

Chapter 12: Exploring the Jetpack Lifecycle Components

In this chapter, we're adding a countdown timer component to our Repositories app from [Chapter 11](#), *Creating Infinite Lists with Jetpack Paging and Kotlin Flow*, while also exploring the Jetpack Lifecycle components.

In the first section, *Introducing the Jetpack Lifecycle components*, we want to explore how the lifecycle events and states are tied to Android components such as **Activity** or **Fragment**, and then how predefined components from the **Lifecycle** package can react to them.

Next, in the *Adding a countdown component in the Repositories app* section, we will be creating and adding a countdown timer component to the Repositories app. When a 60-second countdown finishes, we will award users with a fictional prize.

However, we will want the countdown to run as long as the timer is visible on the screen; otherwise, users could cheat by minimizing the application and having the countdown run in background. In the *Creating your own lifecycle-aware component* section, we will prevent users from cheating by making our timer component aware of the different lifecycle events and states that our Android components traverse.

In the *Making our countdown component aware of the lifecycle of composables* section, we will realize that users can also cheat on the countdown contest by scrolling and hiding the timer countdown UI element. To prevent them from doing that, we will also make sure that our countdown component knows how to react to composition cycles that our Compose UI features.

To summarize, in this chapter, we will be covering the following sections:

- Introducing the Jetpack Lifecycle components
- Adding a countdown component in the Repositories app
- Creating your own lifecycle-aware component
- Making our countdown component aware of the lifecycle of composables

Before jumping in, let's set up the technical requirements for this chapter.

Technical requirements

Building Compose-based Android projects for this chapter usually requires your day-to-day tools. However, to follow along with this chapter smoothly, make sure that you also have the following:

- The Arctic Fox 2020.3.1 version of Android Studio. You can also use a newer Android Studio version or even Canary builds but note that the IDE interface and other generated code files might differ from the ones used throughout this book.
- A Kotlin 1.6.10 or newer plugin installed in Android Studio.
- The existing Repositories app from the GitHub repository of the book.

The starting point for this chapter is represented by the Repositories app developed in the previous chapter. If you haven't followed the implementation from the previous chapter, access the starting point for this chapter by navigating to the **Chapter_11** directory of the repository and importing the Android project entitled **repositories_app_solution_ch11**.

To access the solution code for this chapter, navigate to the **Chapter_12** directory: https://github.com/PacktPublishing/Kickstart-Modern-Android-Development-with-Jetpack-and-Kotlin/tree/main/Chapter_12/repositories_app_ch12.

Introducing the Jetpack Lifecycle components

It's no secret by now that components within the Android framework have certain lifecycles that we must respect when we need to interact with them. The most common components that own a lifecycle are **Activity** and **Fragment**.

As programmers, we cannot control the lifecycle of Android components because their lifecycle is defined and controlled by the system or the way Android works.

Going back to Lifecycle components, a very good example is the entry point to our Android application, represented by the **Activity** component, which, as we know, possesses a lifecycle. This means that in order to create a screen in our Android application, we need to create an **Activity** component – from this point on, all our components must be aware of its lifecycle to not leak any memory.

Now, when we say that **Activity** has a system-defined lifecycle, this actually translates into our **Activity** class inheriting from **ComponentActivity()**, which in turn contains a **Lifecycle** object. If we have a look at our **MainActivity** class from the Repositories app, we can see that it inherits from **ComponentActivity()**:

```
class MainActivity : ComponentActivity() {  
    override fun onCreate(savedInstanceState:  
Bundle?) {  
        super.onCreate(savedInstanceState)  
        [...]  
    }  
}
```

Now, if we dig into the source code of the **ComponentActivity.java** class, we can see that it implements the **LifecycleOwner** interface:

```
95 public class ComponentActivity extends androidx.core.app.ComponentActivity implements
96     ContextAware,
97     LifecycleOwner,
98     ViewModelStoreOwner,
99     HasDefaultViewModelProviderFactory,
100     SavedStateRegistryOwner,
101     OnBackPressedDispatcherOwner,
102     ActivityResultRegistryOwner,
103     ActivityResultCaller {
```

Figure 12.1 – Observing how ComponentActivity implements the LifecycleOwner interface

In other words, the **ComponentActivity** class is an owner of a lifecycle. If we check out the implementation of the **LifecycleOwner** interface a few hundreds of lines downward in the source code, we can see that the **LifecycleOwner** interface contains a single method called **getLifecycle()** that returns a **Lifecycle** object:

```
467 @NonNull
468 @Override
469 public Lifecycle getLifecycle() {
470     return mLifecycleRegistry;
471 }
472
```

Figure 12.2 – Observing the implementation of the LifecycleOwner interface method

From these findings, we can deduct that our **Activity** classes have a system-defined lifecycle, as they implement the **LifecycleOwner** interface, which in turn means that they own a **Lifecycle** object.

NOTE

*There are several other components in Android that have a lifecycle. In the context of the **Activity** classes, there are other classes inheriting directly or indirectly from **ComponentActivity**, therefore owning a **Lifecycle** object – see **AppCompatActivity** or **FragmentActivity**. Alternatively, just as **Activity** classes have a lifecycle, so do **Fragment** components. If you check out the*

*source code of the **Fragment** class, you will notice that it also implements the **LifecycleOwner** interface, and so it also contains a **Lifecycle** object.*

Simply put, the concept of a component having a lifecycle boils down to the idea of it providing a concrete implementation of the **Lifecycle** interface. This brings the idea that components with a lifecycle, such as **Activity**, expose information related to their lifecycle.

To better understand what we can find out about a component's lifecycle, we must explore the source code of the **Lifecycle** abstract class. If we do that, we will learn that the **Lifecycle** class contains information about the lifecycle state of the component that it's bound to, such as **Activity** or **Fragment**. The **Lifecycle** class features two main tracking pieces of information in the form of enumerations:

- **Event:** The events represented by the lifecycle callbacks that are triggered by the system and that we are all familiar with by now (**onCreate()**, **onStart()**, **onResume()**, **onPause()**, **onStop()**, and **onDestroy()**).
- **State:** The current state of the component tracked – **INITIALIZED**, **DESTROYED**, **CREATED**, **STARTED**, and **RESUMED**. If our **Activity** just received the **onResume()** callback, it means that until a new event arrives, it will stay in the **RESUMED** state. Upon every new event (the lifecycle callback), the state changes.

While we were already pretty familiar with the lifecycle events (callbacks), we might need to better understand how lifecycle states are defined.

Let's take a practical example and explore what information a **Lifecycle** object can provide about an **Activity** component. As previously mentioned, the information is structured in the form of events and states:

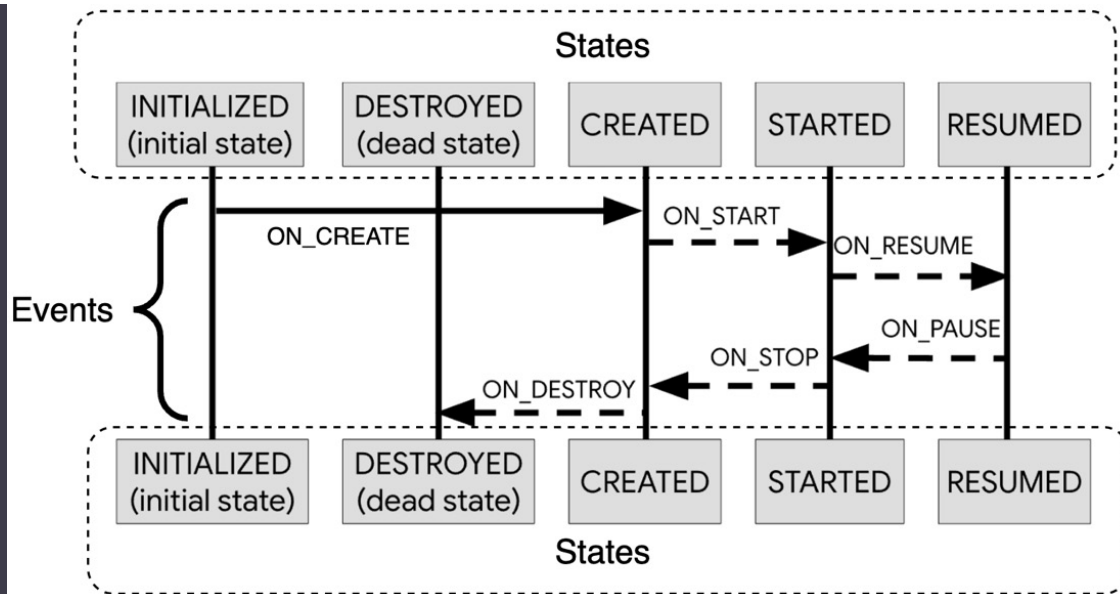


Figure 12.3 – The lifecycle of an Activity picturing its lifecycle events and states

