeping your composables responsive
- Understanding side effects

We start by continuing the exploration of the **`ViewModel`** pattern we began in the *Using a ViewModel* section of *Chapter 5*, *Managing the State of Your Composable Functions*. This time, we will add business logic to the **`ViewModel`** and inject an object that can persist and retrieve data.

The *Keeping your composables responsive* section revisits one of the key requirements of a composable function. As recomposition can occur very often, composables must be as fast as possible. This greatly influences what the code may and may not do. Long-running tasks—for example, complex computations or network calls—should not be invoked synchronously.

The *Understanding side effects* section covers situations when you need to either react to or initiate state changes that happen outside the scope of a composable function. For example, we will be using **`LaunchedEffect`** to start and stop complex computations.

# Technical requirements

The *Persisting and retrieving state* and *Keeping your composables responsive* sections further discuss the sample **`ComposeUnitConverter`** app. The *Understanding side effects* section is based on the **`EffectDemo`** sample. Please refer to the *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_07.

# Persisting and retrieving state

State is app data that may change over time. In a Compose app, state is typically represented as instances of **`State`** or **`MutableState`**. If such objects are used inside composable functions, a recomposition is triggered upon state changes. If a state is passed to several composables, all of them may be recomposed. This leads to the *state hoisting* principle: state is passed to composable functions rather than being remembered inside them. Often, such state is remembered in the composable that is the parent of the ones using the state. An alternative approach is to implement an architectural pattern called **`ViewModel`**. It is used in many **user interface** (**UI**) frameworks on various platforms. On Android, it has been available since 2017 as part of the **Android Architecture Components**.

The general idea of a **`ViewModel`** is to combine data and access logic that is specific to a certain part of an app. Depending on the platform, this may be a screen, a window, a dialog, or another similar top-level container. On Android, it's usually an activity. The data is observable, so UI elements can register and get notified upon changes. How the observable pattern is implemented depends on the platform. The Android Architecture Components introduced **`LiveData`** and **`MutableLiveData`**. In the *Surviving configuration changes* section of *Chapter 5*, *Managing the State of Your Composable Functions*, I showed you how to use them inside a **`ViewModel`** to store data that survives device rotations and how to connect **`LiveData`** instances to composable functions.

Here's a brief recap: to connect **`LiveData`** objects to the Compose world, we first obtain a **`ViewModel`** instance using **`androidx.lifecycle.viewmodel.compose.viewModel()`**, and then invoke the **`observeAsState()`** extension function on a property of the **`ViewModel`**. The returned state is read-only, so if a composable wants to update the property, it must call a setter that needs to be provided by the **`ViewModel`**.

So far, I have not explained how to persist state and restore it later. To put it another way: where do **`ViewModel`** instances get the initial values for their data, and what do they do upon changes? Let's find out in the next section.

## Injecting objects into a ViewModel

If a **`ViewModel`** wants to load and save data, it may need to access a database, the local filesystem, or some remote web service. Yet, it should be irrelevant for the **`ViewModel`** how reading and writing data works behind the scenes. The Android Architecture Components suggest implementing the *Repository* pattern. A repository abstracts the mechanics of loading and saving data and makes it available through a collection-like interface. You can find out more about the Repository pattern at https://martinfowler.com/eaaCatalog/repository.html.

You will see shortly what the implementation of a simple repository may look like, but first, I need to show you how to pass objects to a `ViewModel` upon instantiation. `viewModel()` receives a `factory` parameter of type `ViewModelProvider.Factory`. It is used to create `ViewModel` instances. If you pass `null` (the default value), a built-in default factory is used. `ComposeUnitConverter` has two screens, so its factory must be able to create `ViewModel` instances for each screen.

