Chapter 3: Handling Coroutine Cancelations and Exceptions

In the previous chapter, you dove deep into Kotlin coroutines and learned how to use them for asynchronous programming in Android with simple code. You learned how to create coroutines with coroutine builders. Finally, you explored coroutine dispatchers, coroutine scopes, coroutine contexts, and jobs.

Coroutines can be canceled when their purpose has been fulfilled or their job has been done. You can also cancel them based on specific instances in your app, such as when you want users to manually stop a task with a tap of a button. Coroutines do not always succeed and can fail; developers must be able to handle these cases so that the app will not crash, and they can inform the users by displaying a toast or snackbar message.

In this chapter, we will start by understanding coroutine cancelation. You will learn how to cancel coroutines and handle cancelations and timeouts for your coroutines. Then, you will learn how to manage failures and exceptions that can happen in your coroutines.

In this chapter, we will cover the following topics:

- Canceling coroutines
- Managing coroutine timeouts
- Catching exceptions in coroutines

By the end of this chapter, you will understand coroutine cancelations and how you can make your coroutines cancelable. You will be able to add and handle timeouts in your coroutines. You will also know how to add code to catch exceptions in your coroutines.

Technical requirements

You will need to download and install the latest version of Android Studio. You can find the latest version at https://developer.android.com/studio ↗. For an optimal learning experience, a computer with the following specifications is recommended: Intel Core i5 or equivalent or higher, 4 GB RAM minimum, and 4 GB available space.

The code examples for this chapter can be found on GitHub at https://github.com/PacktPublishing/Simplifying-Android-Development-with-Coroutines-and-Flows/tree/main/Chapter03 ↗.

Canceling coroutines

In this section, we will start by looking at coroutine cancelations Developers can cancel coroutines in their projects manually or programmatically. You must make sure your application can handle these cancelations.

If your application is doing a long-running operation that is taking longer than expected and you think it could cause a crash, you might want to stop that task. You can also end tasks that are no longer necessary to free up memory and resources, such as when the user moves out of the activity that launched the task or closes the application. Users can also manually discontinue certain operations if you have that feature in your application. Coroutines make it easier for developers to cancel these tasks.

If you are using **`viewModelScope`** from **`ViewModel`** or **`lifecycleScope`** from the Jetpack Lifecycle Kotlin extension libraries, you can easily create coroutines without manually handling the cancelation. When **`ViewModel`** is cleared, **`viewModelScope`** is automatically canceled, while **`lifecycleScope`** is automatically canceled when the life cycle is destroyed. If you created your own coroutine scope, you must add the cancelation yourself.

In the previous chapter, you learned that using coroutine builders such as **`launch`** returns a **job**. Using this **job** object, you can call the **`cancel()`** function to cancel the coroutine. Take the following example:

```
class MovieViewModel: ViewModel() {

    init {

        viewModelScope.launch {

            val job = launch {

                fetchMovies()

            }

            ...

            job.cancel()

        }

    }

}
```

The **`job.cancel()`** function will cancel the coroutine launched to call the **`fetchMovies()`** function.

After canceling the job, you may want to wait for the cancelation to be finished before continuing to the next task to avoid race conditions. You can do that by calling the **`join`** function after calling the **`call`** function:

```
class MovieViewModel: ViewModel() {

    init {

        viewModelScope.launch() {
            val job = launch {
                fetchMovies()
            }
            ...
            job.cancel()
            job.join()
            hideProgressBar()
        }
    }
}
```

Adding **`job.join()`** here would make the code wait for the job to be canceled before doing the next task, which is **`hideProgressBar()`**.

You can also use the **`Job.cancelAndJoin()`** extension function, which is the same as calling **`cancel`** and then the **`join`** function:

```
class MovieViewModel: ViewModel() {
    init {
        viewModelScope.launch() {
            val job = launch {
                fetchMovies()
            }
            ...
            job.cancelAndJoin()
            hideProgressBar()
        }
    }
}
```

The **cancelAndJoin** function simplifies the call to the **cancel** and **join** functions into a single line of code.

Coroutine jobs can have child coroutine jobs. When you cancel a job, its child jobs (if there are any) will also be canceled, recursively.