In the preceding diagram, we were able to dissect the lifecycle of an **Activity** component by its events and states. We also now have a better overview of how lifecycle events trigger transitions between lifecycle states.

But why do all these events and states matter to us?

In fact, most of our code is driven with respect to lifecycle information. To avoid potential crashes, memory leaks, or wasting resources, it's essential to perform actions only in the correct state or on the correct lifecycle event.

When we think of lifecycle events, we can say that different types of functionalities can and should only be executed at appropriate times, or after certain lifecycle callbacks. For example, we wouldn't want to update our UI components with data after the `onDestroy()` callback in our **Activity**, as it's very likely that our app would crash simply because the UI has been scrapped by this time. Another example would be that when the `onResume()` event is called in our **Activity**, we would know that our **Activity** has gained (or regained) focus, so we can perform certain actions in our code such as initializing our camera component.

When we think of lifecycle states, we can say that different continuous actions can and should be running only during certain lifecycle periods – for example, we would want to start observing database changes if the state is **RESUMED** because that's when the user can interact with the screen and mutate data. When this state transitions to a different one, such as **CREATED** or **DESTROYED**, we might want to stop observing database changes so that we avoid memory leaks and don't waste resources.

From the previous examples, it's clear that our code should be aware of the lifecycle of Android components. When we write code based on lifecycle events or states, we're writing code that is aware of the lifecycle of a specific component.

Let's take an example and use our imagination a bit – the **Presenter** class features a data stream produced by several network requests. That data stream is observed and passed to the UI. However, any ongoing network requests must be canceled in the `cancelOngoingNetworkRequests()` method, as our UI no longer needs to consume their response:

```
class Presenter() {  
    // observe data and pass it to the UI  
    fun cancelOngoingNetworkRequests() {  
        // stop observing data  
    }  
}
```

Let's say that an instance of our **Presenter** class is used inside **MainActivity**. Naturally, it must respect the lifecycle of the **MainActivity** class. That's why we should stop any ongoing network requests from within the **Presenter** class by calling the `cancelOngoingNetworkRequests()` method of the **Presenter** class inside the `onDestroyed()` lifecycle callback of the **MainActivity** class:

```
class MainActivity : ComponentActivity() {  
    val presenter = Presenter()  
    override fun onStart() {
```



```
        super.onStart()  
        //consume data from presenter  
    }  
    override fun onDestroy() {  
        super.onDestroy()  
        presenter.cancelOngoingNetworkRequests()  
    }  
}
```

We can say that our **Presenter** is aware of the lifecycle of its host, **MainActivity**.

If a component respects the lifecycle of an Android component such as **Activity**, then we can consider that component to be **lifecycle-aware**.

However, we manually made our **Presenter** class be lifecycle-aware by manually calling a certain cleanup method from the **MainActivity** lifecycle callback. In other words, we had our **MainActivity** manually tell **Presenter** that it must stop its ongoing work.

Also, whenever we need to use our **Presenter** in some other **Activity** or **Fragment** classes, that component will need to remember to call the **cancelOngoingNetworkRequests()** method of **Presenter** on a certain lifecycle callback, therefore producing boilerplate code. If **Presenter** needed multiple actions on certain lifecycle callbacks, then that boilerplate code would have multiplied.

With the **Jetpack Lifecycle package**, we no longer need to manually force our Android components to call methods on every other class that cares about their lifecycle events or states. We can create lifecycle-aware components without having **Activity** or **Fragment** components manually inform our classes that a certain lifecycle event was triggered, or a certain state was reached – the **Lifecycle** package will help us receive the callbacks directly inside our components in a more efficient manner.

The Jetpack **Lifecycle** package provides us with the following:

- Predefined lifecycle-aware components with different purposes that require less boilerplate or work from our side. Such components are two Jetpack libraries:
 - **ViewModel**
 - **LiveData**
- A Lifecycle API that allows us to create a custom lifecycle-aware component much easier with less boilerplate code.

Before creating our own lifecycle-aware component, we should briefly cover the two predefined lifecycle-aware components that the Jetpack **Lifecycle** package provides us with. Let's begin with **ViewModel**.

ViewModel

In this book, we have already covered Jetpack's **ViewModel** as a class where our UI state resides and where most of the presentation logic is found. However, we also learned that in order to properly cancel data streams or ongoing network requests, **ViewModel** is aware of the lifecycle of its host **Activity**, **Fragment**, and even its composable destination (in conjunction with the Jetpack Navigation component).

In contrast to our **Presenter** class, whose lifecycle we have manually tied to the lifecycle of a host **Activity**, Jetpack's **ViewModel** is a lifecycle-aware component that we can use to eliminate any boilerplate calls from **Activity** or **Fragment** components.

To be more precise, **ViewModel** knows when its host component with a lifecycle reaches the end of its lifecycle and provides us with a callback method that we can use by overriding the **onCleared()** method. Inside this callback, we can cancel any pending work whose result we're no longer interested in to avoid memory leaks or wasting resources.

As an example, if our **ViewModel** is hosted by an **Activity**, then it knows when in the lifecycle of that **Activity** the **onDestroy()** event was called, and so it automatically triggers the **onCleared()** callback:

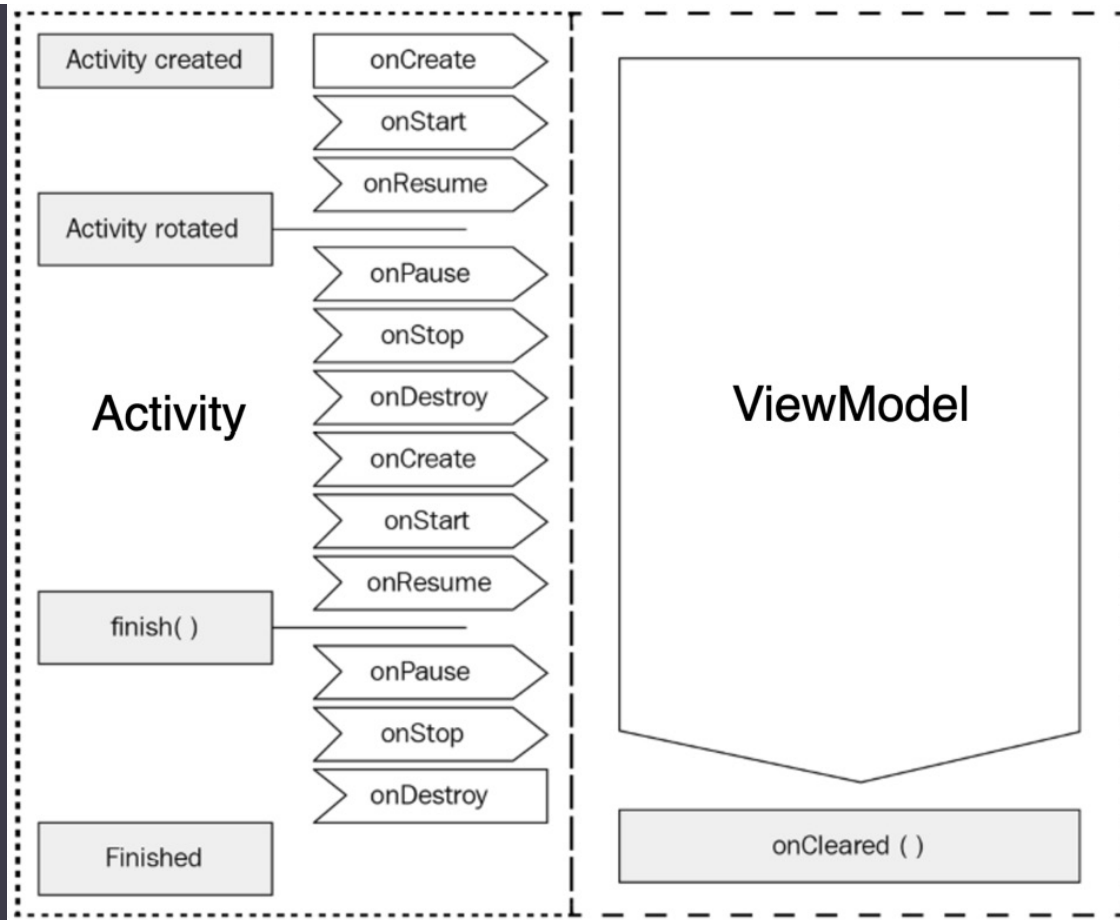


Figure 12.4 – The lifecycle of **ViewModel** is tied to the lifecycle of an **Activity**

This basically means that instead of manually having our **Activity** inform the **ViewModel** that its lifecycle has ended so that it can stop its work, **ViewModel** is a lifecycle-aware component that does that out of the box for you by providing a handle for that event – that is, the **onCleared()** callback:

```
class MyViewModel(): ViewModel() {
    override fun onCleared() {
        super.onCleared()
        // Cancel work
    }
}
```

Additionally, in the context of an **Activity** host, the **ViewModel** component is also aware of any lifecycle callbacks caused by events such as a config-

uration change, so it knows how to outlive those and helps us maintain the UI state, even after a configuration change.

But how does **ViewModel** know about the lifecycle callbacks of an **Activity** component? To answer that, we can look at a traditional way of instantiating a **ViewModel** inside an **Activity** by using the **ViewModelProvider** API and specifying the type of **ViewModel** that must be retrieved – that is, **MyViewModel**:

```
class MyActivity: ComponentActivity() {  
    override fun onCreate(savedInstanceState:  
Bundle?) {  
        super.onCreate(savedInstanceState)  
        val vm =  
            ViewModelProvider(this)  
[MyViewModel::class.java]  
        // Perform operations  
    }  
}
```

To get an instance of **MyViewModel**, we used the **ViewModelProvider()** constructor and passed the **this** instance of the **MyActivity** class to the **owner** parameter that expected a **ViewModelStoreOwner** object. **MyActivity** indirectly implements the **ViewModelStoreOwner** interface because **ComponentActivity** does so.

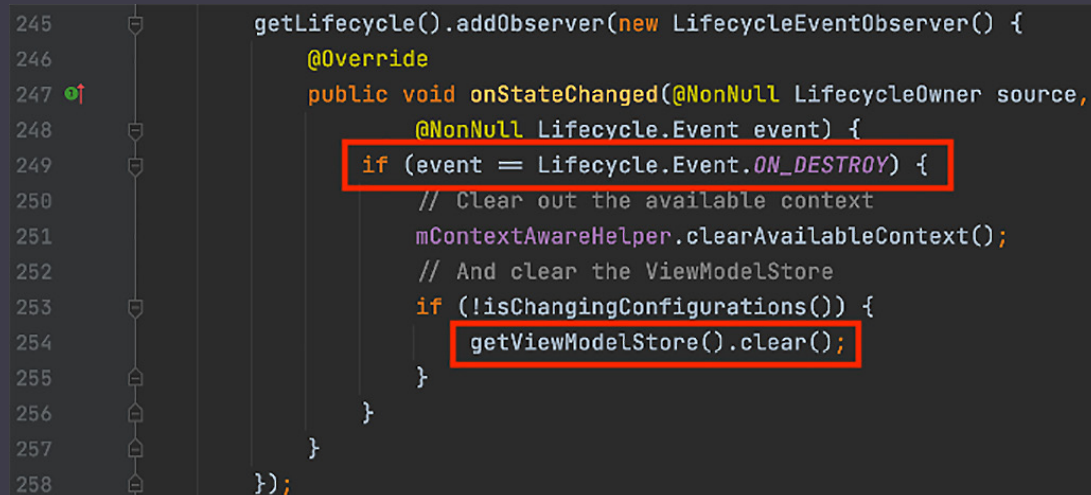
To control the lifetime of the instance of our **ViewModel**, **ViewModelProvider** needs an instance of **ViewModelStoreOwner** because when it instantiates our **MyViewModel**, it will *link* the lifetime of this instance to the lifetime of the **ViewModelStoreOwner** – that is, **MyActivity**.

But how does **ViewModel** know when it must be cleared? In other words, what triggers the **onCleared()** method of the **MyViewModel** class?

ComponentActivity will wait for its **onDestroy()** lifecycle callback, and when that event is triggered, it will call the **getViewModelStore()** method

of the **ViewModelStoreOwner** interface and obtain a **ViewModelStore** object. On this object, it will then call the **clear()** method to clear the **ViewModel** instance that was linked to **ComponentActivity** – in our case, the **MyViewModel** instance.

If you check out the source code of the **ComponentActivity** class, you will find the following implementation, which proves the previous points we're trying to express:



```
245     getLifecycle().addObserver(new LifecycleEventObserver() {
246         @Override
247         public void onStateChanged(@NonNull LifecycleOwner source,
248             @NonNull Lifecycle.Event event) {
249             if (event == Lifecycle.Event.ON_DESTROY) {
250                 // Clear out the available context
251                 mContextAwareHelper.clearAvailableContext();
252                 // And clear the ViewModelStore
253                 if (!isChangingConfigurations()) {
254                     getViewModelStore().clear();
255                 }
256             }
257         }
258     });
```

Figure 12.5 – ViewModel is cleared on the onDestroy() callback of ComponentActivity

Now, the **ViewModel** lifecycle-aware component is helpful because it allows us to easily stop pending work and also persist UI state across configuration changes.

However, there is another important lifecycle-aware component that we haven't covered in this book and that we should briefly mention, and that is **LiveData**.

LiveData

LiveData is an observable data holder class that allows us to get data updates in a lifecycle-aware manner inside our Android components, such as **Activity** and **Fragment**. While specific implementations of Kotlin Flow data streams are similar to **LiveData** because both allow us to receive

multiple data events over time, **LiveData** presents the advantage of being a lifecycle-aware component.

NOTE

*In this section, we won't cover **LiveData** extensively to understand its API. Instead, we will try to highlight its lifecycle-aware character. Right now, you don't have to code along.*

Without going into too much detail, let's see a simple usage of a **LiveData** object kept inside a **ViewModel** class and consumed from an **Activity** component.