Here's what `ViewModelFactory` looks like:

```
class ViewModelFactory(private val repository:
Repository)
:ViewModelProvider.NewInstanceFactory() {
  override fun <T : ViewModel?> create(modelClass:
Class<T>): T =
    if (modelClass.isAssignableFrom
     (TemperatureViewModel::class.java))
      TemperatureViewModel(repository) as T
    else
      DistancesViewModel(repository) as T
}
```

`ViewModelFactory` extends the `ViewModelProvider.NewInstanceFactory` static class and overrides the `create()` method (which belongs to the parent `Factory` interface). The `modelClass` represents the `ViewModel` to be created. Therefore, if the following code is `true`, then we instantiate `TemperatureViewModel` and pass `repository`:

```
modelClass.isAssignableFrom
(TemperatureViewModel::class.java)
```

This parameter was passed to the constructor of `ViewModelFactory`. Otherwise, a `DistancesViewModel` instance is created. Its constructor also receives `repository`. If your factory needs to differentiate between more `ViewModel` instances, you will probably use a `when` instead.

Next, let's look at my **Repository** class to find out how **ComposeUnitConverter** loads and saves data. You can see this in the following code snippet:

```
class Repository(context: Context) {
    private val prefs =
        PreferenceManager.getDefaultSharedPreferences
(context)
    fun getInt(key: String, default: Int) =
        prefs.getInt(key, default)
    fun putInt(key: String, value: Int) {
        prefs.edit().putInt(key, value).apply()
    }
    fun getString(key: String,
        default: String) = prefs.getString(key,
default)
    fun putString(key: String, value: String) {
        prefs.edit().putString(key, value).apply()
    }
}
```

**Repository** uses Jetpack Preference. This library is a replacement for the platform classes and interfaces inside the **android.preference** package, which was deprecated with **application programming interface** (**API**) level 29.

*IMPORTANT NOTE*

*Both the platform classes and the library are designed for user settings. You should not use them to access more complex data, larger texts, or images. Record-like data is best kept in an SQLite database, whereas files are ideal for large texts or images.*

To use Jetpack Preference, we need to add an implementation dependency to **androidx.preference:preference-ktx** in the module-level **build.gradle** file. **getDefaultSharedPreferences()** requires an instance of

**android.content.Context**, which is passed to the constructor of **Repository**.

Before we move on, let's recap what I showed you so far, as follows:

- **TemperatureViewModel** and **DistancesViewModel** receive a **Repository** instance in their constructor.
- **Repository** receives a **Context** object.
- **ViewModel** instances are decoupled from activities. They survive configuration changes.

The last bullet point has an important consequence regarding the context we can pass to the repository. Let's find out more in the next section.

## Using the factory

Here's how both the repository and factory are created:

```
class ComposeUnitConverterActivity :
ComponentActivity() {
  override fun onCreate(savedInstanceState: Bundle?)
{
    super.onCreate(savedInstanceState)
    val factory =
      ViewModelFactory(Repository(applicationContext)
)
    setContent {
      ComposeUnitConverter(factory)
    }
  }
}
```

Both **Repository** and **ViewModelFactory** are ordinary objects, so they are simply instantiated, passing the required parameters to them.

*IMPORTANT NOTE*

*It may be tempting to pass* **`this`** *(the calling activity) as the context. However, as* **`ViewModel`** *instances survive configuration changes (that is, the recreation of an activity), the context may change. If it does, the repository would be accessing a no longer available activity. By using* **`applicationContext`**, *we make sure that this issue does not occur.*

**`ComposeUnitConverter()`** is the root of the composable hierarchy. It passes the factory to **`ComposeUnitConverterNavHost()`**, which in turn uses it inside **`composable {}`** as a parameter for the screens, as illustrated in the following code snippet:

```
composable(ComposeUnitConverterScreen.route_temperatu
re) {
  TemperatureConverter(
    viewModel = viewModel(factory = factory)
  )
}
```

In this section, I showed you how to inject a repository object into a **`ViewModel`** using simple constructor invocation. If your app relies on a **dependency injection** (**DI**) framework, you will need to use its mechanisms (for example, an annotation) instead. However, this is beyond the scope of this book. Next, we will look at how the **`ViewModel`** uses the repository.

# Keeping your composables responsive

When implementing composable functions, you should always keep in mind that their main purpose is to declare the UI and to handle user interactions. Ideally, anything needed to achieve this is passed to the composable, including state and logic (such as click handlers), making it stateless. If state is needed only inside a composable, the function may keep state temporarily using **`remember {}`**. Such composables are called **stateful**. If data is kept in a **`ViewModel`**, composables must interact with it. So, the **`ViewModel`** code must be fast, too.