If your coroutine scope has multiple coroutines and you need to cancel all of them, you can use the **cancel** function from the coroutine scope instead of canceling the jobs one by one. This will cancel all the coroutines in the scope. Here's an example of using the coroutine scope's **cancel** function to cancel coroutines:

```
class MovieViewModel: ViewModel() {
    private val scope = CoroutineScope(Dispatchers.Main +
      Job())
    init {
        scope.launch {
            val job1 = launch {
                fetchMovies()
            }
            val job2 = launch {
                displayLoadingText()
            }
        }
    }
    override fun onCleared() {
        scope.cancel()
    }
}
```

In this example, when **`scope.cancel()`** is called, it will cancel both the **`job1`** and **`job2`** coroutines, which were created in the coroutine **`scope`** scope.

Using the **`cancel`** function from the coroutine scope makes it easier to cancel multiple jobs launched with the specified scope. However, the coroutine scope won't be able to launch new coroutines after you called the **`cancel`** function on it. If you want to cancel the scope's coroutines but still want to create coroutines from the scope later, you can use **`scope.coroutineContext.cancelChildren()`** instead:

```
class MovieViewModel: ViewModel() {
    private val scope = CoroutineScope(Dispatchers.Main +
      Job())
    init {
        scope.launch() {
            val job1 = launch {
                fetchMovies()

            }



            val job2 = launch {

                displayLoadingText()

            }

        }

    }




    fun cancelAll() {

        scope.coroutineContext.cancelChildren()

    }



    ...



}
```

Calling the **`cancelAll`** function will cancel all the child jobs in the coroutine context of the scope. You will still be able to use the scope later to create coroutines.

Canceling a coroutine will throw **`CancellationException`**, a special exception that indicates the coroutine was canceled. This exception will not crash the application. You will learn more about coroutines and exceptions later in this chapter.

You can also pass a subclass of **`CancellationException`** to the **`cancel`** function to specify a different cause:

```
class MovieViewModel: ViewModel() {

private lateinit var movieJob: Job



    init {

        movieJob = scope.launch() {

            fetchMovies()

        }

    }





    fun stopFetching() {

        movieJob.cancel(CancellationException("Cancelled by

          user"))

    }



    ...
    }
```

This cancels the **`movieJob`** job with **`CancellationException`** containing the message **`Cancelled by user`** as the cause when the user calls the **`stopFetching`** function.

When you cancel a coroutine, the coroutine's job's state will change to **`Cancelling`**. It won't automatically go to the **`Cancelled`** state and cancel the coroutine. The coroutine can continue to run even after the cancelation, unless your coroutine has code that can stop it from running. These states of a job and its life cycle are summarized in the following diagram:

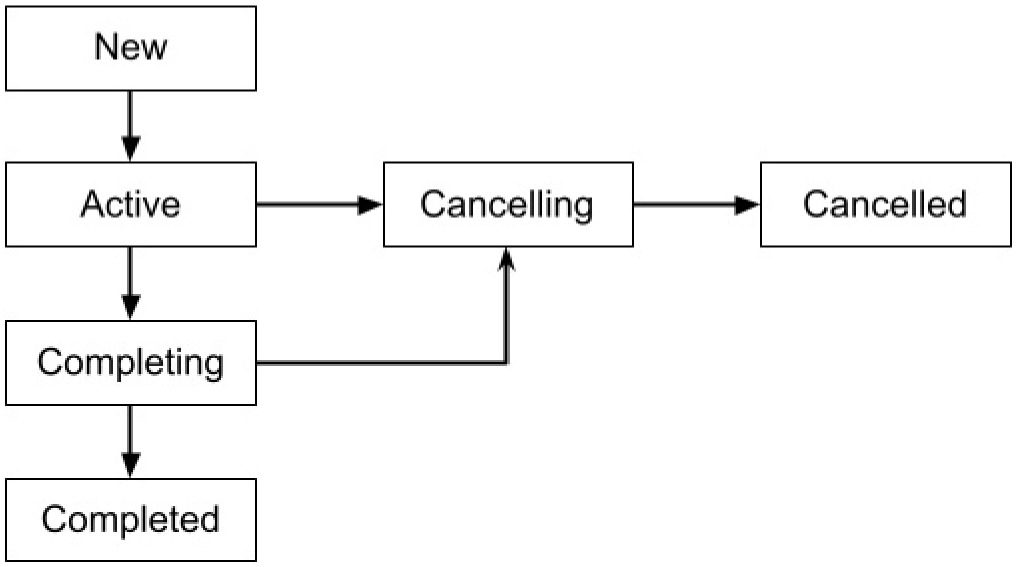

Figure 3.1 – Coroutine job life cycle

Your coroutine code needs to cooperate to be cancelable. The coroutine should handle cancelations as quickly as possible. It must check for cancelations of the coroutine and if the coroutine is already canceled, it throws **`CancellationException`**.