Inside **ViewModel**, we instantiated a **MutableLiveData** object that will hold values of type **Int**, passed an initial value of **0**, and then in the **init{}** block launched a coroutine, where we've set the value to **100** after a **5000**-millisecond delay:

```
class MyViewModel(): ViewModel() {
    val numberLiveData: MutableLiveData<Int> =
        MutableLiveData(0)
    init {
        viewModelScope.launch {
            delay(5000)
            numberLiveData.value = 100
        }
    }
}
```

numberLiveData is now a data holder that will first notify any components observing it of the value **0** and, after 5 seconds, the value **100**.

Now, an **Activity** can be observing these values by first obtaining an instance of **MyViewModel**, tapping into its **numberLiveData** object, and then starting to observe the changes through the **observe()** method:

```
class MyActivity: ComponentActivity() {
```

```
        override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            val vm =
                ViewModelProvider(this)
                [MyViewModel::class.java]
            vm.numberLiveData.observe(this, object:
            Observer<Int> {
                override fun onChanged(t: Int?) {
                    // Consume values
                }
            })
        }
    }
```

Now, to the `observe()` method, we've passed the following:

- First, the `this` instance of the `MyActivity` class to the `owner` parameter that expected a `LifecycleOwner` object. This worked because `MyActivity` indirectly implements (through `ComponentActivity`) the `LifecycleOwner` interface and therefore owns a `Lifecycle` object. The `observe()` method expected a `LifecycleOwner` as its first parameter, so that the observing feature is lifecycle-aware of the lifecycle of `MainActivity`.
- An `Observer<Int>` Kotlin inner `object` that allows us to receive the data events (holding the `Int` values) from the `MutableLiveData` object inside the `onChanged()` callback. Each time a new value is propagated, this callback will be triggered, and we will receive the latest value.

Now that we have briefly covered how to use `LiveData`, let's better understand the whole reason why we are talking about `LiveData`. As we've mentioned, `LiveData` is a lifecycle-aware component, but how does it achieve that?

When we passed our `MainActivity` as `LifecycleOwner` to the `owner` parameter of the `observe()` method, behind the scenes, `LiveData` started an observing process dependent on the `Lifecycle` object of the provided `owner`.

More precisely, the **Observer** object provided as the second parameter to the **observe()** method will only receive updates if the owner – that is, **MainActivity** – is in the **STARTED** or **RESUMED** lifecycle state.

This behavior is essential, as it allows Activity components to only receive UI updates from ViewModel components when they are visible or in focus, therefore making sure that the UI can safely handle the data events and not waste resources.

If, however, updates would have occurred in other states when the UI would not have been initialized, our app could have misbehaved or, even worse, crashed or introduced memory leaks. To be sure that such behavior doesn't occur, if the owner moves to the **DESTROYED** state, the **Observer** object will be automatically removed.

In the following diagram, you will be able to visualize how **LiveData** updates only come when the **Activity** component is in the **RESUMED** or **STARTED** state, while also automatically removing the **Observer** object when the state becomes **DESTROYED**:

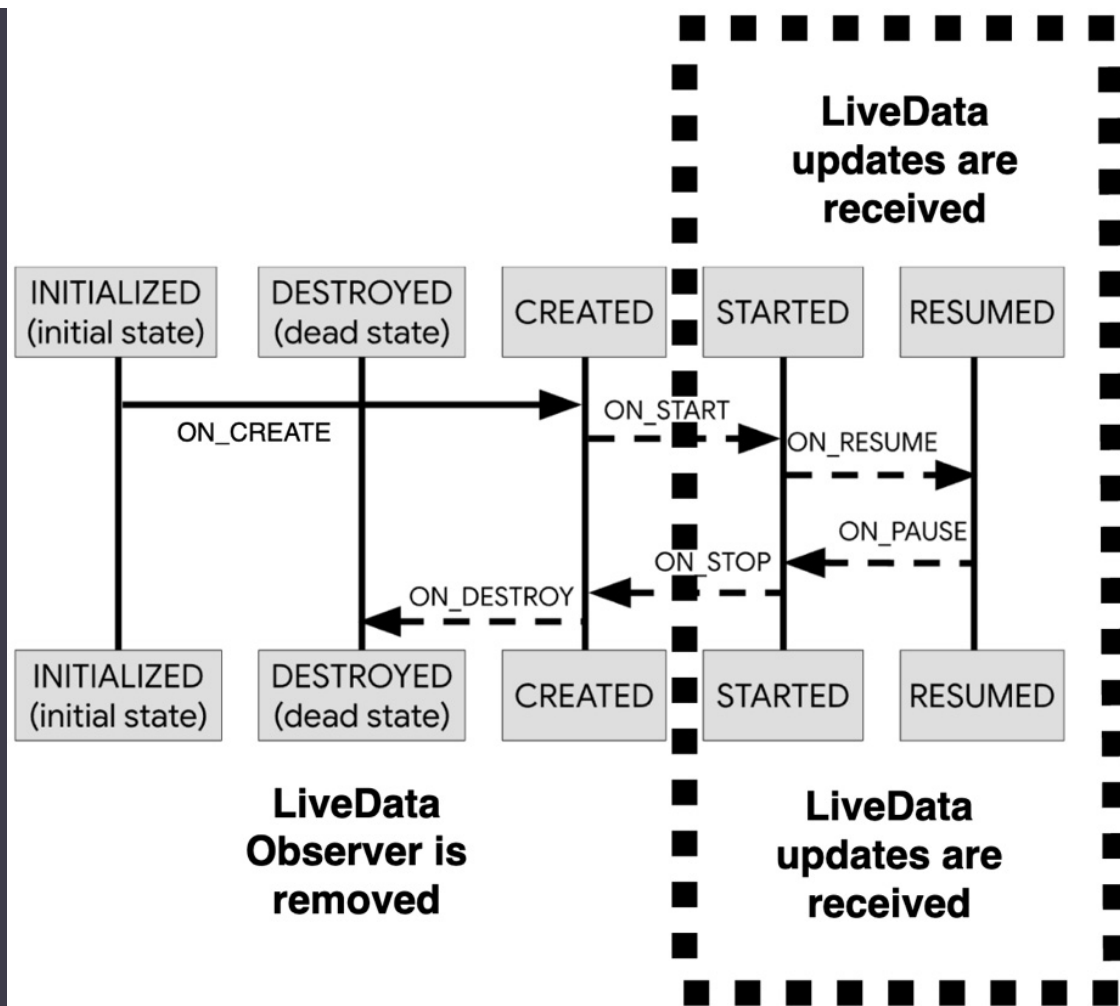


Figure 12.6 – Lifecycle states and events when LiveData updates are received and the LiveData Observer is removed

With such behavior, **LiveData** becomes a lifecycle-aware component in the sense that any **LifecycleOwner** must be in an active lifecycle state to be receiving updates from it.

Now that we have covered the two predefined lifecycle-aware components (**ViewModel** and **LiveData**) that are part of the **Lifecycle** package, it's time to add a countdown timer component in our **Repositories** app so that later on, we can transform it into a custom lifecycle-aware component with the help of the **Lifecycle** APIs.

Adding a countdown component in the **Repositories** app

Our plan is to learn how to create our own lifecycle-aware component. However, before we can do that, we must first create a normal component that, by default, is not aware of the lifecycle of any Android component.

To do that, we can create a countdown timer component inside our **Repositories** app that will track whether the user has spent at least 60 seconds on the app, and if so, we will award the user with a fictional prize.

More precisely, our plan is to create a countdown timer widget inside the **RepositoriesScreen()** that will award the user with a prize upon a 60-second countdown. However, for the countdown to work and the prize to be awarded, the user must be inside **RepositoriesScreen()** and have the countdown composable visible.

The countdown will behave like so:

- It will start from 60 and finish when the countdown reaches 0. Upon every second, the timer will decrease by 1 unit.
- When the countdown has finished, a prize message will be displayed.
- It will be paused if the countdown composable is not visible. In other words, if the user is not inside the **RepositoriesScreen()** composable or the timer composable is not visible or hidden within **RepositoriesScreen()**, then the countdown should be paused.