## Communicating with ViewModel instances

Data inside a **ViewModel** should be observable. **ComposeUnitConverter** uses **LiveData** and **MutableLiveData** from the Android Architecture Components to achieve this. You can choose other implementations of the *Observer* pattern, provided there is a way to obtain **State** or **MutableState** instances that are updated upon changes in the **ViewModel**. This is beyond the scope of this book. **TemperatureViewModel** is the **ViewModel** for the **TemperatureConverter()** composable function.

Let's look at its implementation. In the following code snippet, I omitted code related to the **scale** property for brevity. You can find the full implementation in the GitHub repository:

```
class TemperatureViewModel(private val repository:
Repository): ViewModel() {
  ...
  private val _temperature: MutableLiveData<String>
             = MutableLiveData(
                 repository.getString("temperature",
"")
  )
  val temperature: LiveData<String>
    get() = _temperature
  fun getTemperatureAsFloat(): Float
          = (_temperature.value ?: "").let {
    return try {
      it.toFloat()
    } catch (e: NumberFormatException) {
      Float.NaN
    }
  }
  fun setTemperature(value: String) {
    _temperature.value = value
    repository.putString("temperature", value)
```

```
  }
  fun convert() = getTemperatureAsFloat().let {
    if (!it.isNaN())
      if (_scale.value == R.string.celsius)
        (it * 1.8F) + 32F
      else
        (it - 32F) / 1.8F
    else
      Float.NaN
  }
}
```

**`ViewModel`** instances present their data through pairs of variables, as follows:

- A public read-only property (**`temperature`**)
- A private writeable backing variable (**`_temperature`**)

Properties are not changed by assigning a new value but by invoking some setter functions (**`setTemperature()`**). You can find an explanation of why this is the case in the *Using a ViewModel* section of *Chapter 5*, *Managing the State of Your Composable Functions*. There may be additional functions that can be invoked by the composable—for example, logic to convert a temperature from °C to °F (**`convert()`**) should not be part of the composable code. The same applies to format conversions (from **`String`** to **`Float`**). These are best kept in the **`ViewModel`**.

Here's how the **`ViewModel`** is used from a composable function:

```
@Composable
fun TemperatureConverter(viewModel:
TemperatureViewModel) {
  …
  val currentValue =
viewModel.temperature.observeAsState(
```

```
                        viewModel.temperature.value
?: "")
  val scale = viewModel.scale.observeAsState(
                 viewModel.scale.value ?:
R.string.celsius)
  var result by remember { mutableStateOf("") }
  val calc = {
    val temp = viewModel.convert()
    result = if (temp.isNaN())
      ""
    else
      "$temp${
        if (scale.value == R.string.celsius)
          strFahrenheit
        else strCelsius
      }"
  }
  …
  Column(
    …
  ) {
    TemperatureTextField(
      temperature = currentValue,
      modifier = Modifier.padding(bottom = 16.dp),
      callback = calc,
      viewModel = viewModel
    )
    …
    Button(
      onClick = calc,
      …
    if (result.isNotEmpty()) {
      Text(
        text = result,
        style = MaterialTheme.typography.h3
```

```
      )
    }
    …
```

Have you noticed that **`TemperatureConverter()`** receives its **`ViewModel`** as a parameter?

*TIP*

*You should provide a default value (`viewModel()`) for preview and testability, if possible. However, this doesn't work if the `ViewModel` requires a repository (as in my example) or other constructor values.*

**`State`** instances are obtained by invoking **`observeAsState()`** of **`ViewModel`** properties (**`temperature`** and **`scale`**), which are **`LiveData`** instances. The code assigned to **`calc`** is executed when either the **Convert** button or **Done** button of the virtual keyboard is pressed. It creates a string representing the converted temperature, including scale, and assigns it to **`result`**, a state being used in a **`Text()`** composable. Please note that the **`calc`** lambda expression calls the **`convert()`** function of **`ViewModel`** function to get the converted temperature. You should always try to remove business logic from composables and instead put it inside the **`ViewModel`**.