One way to make your coroutine cancelable is to check whether the coroutine job is active (still running or completing) or not by using **`isActive`**. The value of **`isActive`** will become false once the coroutine job changes its state to **Cancelling**, **Cancelled**, or **Completed**. You can make your coroutine cancelable with **isActive** with the following approaches:

- Perform tasks while **isActive** is true.
- Perform tasks only if **isActive** is true.
- Return or throw an exception if **isActive** is false.

Another function you can also use is **Job.ensureActive()**. It will check whether the coroutine job is active, and if it's not, it will throw **CancellationException**.

Here's an example of how you can make your coroutine cancelable with **isActive**:

```
class SensorActivity : AppCompatActivity() {

    private val scope = CoroutineScope(Dispatchers.IO)

    private lateinit var job: Job

   …

    private fun processSensorData() {

        job = scope.launch {

            if (isActive) {

                val data = fetchSensorData()

                saveData(data)

            }

        }

    }
```

```
        fun stopProcessingData() {

            job.cancel()

        }

        ...

    }
```

The coroutine in the **`processSensorData`** function will check whether the job is active and will only proceed with the task if the value of **`isActive`** is true.

Another way to make your coroutine code cancelable is to use suspending functions from the **`kotlinx.coroutines`** package, such as **`yield`** or **`delay`**. The **`yield`** function yields a thread (or a thread pool) of the current coroutine dispatcher to other coroutines to run.

The **`yield`** and **`delay`** functions already check for cancelation and stop the execution or throw **`CancellationException`**. Thus, you no longer need to manually check for cancelation when you are using them in your coroutines. Here's an example using the preceding code snippet, which has been updated with suspending function delay to make the coroutine cancelable:

```
    class SensorActivity : AppCompatActivity() {

        private val scope = CoroutineScope(Dispatchers.IO)

        private lateinit var job: Job

        override fun onCreate(savedInstanceState: Bundle?) {

            ...

            processSensorData()
        }

        private fun processSensorData() {

            job = scope.launch {

                delay (1_000L)

                val data = fetchSensorData()

                saveData(data)

            }

        }

        fun stopProcessingData() {

            job.cancel()

        }

        ...

    }
```

The **`delay`** suspending function will check whether the coroutine job is canceled and will throw **`CancellationException`** if it is, making your coroutine cancelable.

Let's learn how to implement a coroutine cancelation for an Android project in the next section.

Exercise 3.01 – canceling coroutines in an Android app

In this exercise, you will work on an application that uses a coroutine that slowly counts down from 100 to 0 and displays the value on **`TextView`**.

You will then add a button to cancel the coroutine to stop the countdown before it reaches 0:

1. Create a new project in Android Studio. Don't change the suggested name of **`MainActivity`** for the activity.
2. Open the **`app/build.gradle`** file and add the dependency for **`kotlinx-coroutines-android`**:

```
implementation 'org.jetbrains.kotlinx:kotlinx-
  coroutines-android:1.6.0'
```

This will add the **`kotlinx-coroutines-core`** and **`kotlinx-coroutines-android`** libraries to your project, allowing you to use coroutines in your code.

3. Open the **`activity_main.xml`** layout file and add an **`id`** attribute to **`TextView`**:

```
<TextView
    android:id="@+id/textView"
    style="@style/TextAppearance.AppCompat.Large"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    tools:text="0" />
```

The **`id`** attribute will allow you to change the content of this **`TextView`** later.

4. Open the **`MainActivity`** file. Add the following properties to the **`MainActivity`** class:

```
private val scope =
CoroutineScope(Dispatchers.Main)
private?var job: Job? = null
```

```
private lateinit var textView: TextView
private var count = 100
```

5. The first line specifies the scope for the coroutine, **`CoroutineScope`**, with **`Dispatchers.Main`** as the dispatcher. The second line creates a **`job`** property for the coroutine job. The **`textView`** property will be used to display the countdown text and **`count`** initializes the countdown to 100. In the **`onCreate`** function of the **`MainActivity`** file, initialize the value for **`TextView`**:

```
textView = findViewById(R.id.textView)
```

You will update this **`textView`** with the decreasing value of value later.