Now that we have a plan, let's implement a countdown timer component:

1. Inside the root package, create a new class called **CustomCountdown** and define its constructor to feature two function parameters that will be called as the countdown timer functions:

```
class CustomCountdown(  
    private val onTick: ((currentValue: Int) ->  
Unit),  
    private val onFinish: (() -> Unit),  
) {
```

}

We will have to call the `onTick()` function after every second has passed and the `onFinish()` function when the countdown has ended.

2. Now, inside the `CustomCountdown` class, let's create an inner class called `InternalTimer` that will inherit from the built-in Android `android.os.CountDownTimer` class and handle the actual countdown sequence:

```
class CustomCountdown(
    private val onTick: ((currentValue: Int) ->
Unit),
    private val onFinish: (() -> Unit),
) {
    class InternalTimer(
        private val onTick: ((currentValue: Int) ->
Unit),
        private val onFinish: (() -> Unit),
        millisInFuture: Long,
        countDownInterval: Long
    ) : CountDownTimer(millisInFuture,
        countDownInterval){

    }
}
```

While the constructor of `InternalTimer` also accepts two identical function parameters, as `CustomCountdown` does, it's essential to note its `millisInFuture` and `countDownInterval` parameters that it forwards to the built-in Android `CountDownTimer` class. These two parameters will configure the core functionality of the timer – the countdown starting point in time and the time period that passes between timer ticks.

3. Next up, let's finish the implementation of the `InternalTimer` class:

```
class CustomCountdown(
```

```
        private val onTick: ((currentValue: Int) ->
Unit),
        private val onFinish: (() -> Unit),
    ) {
        class InternalTimer(
            private val onTick: ((currentValue: Int) ->
Unit),
            private val onFinish: (() -> Unit),
            millisInFuture: Long,
            countDownInterval: Long
        ) : CountDownTimer(millisInFuture,
            countDownInterval) {
            init {
                this.start()
            }
            override fun onFinish() {
                onFinish.invoke()
            }
            override fun onTick(millisUntilFinished:
Long) {
                onTick(millisUntilFinished.toInt())
            }
        }
    }
}
```

To make sure the timer works as expected, we have done the following:

- Called the **start()** method provided by the inherited parent, **CountDownTimer**, inside the **init{} block**. This should automatically start the timer upon inception.
- Implemented the two mandatory **onFinish()** and **onTick()** methods of the inherited parent, **CountDownTimer**, and propagated the events to the caller of **InternalTimer** by calling its **onFinish()** and **onTick()** function parameters.

4. Then, back in the **CustomCountdown** class, let's create an instance of **InternalTimer** and configure it to work like a 60-second countdown timer that starts from **60** and finishes at **0**.

To do that, let's pass to its constructor not only the **onFinish** and **onTick** function parameters but also 60 seconds (as **60000** milliseconds) to the **millisInFuture** parameter and 1 second (as **1000** milliseconds) to the **countDownInterval** parameter:

```
class CustomCountdown(
    private val onTick: ((currentValue: Int) ->
Unit),
    private val onFinish: (() -> Unit),
) {
    var timer: InternalTimer = InternalTimer(
        onTick = onTick,
        onFinish = onFinish,
        millisInFuture = 60000,
        countDownInterval = 1000)
    class InternalTimer(
        private val onTick: ((currentValue: Int) ->
Unit),
        private val onFinish: (() -> Unit),
        millisInFuture: Long,
        countDownInterval: Long
    ): CountdownTimer(millisInFuture,
countDownInterval)
        { ... }
}
```

5. Still inside **CustomCountdown**, to provide a way for canceling the count-down, add a **stop()** method that will allow us to call the **cancel()** method inherited by **InternalTimer** from the Android **CountDownTimer** class:

```
class CustomCountdown(...) {
```

```

var timer: InternalTimer = InternalTimer(...)
fun stop() {
    timer.cancel()
}
class InternalTimer(
    [...]
): CountdownTimer(millisInFuture,
countDownInterval)
{ ... }
}

```

6. Then, in `RepositoriesViewModel`, add not only a `timerState` variable that will hold the text state displayed by our countdown composable but also a `timer` variable that will hold a `CustomCountdown` object:

```

class RepositoriesViewModel(...) : ViewModel() {
    val repositories: Flow<PagingData<Repository>>
= [...]
    val timerState = mutableStateOf("")
    var timer: CustomCountdown = CustomCountdown(
        onTick = { msLeft ->
            timerState.value =
                (msLeft / 1000).toString() +
                    " seconds left"
        },
        onFinish = {
            timerState.value = "You won a prize!"
        })
}

```

Inside the `onTick` callback, we are computing the remaining seconds and setting a `String` message about our countdown to `timerState`. Then, in the `onFinish` callback, we're setting a prize message to `timerState`.

7. As a good practice, inside `RepositoriesViewModel`, make sure to stop the timer inside the `onCleared()` callback if the user moves to a different screen. This would mean that `RepositoriesScreen()` wouldn't be com-

posed anymore, so this **ViewModel** would be cleared and the countdown should be stopped so that it doesn't send events and waste resources:

```
class RepositoriesViewModel(...) : ViewModel() {
    val repositories: Flow<PagingData<Repository>>
= [...]
    val timerState = mutableStateOf("")
    var timer: CustomCountdown = CustomCountdown(...)
    override fun onCleared() {
        super.onCleared()
        timer.stop()
    }
}
```

8. Now, move to **MainActivity** and make sure that just as the repositories are consumed and passed to the **RepositoriesScreen()** composable, the countdown timer text produced by **ViewModel** is also consumed and passed to **RepositoriesScreen()**:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            RepositoriesAppTheme {
                val viewModel:
RepositoriesViewModel = ...
                val reposFlow =
viewModel.repositories
                val timerText =
                    viewModel.timerState.value
                val lazyRepoItems: [...] = [...]
                RepositoriesScreen(
                    lazyRepoItems,
                    timerText
                )
            }
        }
    }
}
```



```

    }
}
}

```

9. Then, at the end of the **RepositoriesScreen.kt** file, create a simple **CountdownItem()** composable function that takes in a **timerText: String** parameter and sets its value to a **Text** composable:

```

@Composable
private fun CountdownItem(timerText: String) {
    Text(timerText)
}

```

10. Next, in the **RepositoriesScreen()** composable, add a new parameter for the countdown text called **timerText**, and inside the **LazyColumn** scope, before the **itemsIndexed()** call, add a singular **item()** **Domain-Specific Language (DSL)** function call where you should add the **CountdownItem()** composable while passing the **timerText** variable to it:

```

@Composable
fun RepositoriesScreen(
    repos: LazyPagingItems<Repository>,
    timerText: String
) {
    LazyColumn(...) {
        item {
            CountdownItem(timerText)
        }
        itemsIndexed(repos) { index, repo -> [...] }
        [...]
    }
}

```

By doing so, we make sure that the countdown timer is displayed at the top of the screen as the first item within the list of repositories.

11. Build and run the application. You should first see the countdown timer telling you how much time you need to wait, and after approximately 1 minute, you should see the prize message displayed:

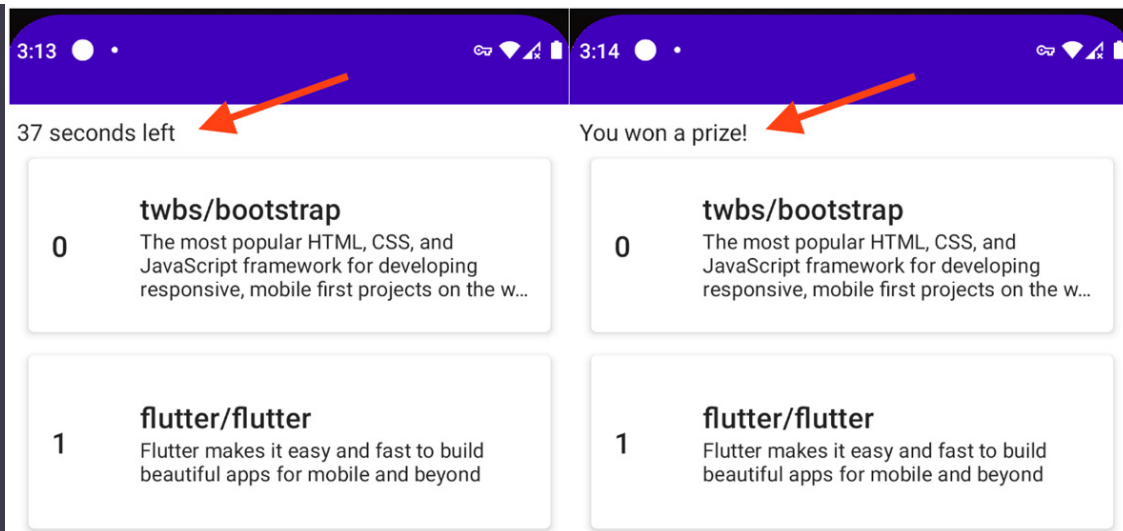


Figure 12.7 – Observing how the countdown timer works

We have now finished incorporating the countdown timer that ends by awarding the user with a fictional prize.

However, there is a scenario where our timer doesn't work as expected. Let's identify it:

1. Restart the application. You can do that by closing the current instance of the app and reopening it.

The countdown should have started from 60 again at this point.

2. Before the countdown finishes, remember or write down somewhere the current countdown value and then put the app in background.
3. Wait for a few seconds and then bring the app back to the foreground.

You should notice that while the app was in the background, the countdown kept going. We wanted the timer to be paused when the app was put in the background and then resumed when the app was brought back to the foreground – this would have allowed us to award the prize to users that actively use the app and have the countdown timer visible. This behavior didn't occur, as the timer kept on counting while the app was not visible or in focus.

This is happening because we didn't do anything to pause the timer when the app goes into the background or resume it when the app comes back to the foreground. In other words, our countdown timer is not lifecycle-aware, so it doesn't get notified and can't react to the lifecycle events of the **Activity** host.

Next, let's make our countdown timer a lifecycle-aware component.

Creating your own lifecycle-aware component

We need to make our **CustomCountdown** aware of the lifecycle of **MainActivity**. In other words, our countdown logic should observe and react to the lifecycle events of our **LifecycleOwner** – that is, **MainActivity**.

To make our **CustomCountdown** lifecycle-aware, we must force it to implement the **DefaultLifecycleObserver** interface. By doing so, the **CustomCountdown** will be observing the lifecycle events or states defined by the **Lifecycle** object that **LifecycleOwner** provides.

Our main goal is to pause the countdown when the app is put in the background and to resume it when the app is brought back into the foreground. More precisely, our **CustomCountdown** must react to the following lifecycle events of **MainActivity**:

- **onPause()**: When the **onPause()** callback comes in **MainActivity**, **CustomCountdown** must pause its countdown.
- **onResume()**: When the **onResume()** callback comes in **MainActivity**, **CustomCountdown** must resume its countdown.

With this behavior, we can award the prize to users that actively use the app and have the countdown timer visible and in focus.

Now that we have a plan, let's start coding.

1. Make the **CustomCountdown** class implement the **DefaultLifecycleObserver** interface and then override the two lifecycle callbacks we're interested in, **onResume()** and **onPause()**:

```
class CustomCountdown(
    [...]
): DefaultLifecycleObserver {
    var timer: InternalTimer = InternalTimer(
        onTick = onTick,
        onFinish = onFinish,
        millisInFuture = 60000,
        countdownInterval = 1000)

    override fun onResume(owner: LifecycleOwner) {
        super.onResume(owner)
    }

    override fun onPause(owner: LifecycleOwner) {
        super.onPause(owner)
    }

    fun stop() { timer.cancel() }
    class InternalTimer(...) {...}
}
```

Once we make our **CustomCountdown** observe the lifecycle of **MainActivity**, its **onResume(owner: LifecycleOwner)** callback will be called when the **onResume()** callback of **MainActivity** is called, and similarly, its **onPause(owner: LifecycleOwner)** callback will be called when the **onPause()** callback of **MainActivity** is called.

2. Now that we know when to pause and resume our countdown timer, we need to find ways to actually pause and resume it.

First, let's pause the countdown in the **onPause()** callback by calling the **cancel()** method of the **timer** variable:

```
class CustomCountdown(
    [...]
```

```

): DefaultLifecycleObserver {
    var timer: InternalTimer = InternalTimer(...)
    override fun onResume(owner: LifecycleOwner) {
        super.onResume(owner)
    }
    override fun onPause(owner: LifecycleOwner) {
        super.onPause(owner)
        timer.cancel()
    }
    fun stop() { timer.cancel() }
    class InternalTimer(...) : CountdownTimer(...) {...}
}

```

With this behavior, when **MainActivity** is paused, we are stopping the countdown run by the **InternalTime** instance held inside the **timer** variable.

3. Next up, we need to resume the **timer** in the **onResume()** callback.

However, to resume it, we need to know the value of the last countdown before the **onPause()** callback was triggered and the timer was canceled. With that last known countdown value, we can reinitiate our timer in the **onResume()** callback.

Inside the inner **InternalTimer** class, create a **lastKnownTime** variable, initiate it with the value of **millisInFuture**, and then make sure to update it in the **onFinish()** and **onTick()** timer callbacks:

```

class CustomCountdown(
    [...]
): DefaultLifecycleObserver {
    var timer: InternalTimer = InternalTimer(
        [...]
        millisInFuture = 60000,
        countdownInterval = 1000)
    override fun onResume(owner: LifecycleOwner) { ...
}

```

```

        override fun onPause(owner: LifecycleOwner) { ... }
        fun stop() { timer.cancel() }

        class InternalTimer(...) : CountdownTimer(...) {
            var lastKnownTime: Long = millisInFuture
            init { this.start() }
            override fun onFinish() {
                lastKnownTime = 0
                onFinish.invoke()
            }
            override fun onTick(millisUntilFinished:
Long) {
                lastKnownTime = millisUntilFinished
                onTick(millisUntilFinished.toInt())
            }
        }
    }
}

```

While in the `onFinish()` callback, we've set `lastKnownTime` to `0` because the countdown has finished, in the `onTick()` callback, we've made sure to save inside the `lastKnownTime` variable the latest value received from the `onTick()` callback – that is, `millisUntilFinished`.

4. Now, going back in the parent `CustomCountdown` class, resume the countdown in the `onResume()` callback of `CustomCountdown` by first canceling the countdown of the previous timer and then by storing inside the `timer` variable another instance of `InternalTimer`, which now starts the countdown from the `lastKnownTime` value of the previous `InternalTimer` instance:

```

class CustomCountdown(
    [...]
): DefaultLifecycleObserver {
    var timer: InternalTimer = InternalTimer(
        onTick = onTick,
        onFinish = onFinish,
        millisInFuture = 60000,

```