So far, I showed you how to observe changes in the **`ViewModel`** and how to invoke logic inside it. There is one piece left: changing a property. In the preceding code snippet, **`TemperatureTextField()`** receives the **`ViewModel`**. Let's see what it does with it here:

```
@Composable
fun TemperatureTextField(
  temperature: State<String>,
  modifier: Modifier = Modifier,
  callback: () -> Unit,
  viewModel: TemperatureViewModel
) {
  TextField(
```

```
    value = temperature.value,
    onValueChange = {
      viewModel.setTemperature(it)
    },
    …
```

Whenever the text changes, **setTemperature()** is invoked with the new value. Please recall that the setter does the following:

```
_temperature.value = value
```

The **ViewModel** updates the value of the **_temperature** (**MutableLiveData**) backing variable. As the **temperature** public property references **_temperature**, its observers (in my example, the state returned by **observeAsState()** in **TemperatureConverter()**) are notified. This triggers a recomposition.

In this section, we focused on how communication flows between composable functions and **ViewModel** instances. Next, we examine what can go wrong if the **ViewModel** breaks the contract with the composable and what you can do to prevent this.

## Handling long-running tasks

Composable functions actively interact with a **ViewModel** by setting new values for properties (**setTemperature()**) and by invoking functions that implement business logic (**convert()**). As recompositions can occur frequently, these functions may be called very often. Consequently, they must return very fast. This surely is the case for simple arithmetic, such as converting between °C and °F.

On the other hand, some algorithms may become increasingly time-consuming for certain inputs. Here's an example. Fibonacci numbers can be computed recursively and iteratively. While a recursive algorithm is simpler to implement, it takes much longer for large numbers. If a synchronous function call does not return in a timely fashion, it may affect how the user perceives your app. You can test this by adding **while (true) ;** as the first line of code inside **convert()**. If you then run **ComposeUnitConverter**, enter some number, and press **Convert**, the app will no longer respond.

*IMPORTANT NOTE*

*Potentially long-running tasks must be implemented asynchronously.*

To avoid situations where the app is not responding because a computation takes too much time, you must decouple the computation from delivering the result. This is done with just a few steps, as follows:

1. Provide the result as an observable property.
2. Compute the result using a coroutine or a Kotlin flow.
3. Once the computation is finished, update the **result** property.

Here's a sample implementation taken from **DistancesViewModel**:

```
private val _convertedDistance:
MutableLiveData<Float>
                = MutableLiveData(Float.NaN)
val convertedDistance: LiveData<Float>
  get() = _convertedDistance
fun convert() {
  getDistanceAsFloat().let {
    viewModelScope.launch {
      _convertedDistance.value = if (!it.isNaN())
        if (_unit.value == R.string.meter)
          it * 0.00062137F
        else
          it / 0.00062137F
      else
        Float.NaN
    }
  }
```

```
}
```

`viewModelScope` is available via an implementation dependency to `androidx.lifecycle:lifecycle-viewmodel-ktx` in the module-level `build.gradle` file. `convert()` spawns a coroutine, which will update the value of `_convertedDistance` once the computation is finished. Composable functions can observe changes by invoking `observeAsState()` on the `convertedDistance` public property. But how do you access `convertedDistance` and `convert()`? Here's a code snippet from `DistancesConverter.kt`:

```
val convertedValue by
        viewModel.convertedDistance.observeAsState()
val result by remember(convertedValue) {
  mutableStateOf(
    if (convertedValue?.isNaN() != false)
      ""
    else
      "$convertedValue ${
        if (unit.value == R.string.meter)
          strMile
        else strMeter
      }"
  )
}
val calc = {
  viewModel.convert()
}
```

`result` receives the text to be output once a distance has been converted, so it should update itself whenever `convertedValue` changes. Therefore, I pass `convertedValue` as a key to `remember {}`. Whenever the key changes, the `mutableStateOf()` lambda expression is recomputed, so `result` gets updated. `calc` is invoked when the **Convert** button or the **Done** button on the virtual keyboard is pressed. It spawns an asynchronous operation, which eventually will update `convertedValue`.