6. Create a countdown function that will do the counting down of the value:

```
private fun countdown() {
    count--
    textView.text = count.toString()
}
```

This decreases the value of **`count`** by 1 and displays it on the text view.

7. In the **`onCreate`** function, below the **`textView`** initialization, add the following to start the coroutine to count down the value and display it on the text view:

```
job = scope.launch {
    while (count > 0) {
        delay(100)
        countdown()
    }
}
```

This will call the countdown function every **`0.1`** seconds, which will count down and display the value on the text view.

8. Run the application. You will see that it slowly counts down and displays the value from 100 to 0, similar to the following:

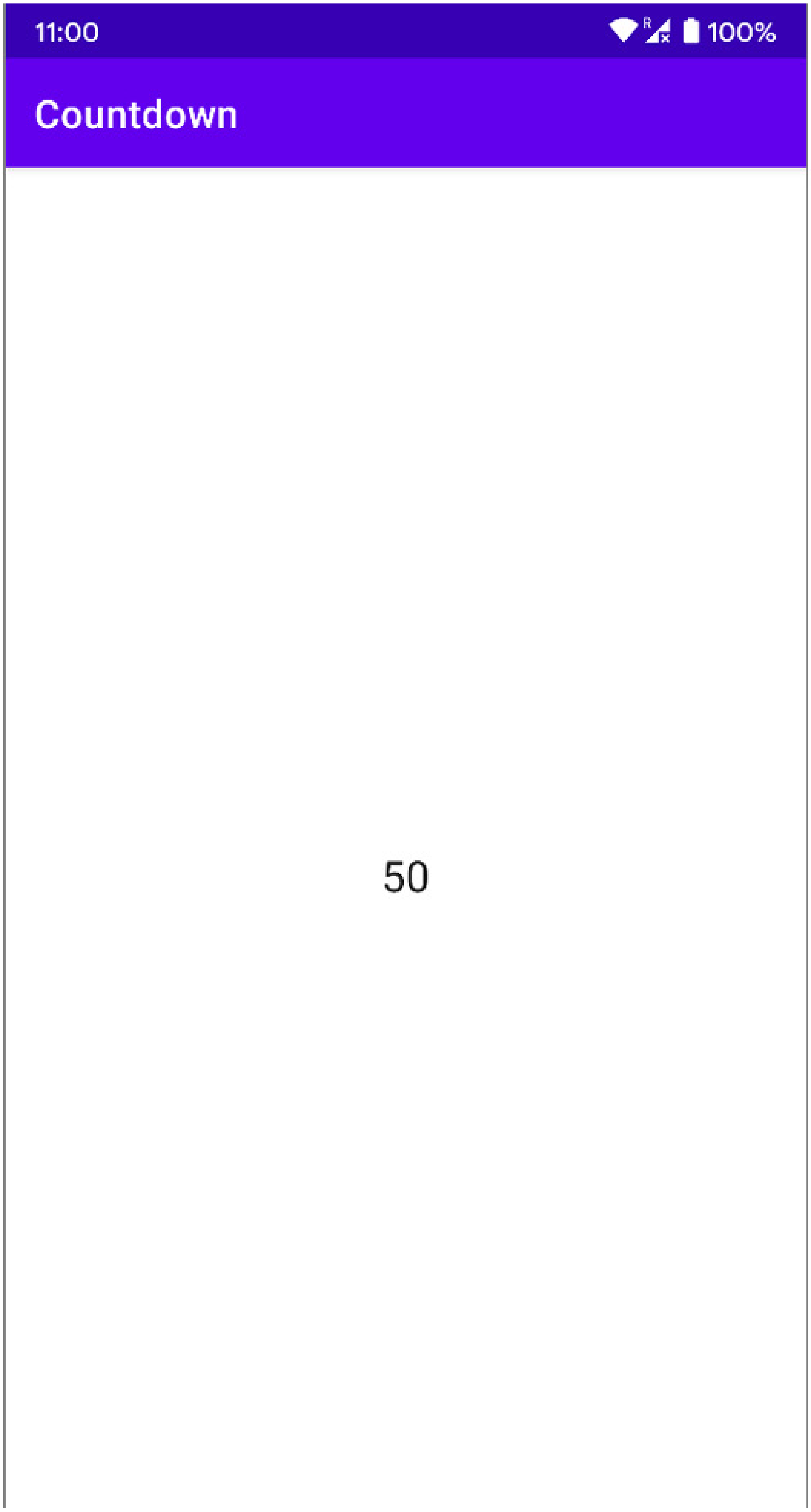
11:00
100%
Countdown
50

Figure 3.2 – The app counting down from 100 to 0

9. Open the **`strings.xml`** file and add a string for the button:

```
<string name="stop">Stop</string>
```

You will use this as the text for the button to stop the countdown.