```
        countdownInterval = 1000)
    override fun onResume(owner: LifecycleOwner) {
        super.onResume(owner)
        if (timer.lastKnownTime > 0) {
            timer.cancel()
            timer = InternalTimer(
                onTick = onTick,
                onFinish = onFinish,
                millisInFuture =
timer.lastKnownTime,
                countdownInterval = 1000)
        }
    }
    override fun onPause(owner: LifecycleOwner) {
[...] }
    fun stop() { timer.cancel() }
    class InternalTimer(...) : CountdownTimer(...) {...}
}
```

With this behavior, when **MainActivity** is resumed, we are creating a new **InternalTimer** instance that starts off the countdown from the value that the previous timer recorded before being paused. Also, note that the new instance of **InternalTimer** receives the same parameters as the first initialization of the **timer** variable – the same **onTick()** and **onFinish()** callbacks and the same **countdownInterval** – the only difference is the starting point of the countdown, which should now be less than 60 seconds.

For the **onPause()** and **onResume()** callbacks of the **CustomCountdown** class to be called when their corresponding lifecycle events are called inside **MainActivity**, we must effectively bind our **DefaultLifecycleObserver** – that is, the **CustomCountdown** instance – to the lifecycle of our **LifecycleOwner** – that is, **MainActivity**.

Let's do that next.

5. Go back inside the **RepositoriesScreen.kt** file, and inside the **CountdownItem()** composable, first obtain the **LifecycleOwner** instance that the composable function belongs to by tapping into the **LocalLifecycleOwner** API and then get the owner by accessing its **current** variable:

```
@Composable
private fun CountdownItem (timerText: String) {
    val lifecycleOwner: LifecycleOwner =
        LocalLifecycleOwner.current
    Text(timerText)
}
```

Finally, we've stored the **LifecycleOwner** instance into the **lifecycleOwner** variable.

It's important to mention that since the parent composable of **CountdownItem()** – that is, **RepositoriesScreen()** – is hosted by **MainActivity**, it's only natural that the **LifecycleOwner** instance that we have obtained is in fact **MainActivity**.

6. Then, we need to make sure that the **Lifecycle** instance of our **lifecycleOwner** adds and removes our **DefaultLifecycleObserver** timer.

To achieve that, we need to first create a composition side effect that allows us to know when the **CountdownItem()** composable first entered composition so that we can add the observer, and then when it was removed from composition so that we can remove the observer.

For such a case, we can use the **DisposableEffect()** composable, which provides us with a block of code where we can perform actions when the composable enters composition, and then perform other actions when the composable leaves composition through its inner **onDispose()** block:

```
@Composable
private fun CountdownItem (timerText: String) {
```

```

val lifecycleOwner: LifecycleOwner =
    LocalLifecycleOwner.current

DisposableEffect(key1 = lifecycleOwner) {
    onDispose {

    }
}

Text(timerText)
}

```

Since this is a side effect, anything we add inside the block of code exposed by the **DisposableEffect** function will not be re-executed upon re-composition. However, this effect will be restarted if the value provided to the **key1** parameter changes. In our case, we want this effect to be restarted if the value of **lifecycleOwner** changes - this will allow us to have access to the correct **lifecycleOwner** instance inside this side-effect composable.

7. Now that we know when and where we can add and then remove the observer, let's first obtain the **Lifecycle** object from the **lifecycleOwner** variable so that we can store it inside the **lifecycle** variable:

```

@Composable
private fun CountdownItem(timerText: String) {
    val lifecycleOwner: LifecycleOwner =
        LocalLifecycleOwner.current
    val lifecycle = lifecycleOwner.lifecycle
    DisposableEffect(key1 = lifecycleOwner) {
        onDispose {

        }
    }
    Text(timerText)
}

```

Next, on the **Lifecycle** object from within the **lifecycle** variable, we will add and remove the observer.

8. Inside the block of code exposed by the **DisposableEffect()** composable, add the observer on the **lifecycle** variable by calling its **addObserver()** method, and then inside its exposed **onDispose()** callback, remove it with the **removeObserver()** method:

```
@Composable
private fun CountdownItem(timerText: String) {
    val lifecycleOwner: LifecycleOwner
        = LocalLifecycleOwner.current
    val lifecycle = lifecycleOwner.lifecycle
    DisposableEffect(key1 = lifecycleOwner) {
        lifecycle.addObserver()
        onDispose {
            lifecycle.removeObserver()
        }
    }
    Text(timerText)
}
```

With this approach, when the **CountdownItem()** composable is first composed, we will make our countdown component observe the lifecycle events of **MainActivity**. Then, when the **CountdownItem()** leaves composition, our countdown component will no longer observe such events.

However, you might have noticed that both the **addObserver()** and **removeObserver()** methods expect a **LifecycleObserver** object, but we didn't provide any.

In fact, we should have passed the **CustomCountdown** instance to the **addObserver()** and **removeObserver()** methods because **CustomCountdown** is the component that implements **DefaultLifecycleObserver** and that we want to react to the lifecycle changes of our **MainActivity**.

Next, let's obtain the **CustomCountdown** instance.

9. Update the `CountdownItem()` function definition to receive a `getTimer()` function parameter that returns a `CustomCountdown` timer. This callback method should be called to provide the `addObserver()` and `removeObserver()` methods with a `LifecycleObserver` instance:

```
@Composable
private fun CountdownItem(timerText: String,
    getTimer: () -> CustomCountdown) {
    val lifecycleOwner: LifecycleOwner
        = LocalLifecycleOwner.current
    val lifecycle = lifecycleOwner.lifecycle
    DisposableEffect(key1 = lifecycleOwner) {
        lifecycle.addObserver(getTimer())
        onDispose {
            lifecycle.removeObserver(getTimer())
        }
    }
    Text(timerText)
}
```

Since the `CustomCountdown` class implements `DefaultLifecycleObserver`, which extends `FullLifecycleObserver`, which in turn extends `LifecycleObserver`, the `addObserver()` and `removeObserver()` methods accept our `CustomCountdown` instance as an observer to the `Lifecycle` object of our `lifecycleOwner` – that is, `MainActivity`.

10. Since `CountdownItem()` now expects a `getTimer: () -> CustomCountdown` callback function, we must also force our `RepositoriesScreen()` composable to accept such a callback function as well and then pass it to our `CountdownItem()` composable:

```
@Composable
fun RepositoriesScreen(
    repos: LazyPagingItems<Repository>,
    timerText: String,
    getTimer: () -> CustomCountdown
) {
```

```

        LazyColumn(...) {
            item {
                CountdownItem(timerText, getTimer)
            }
            itemsIndexed(repos) { ... }
            [...]
        }
    }
}

```

11. Lastly, inside **MainActivity**, update the **RepositoriesScreen()** composable call to provide a **getTimer()** function implementation, where we will get the **CustomCountdown** instance from the **viewModel** variable through its **timer** field:

```

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            RepositoriesAppTheme {
                [...]
                RepositoriesScreen(
                    lazyRepoItems,
                    timerText,
                    getTimer = {viewModel.timer}
                )
            }
        }
    }
}

```

We have finally tied our **DefaultLifecycleObserver** – that is, the **CustomCountdown** instance – to the lifecycle of our **LifecycleOwner** – that is, **MainActivity**. Now that the **CustomCountdown** class should react to the lifecycle events of our **MainActivity**, let's test our problematic scenario from before.

12. Build and run the app. The countdown should have started from 60 again at this point.
13. Before the countdown finishes, remember or write down somewhere the current countdown value and put the app in background.
14. Wait for a few seconds and then bring the app back to foreground.

You should now notice that while the app was in background, the countdown was paused. We wanted the timer to be paused when the app was put in background and then resumed when the app was brought back to foreground – and now this is happening! We can now award the prize to users that actively use the app.

However, there is still an edge case that we haven't covered. Let's discover it:

1. Build and run the app. The countdown should have started from 60 again at this point.
2. Before the countdown finishes, remember or write down somewhere the current countdown value and then quickly scroll down past four or five repositories within the list until the countdown is not visible anymore.
3. Wait for a few seconds and then scroll back up to the top of the list so that the countdown is visible again.

Note that after we scrolled down, while the timer wasn't visible, the countdown kept going. We wanted the timer to be paused when the timer isn't visible anymore and then resumed when the timer is visible again – this would have allowed us to award the prize to users that have the countdown timer visible so that they didn't cheat on our contest. This behavior didn't occur, as the timer kept on counting while the timer wasn't visible.

This is happening because we didn't do anything to pause the timer when the timer composable leaves composition or resume it when the timer

composable is composed again. In other words, our countdown timer is not aware of the lifecycle of our timer composable.

Next, let's make our countdown timer aware of Compose composition cycles so that users don't cheat in our contest.

Making our countdown component aware of the lifecycle of composables

The main issue is that our `CustomCountdown` component still runs its countdown even after the `CountdownItem()` composable leaves composition. We want to pause the timer when its corresponding composable is not visible anymore. With such an approach, we can prevent users from cheating, and we can award the prize only to users that have had the countdown timer visible for the full amount of time. Basically, if the timer is not visible anymore, the countdown should stop.

To pause the timer when its corresponding composable function leaves composition, we must somehow call the `stop()` function exposed by `CustomCountdown`. But when should we do that?

If you look inside the body of the `CountdownItem()` composable, you will notice that we have already registered a `DisposableEffect()` composable that notifies us when the `CountdownItem()` composable leaves composition by exposing the `onDispose()` callback:

```
@Composable
private fun CountdownItem(...) {
    val lifecycleOwner: [...] =
        LocalLifecycleOwner.current
    val lifecycle = lifecycleOwner.lifecycle
    DisposableEffect(key1 = lifecycleOwner) {
```



```

        lifecycle.addObserver(getTimer())
        onDispose {
            lifecycle.removeObserver(getTimer())
        }
    }
    Text(timerText)
}

```

When the composable leaves composition, inside the `onDispose()` callback, we are already removing the `CustomCountdown` as an observer to the lifecycle of our `MainActivity`. Exactly at this point, we can also pause the timer because the composable leaves composition:

1. Update the `CountdownItem()` function definition to accept a new `onPauseTimer()` callback function and then make sure to call it inside the `onDispose()` callback of `DisposableEffect()`:

```

@Composable
private fun CountdownItem(timerText: String,
    getTimer: () -> CustomCountdown,
    onPauseTimer: () -> Unit) {
    val lifecycleOwner: [..] =
        LocalLifecycleOwner.current
    val lifecycle = lifecycleOwner.lifecycle
    DisposableEffect(key1 = lifecycleOwner) {
        lifecycle.addObserver(getTimer())
        onDispose {
            onPauseTimer()
            lifecycle.removeObserver(getTimer())
        }
    }
    Text(timerText)
}

```

2. Since `CountdownItem()` now expects an `onPauseTimer: () -> Unit` callback function, we must also force our `RepositoriesScreen()` composable to accept such a callback function and then pass it to our `CountdownItem()` composable:

```

@Composable
fun RepositoriesScreen(
    repos: LazyPagingItems<Repository>,
    timerText: String,
    getTimer: () -> CustomCountdown,
    onPauseTimer: () -> Unit
) {
    LazyColumn(...) {
        item {
            CountdownItem(
                timerText,
                getTimer,
                onPauseTimer
            )
        }
        itemsIndexed(repos) { ... }
        [...]
    }
}

```

3. Lastly, inside **MainActivity**, update the **RepositoriesScreen()** composable call to provide an **onPauseTimer()** function implementation, where we will pause the timer by calling the **stop()** method of the **CustomCountdown** instance obtained from the **viewModel** variable through its **timer** field:

```

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            RepositoriesAppTheme {
                [...]
                RepositoriesScreen(lazyRepoItems,
                                    timerText,
                                    getTimer = { viewModel.timer },

```

```
onPauseTimer =  
    { viewModel.timer.stop() }  
)  
}  
}  
}
```

4. Build and run the app. The countdown should have started from 60 again at this point.
5. Before the countdown finishes, remember or write down somewhere the current countdown value and then quickly scroll down past four or five repositories within the list until the countdown is not visible anymore. Make sure to scroll past a few repositories so that Compose removes the node of the timer composable – if you scroll just a bit, the node of the timer won't be removed.
6. Wait for a few seconds and then scroll back up to the top of the list so that the countdown is visible again.

Note that the timer was now paused while the **CountdownItem()** composable was not visible. We have now achieved the desired effect!

But how come the countdown is resumed when the composable becomes visible again? We didn't do anything to cover that case – we only stopped the timer when the **CountdownItem()** composable left composition, but we didn't resume it when it became visible again as it re-entered composition.

Fortunately, the timer is resumed out of the box when the **CountdownItem()** composable re-enters composition – but why is this happening?

This behavior is exhibited because of an interesting side effect provided by the Lifecycle APIs. More precisely, as soon as we're binding the **LifecycleObserver** instance to the **Lifecycle** instance of our

LifecycleOwner, the observer instantly receives as a first event the event corresponding to the current state of **LifecycleOwner**.

Let's have a look inside the **CountdownItem()** composable and see how this could be happening:

```
@Composable
private fun CountdownItem(timerText: String,
    getTimer: () -> CustomCountdown,
    onPauseTimer: () -> Unit) {
    val lifecycleOwner: LifecycleOwner
        = LocalLifecycleOwner.current
    val lifecycle = lifecycleOwner.lifecycle
    DisposableEffect(key1 = lifecycleOwner) {
        lifecycle.addObserver(getTimer())
        onDispose {
            onPauseTimer()
            lifecycle.removeObserver(getTimer())
        }
    }
    Text(timerText)
}
```

In our case, as soon as we're binding the **DefaultLifecycleObserver** instance – that is, **CustomCountdown** – to the **Lifecycle** of the **LifecycleOwner** instance – that is, **MainActivity** – the observer receives as a first event the event corresponding to the current state.

In other words, as soon as our timer composable is visible, we're adding the timer as an observer to the lifecycle of our **MainActivity** class. At that point, the **RESUMED** state is the current state of **MainActivity**, so the **onResume()** callback is triggered inside the **CustomCountdown** component, which effectively resumes the timer countdown in our specific scenario:

```
class CustomCountdown([...]): DefaultLifecycleObserver
{
```

```
var timer: InternalTimer = InternalTimer(...)
override fun onResume(owner: LifecycleOwner) {
    super.onResume(owner)
    if (timer.lastKnownTime > 0) {
        timer.cancel()
        timer = InternalTimer(
            onTick = onTick,
            onFinish = onFinish,
            millisInFuture = timer.lastKnownTime,
            countdownInterval = 1000)
    }
}
override fun onPause(owner: LifecycleOwner) { [...]}
fun stop() { timer.cancel() }
class InternalTimer(...) : CountdownTimer(...) {...}
}
```

We have now made our countdown timer aware of the Compose composition cycles as well.

Summary

In this chapter, we understood what a lifecycle-aware component is and how we can create one.

We first explored how the lifecycle events and states are tied to Android components, such as **Activity** or **Fragment**, and then how predefined components from the **Lifecycle** package can react to them. Then, we created and added a countdown timer component to the Repositories app.

Finally, we prevented users from cheating by making our timer component aware not only of the different lifecycle events and states of **Activity** components but also of the lifecycle of composables.

Further reading

In this chapter, we briefly covered how to create a lifecycle-aware component by making our `CustomCountdown` component aware of the lifecycle events that `MainActivity` exhibits. However, when needed, we can also tap into the lifecycle states of `LifecycleOwner`. To understand how you can do that, check out the official docs for an example:

<https://developer.android.com/topic/libraries/architecture/lifecycle#lco>.

[Support](#) | [Sign Out](#)

©2022 O'REILLY MEDIA, INC. [TERMS OF SERVICE](#) | [PRIVACY POLICY](#)