In this section, I have often used the term *computation*. Computation does not only mean arithmetic. Accessing databases, files, or web services may also consume considerable resources and be time-consuming. Such operations must be executed asynchronously. Please keep in mind that long-running tasks may not be part of the **`ViewModel`** itself but be invoked from it (for example, a repository). Consequently, such code must be fast too. My **`Repository`** implementation accesses the **`Preferences`** API synchronously for simplicity. Strictly speaking, even such basic operations should be asynchronous.

*TIP*

*Jetpack DataStore allows you to store key-value pairs or typed objects with protocol buffers. It uses Kotlin coroutines and Flow to store data asynchronously. You can find more information about Jetpack DataStore at [https://developer.android.com/topic/libraries/architecture/datastore](https://developer.android.com/topic/libraries/architecture/datastore).*

This concludes our look at the communication between composable functions and **`ViewModel`** instances. In the next section, I will introduce you to composables that do not emit UI elements but cause side effects to run when a composition completes.

# Understanding side effects

In the *Using Scaffold() to structure your screen* section of *Chapter 6*, *Putting Pieces Together*, I showed you how to display a snack bar using **`rememberCoroutineScope {}`** and **`scaffoldState.snackbarHostState.showSnackbar()`**. As **`showSnackbar()`** is a suspending function, it must be called from a coroutine or another suspending function. Therefore, we created and remembered **`CoroutineScope`** using **`rememberCoroutineScope()`** and invoked its **`launch {}`** function.

## Invoking suspending functions

The **`LaunchedEffect()`** composable is an alternative approach for spawning a suspending function. To see how it works, let's look at the **`LaunchedEffectDemo()`** composable. It belongs to the **`EffectDemo`** sample, as illustrated in the following screenshot:

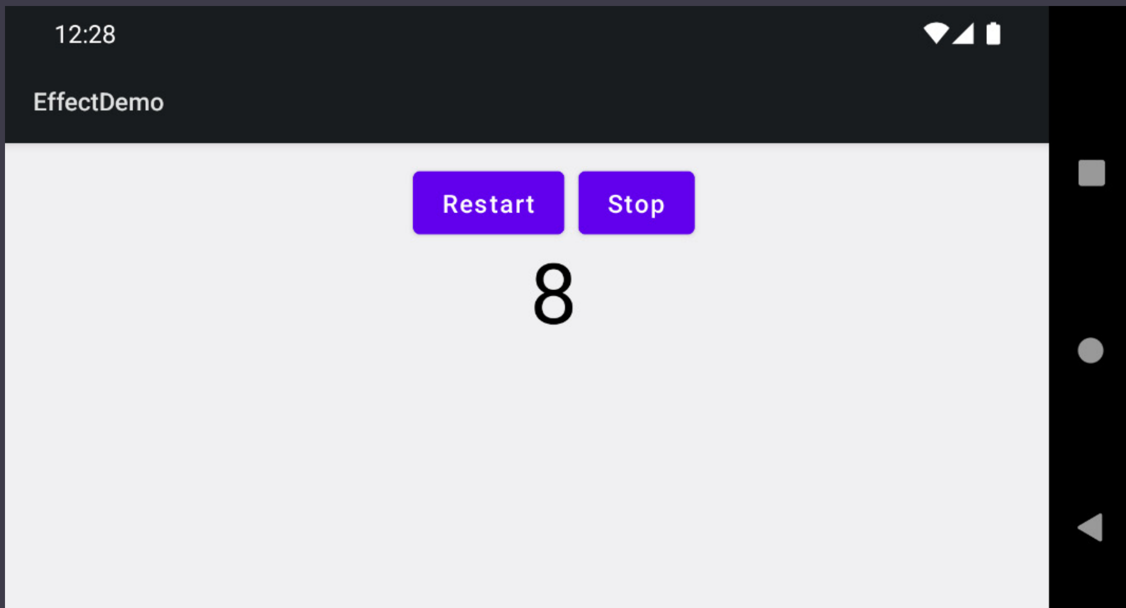


Figure 7.1 – The EffectDemo sample showing LaunchedEffectDemo()