10. Go to the **`activity_main.xml`** file again and add a button below **`TextView`**:

```
<Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="16dp"
        android:text="@string/stop"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="paren
t"
        app:layout_constraintTop_toBottomOf="@id/te
xtView" />
```

This will add a **`Button`** below **`TextView`**. The button will be used to stop the countdown later.

11. Open **`MainActivity`** and after the job initialization, create a variable for the button:

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

This button, when tapped, will allow the user to stop the countdown.

12. Below that, add a click listener to the button that cancels the job:

```
button.setOnClickListener {
    job?.cancel()
    }
```

When you click the button, it will cancel the coroutine.

13. Run the application again. Tap on the **STOP** button and notice that the counting down stops, as shown in the following figure:

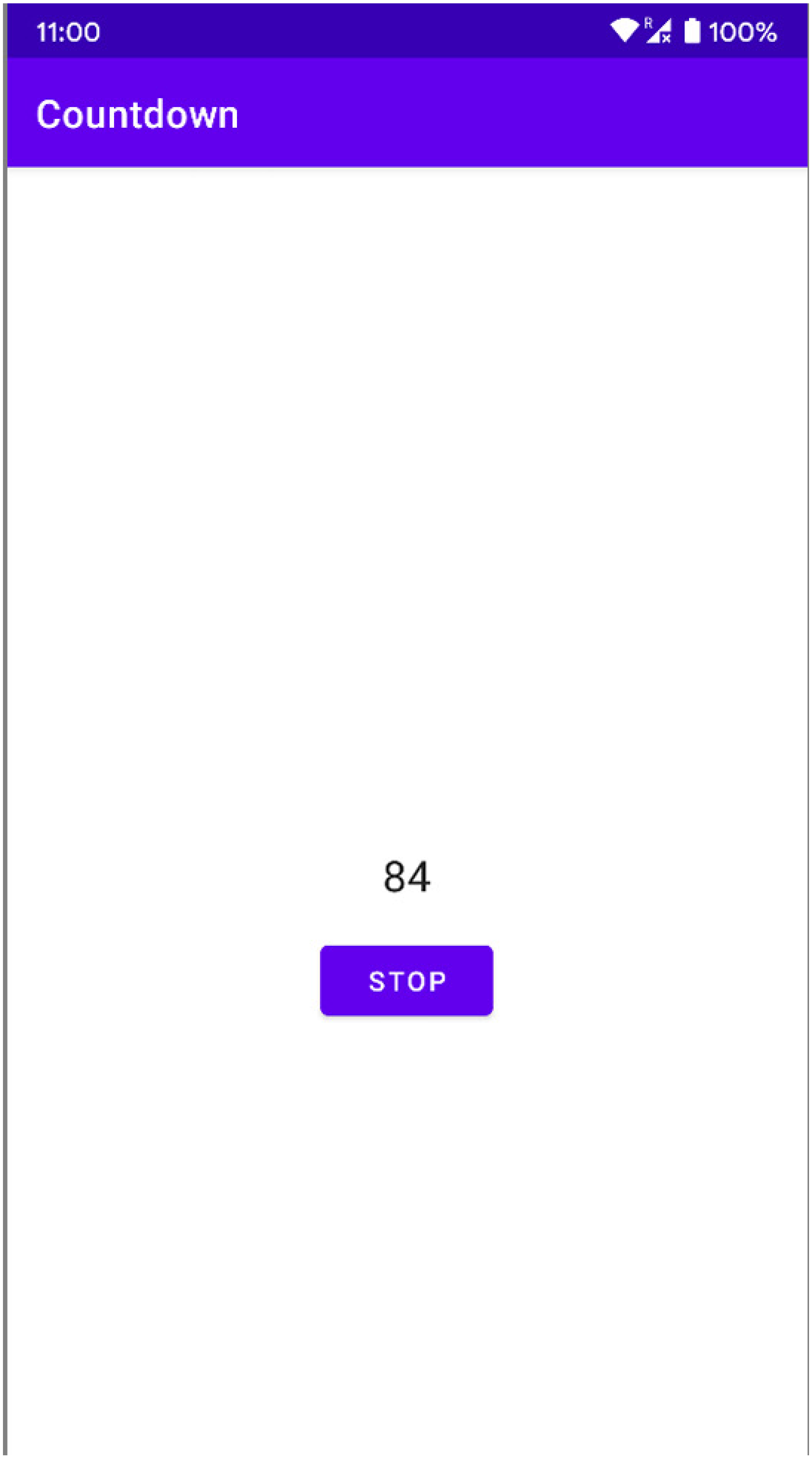
11:00
100%
Countdown
84
STOP

Figure 3.3 – Clicking the STOP button cancels the coroutine

Tapping on the **STOP** button cancels the coroutine with the **`job.cancel()`** call. This works because the coroutine is using the suspending **`delay`** function, which checks whether the coroutine is active or not.

In this exercise, you have worked on adding code to cancel a running coroutine in an Android app by tapping on a button.

There may be instances where you want to continue work even if you have canceled the job. To ensure the tasks will be done even if the coroutine is canceled, you can use **`withContext(NonCancellable)`** on the task.

In this section, you learned how to cancel coroutines and how to make sure your coroutine code is cancelable. You will learn how to handle coroutine timeouts in the next section.

Managing coroutine timeouts

In this section, you will learn about timeouts and how you can cancel your coroutines with a timeout. Setting a fixed amount of time after which to stop asynchronous code that is running longer than expected can help you save resources and immediately notify users of any issues.

When your application is doing a background task, you may want to stop it because it is taking too long. You can manually track the time and cancel the task. Or you can use the **`withTimeout`** suspending function. With the **`withTimeout`** function, you can set your timeout in milliseconds or **`Duration`**. Once this timeout is exceeded, it will throw **`TimeOutCancellationException`**, a subclass of **`CancellationException`**. Here's an example of how you can use **`withTimeout`**:

```
class MovieViewModel: ViewModel() {

    init {

        viewModelScope.launch {

            val job = launch {

                withTimeout(5_000L) {

                    fetchMovies()

                }

            }

            ...

        }

    }

}
```

A timeout of 5,000 milliseconds (5 seconds) has been set for the coroutine. If the **`fetchMovies`** task takes longer than that, the coroutine will time out and throw **`TimeoutCancellationException`**.

Another function you can use is **`withTimeoutOrNull`**. It is similar to the **`withTimeout`** function, but it will return null if the timeout was exceeded. Here's an example of how you can use **`withTimeoutOrNull`**:

```
class MovieViewModel: ViewModel() {

    init {

        viewModelScope.launch() {

            val job = async {
                fetchMovies()
            }
            val movies = withTimeoutOrNull(5_000L) {
                job.await()
            }
            ...
        }
    }
    ...
}
```

The coroutine will return null if **`fetchMovies`** times out after 5 seconds, and if not, it will return the list of movies fetched.

As you learned in the previous section, the coroutine must be cancelable so that it will be canceled after the timeout. In the next section, you will learn how to handle the cancelation exception from coroutines.

In this section, you have learned about coroutine timeouts and how you can set an amount of time after which to automatically cancel a coroutine.

Catching exceptions in coroutines

In this section, you will learn about coroutine exceptions and how to handle them in your application. As it is always possible that your coroutines will fail, it is important to learn how to catch exceptions so that you can avoid crashes and notify your users.

To handle exceptions in your coroutines, you can simply use **`try-catch`**. For example, if you have a coroutine started with a **`launch`** coroutine builder, you can do the following to handle exceptions:

```
class MovieViewModel: ViewModel() {

    init {

        viewModelScope.launch() {

            try {

                fetchMovies()

            } catch (exception: Exception) {

                Log.e("MovieViewModel",

                  exception.message.toString())

            }

        }

    }

    ...

}
```

If **`fetchMovies`** has an exception, **`ViewModel`** will write the exception message to the logs.