**`LaunchedEffectDemo()`** implements a counter. Once the **Start** button has been clicked, a counter is incremented every second. Clicking on **Restart** resets the counter. **Stop** terminates it. The code to achieve this is illustrated in the following snippet:

```
@Composable
fun LaunchedEffectDemo() {
   var clickCount by rememberSaveable {
mutableStateOf(0) }
   var counter by rememberSaveable {
mutableStateOf(0) }
    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp),
        horizontalAlignment =
Alignment.CenterHorizontally
```

```
    ) {
        Row {
            Button(onClick = {
                clickCount += 1
            }) {
                Text(
                    text = if (clickCount == 0)
                      stringResource(id =
R.string.start)
                    else
                      stringResource(id =
R.string.restart)
                )
            }
            Spacer(modifier = Modifier.width(8.dp))
            Button(enabled = clickCount > 0,
                onClick = {
                    clickCount = 0
                }) {
                Text(text = stringResource(id =
                              R.string.stop))
            }
            if (clickCount > 0) {
                LaunchedEffect(clickCount) {
                    counter = 0
                    while (isActive) {
                        counter += 1
                        delay(1000)
                    }
                }
            }
        }
        Text(
            text = "$counter",
            style = MaterialTheme.typography.h3
```

```
        )
    }
}
```

**clickCount** counts how often **Start** or **Restart** has been clicked. **Stop** resets it to **0**. A value greater than **0** indicates that another remembered variable (**counter**) should be increased every second. This is done by a suspending function that is passed to **LaunchedEffect()**. This composable is used to safely call suspend functions from inside a composable. Let's see how it works.

When **LaunchedEffect()** enters the composition (**if (clickCount > 0) …**), it launches a coroutine with the block of code passed as a parameter. The coroutine will be cancelled if **LaunchedEffect()** leaves the composition (**clickCount <= 0**). Have you noticed that it receives one parameter? If **LaunchedEffect()** is recomposed with different keys (my example uses just one, but you can pass more if needed), the existing coroutine will be canceled and a new one is started.

As you have seen, **LaunchedEffect()** makes it easy to start and restart asynchronous tasks. The corresponding coroutines are cleaned up automatically. But what if you need to do some additional housekeeping (such as unregistering listeners) when keys change or when the composable leaves the composition? Let's find out in the next section.

## Cleaning up with DisposableEffect()

The **DisposableEffect()** composable function runs code when its key changes. Additionally, you can pass a lambda expression for cleanup purposes. It will be executed when the **DisposableEffect()** function leaves the composition. The code is illustrated in the following snippet:

```
DisposableEffect(clickCount) {
  println("init: clickCount is $clickCount")
  onDispose {
    println("dispose: clickCount is $clickCount")
```

```
    }
  }
```

A message starting with **`init:`** will be printed each time **`clickCount`** changes (that is, when **Start** or **Restart** is clicked). A message starting with **`dispose:`** will appear when **`clickCount`** changes or when **`DisposableEffect()`** leaves the composition.

*IMPORTANT NOTE*

**`DisposableEffect()`** *must include an* **`onDispose {}`** *clause as the final statement in its block.*

I have given you two hands-on examples that use side effects in a Compose app. The **`Effect`** APIs contain several other useful composables —for example, you can use **`SideEffect()`** to publish Compose state to non-Compose parts of your app, and **`produceState()`** allows you to convert non-Compose state into **`State`** instances.

You can find additional information about the **`Effect`** APIs at https://developer.android.com/jetpack/compose/side-effects.

# Summary

This chapter covered additional aspects of the **`ComposeUnitConverter`** example. We continued the exploration of the **`ViewModel`** pattern we began looking at in the *Using a ViewModel* section of *Chapter 5*, *Managing the State of Your Composable Functions*. This time, we added business logic to the **`ViewModel`** and injected an object that can persist and retrieve data.

The *Keeping your composables responsive* section revisited one of the key requirements of a composable function. Recomposition can occur very often, therefore composables must be as fast as possible, which dictates what code inside them may and may not do. I showed you how a simple loop can cause a Compose app to stop responding, and how coroutines counteract this.

In the final main section, *Understanding side effects*, we examined so-called side effects and used `LaunchedEffect` to implement a simple counter.

In *Chapter 8*, *Working with Animations*, you will learn how to show and hide UI elements with animations. We will spice up transitions through visual effects and use animation to visualize state changes.