If your coroutine was built using the **`async`** coroutine builder, the exception will be thrown when you call the **`await`** function on the **`Deferred`** object. Your code to handle the exception would look like the following:

```
class MovieViewModel: ViewModel() {
      init {
          viewModelScope.launch() {
              val job = async {
                  fetchMovies()
              }
              var movies = emptyList<Movie>()
              try {
                  movies = job.await()
              } catch (exception: Exception) {
                  Log.e("MovieViewModel",
                    exception.message.toString())
              }
          }
      }
      ...
  }
```

If an exception is encountered while the **`fetchMovies`** call is running, the movies list will be an empty list of movies, and **`ViewModel`** will write the exception message to the logs.

When a coroutine encounters an exception, it will cancel the job and pass on the exception to its parent. This parent coroutine will be canceled, as well as its children. Exceptions in the child coroutines will not affect the parent and its sibling coroutines if you use **`SupervisorJob`** as follows:

- Creating the coroutine scope with the suspending **`supervisorScope{}`** builder
- Using **`SupervisorJob`** for your coroutine scope: **`CoroutineScope(SupervisorJob())`**

If the exception of your coroutine is a subclass of **`CancellationException`**, for example, **`TimeoutCancellationException`** or a custom one you pass to the **`cancel`** function, the exception will not be transmitted to the parent.

When handling coroutine exceptions, you can also use a single place to handle these exceptions with **`CoroutineExceptionHandler`**. **`CoroutineExceptionHandler`** is a coroutine context element that you can add to your coroutine to handle uncaught exceptions. The following lines of code show how you can use it:

```
class MovieViewModel: ViewModel() {

    private val exceptionHandler =

      CoroutineExceptionHandler { _, exception ->

        Log.e("MovieViewModel",

          exception.message.toString())

    }



    private val scope = CoroutineScope(exceptionHandler)

    ...

}
```

The exceptions from the coroutines started from the scope will be handled by **exceptionHandler**, if it's not handled wherever an error could occur, which will write the exception message to the logs.

Let's try to add code to handle exceptions in your coroutines.

Exercise 3.02 – catching exceptions in your coroutines

In this exercise, you will continue working on the application that displays on **TextView** a number from 100 and slowly decreases it down to 0. You will be adding code to handle exceptions in the coroutine:

1. Open the countdown app you built in the previous exercise.
2. Go the **MainActivity** file and at the end of the countdown function, add the following to simulate an exception:
```
if ((0..9).random() == 0) throw Exception("An error
  occurred")
```

This will generate a random number from 0 to 9 and if it's 0, it will throw an exception. It will simulate the coroutine encountering an exception.

3. Run the application. It will start to count down and some point later, it will throw the exception and crash the app.
4. Surround the code in your coroutine with a **try-catch** block to catch the exception in the app:
```
job = scope.launch {
    try {
        while (count > 0) {
            delay(100)
            countdown()
        }
    } catch (exception: Exception) {
        //TODO
    }
    }
```

This will catch the exception from the countdown function. The app will no longer crash but you will need to inform the user about the exception.

5. Inside the **`catch`** block, replace **`//TODO`** with **`Snackbar`** to display the exception message:
   ```
   Snackbar.make(textView,
   exception.message.toString(),
     Snackbar.LENGTH_LONG).show()
   ```

This will display a snackbar message with the text **`An error occurred`**, which is the message of the exception.

6. Run the application again. It will start to count down but instead of crashing, a snackbar message will be displayed, as shown in the following figure:

Figure 3.4 – Snackbar displayed when the coroutine has encountered the exception

In this exercise, you updated your application so that it can handle exceptions in the coroutines instead of crashing.

In this section, you have learned about coroutine exceptions and how you can catch them in your Android apps.

Summary

In this chapter, you learned about coroutine cancelations. You can cancel coroutines by using the `cancel` or `cancelAndJoin` function from the coroutine job or the `cancel` function from the coroutine scope.

You learned that a coroutine cancelation needs to be cooperative. You also learned how you can change your code to make your coroutine cancelable by using `isActive` checks or by using suspending functions from the `kotlinx.coroutines` package.

Then, you learned about coroutine timeouts. You can set a timeout (in milliseconds or `Duration`) using `withTimeout` or `withTimeoutOrNull`.

You also learned about coroutine exceptions and how to catch them. `try-catch` blocks can be used to handle exceptions. You can also use `CoroutineExceptionHandler` in your coroutine scope to catch and handle exceptions in a single location.

Finally, you worked on an exercise to add cancelation to a coroutine and another exercise to update your code to handle coroutine exceptions.

In the next chapter, you will dive into creating and running tests for the coroutines in your Android projects.