

Chapter 1: Introduction to Asynchronous Programming in Android

There are Android applications that work on their own. But most apps retrieve data from or send data to a local database or a backend server. Examples of these include fetching posts from a social network, saving your favorites from a list, uploading an image, or updating your profile information. These tasks and other resource-intensive computations may happen instantly or take a while to finish. Factors such as internet connection, device specifications, and server settings affect how long these operations take.

Long-running operations must not be performed on the main UI thread as the application will be blocked until they are completed. The application might become unresponsive to the users. Users may not be aware of what's happening, and this might prompt them to close the app and reopen it (canceling the original task or doing it again). The app can also suddenly crash. Some users might even stop using your app if this happens frequently.

To prevent this from happening, you need to use asynchronous programming. Tasks that can take an indefinite amount of time must be done asynchronously. They must run in the background, parallel to other tasks. For example, while posting information to your backend server, the app displays the UI, which the users can interact with. When the operation finishes, you can then update the UI or notify the users (with a dialog or a snackbar message).

With this book, you will learn how to simplify asynchronous programming in Android using Kotlin coroutines and flows.


In this chapter, you will first start by revisiting the concept of asynchronous programming. After that, you will look into the various ways it is being done now in Android and how they may no longer be the best way moving forward. Then, you will be introduced to the new, recommended way of performing asynchronous programming in Android: coroutines and flows.


This chapter covers three main topics:

- Asynchronous programming
- Threads, AsyncTasks, and **Executors**
- The new way to do it – coroutines and flows

By the end of this chapter, you will have a basic understanding of asynchronous programming, and know how to do it in Android using threads, AsyncTasks, and **Executors**. Finally, you will discover Kotlin coroutines and flows as these are the recommended ways of doing asynchronous programming in Android.

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 book can be found on GitHub at <https://github.com/PacktPublishing/Simplifying-Android-Development-with-Coroutines-and-Flows> .

Understanding asynchronous programming

In this section, we will start by looking at asynchronous programming. Asynchronous programming is a programming method that allows work to be done independently of the main application thread.

A normal program will run sequentially. It will perform one task and move to the next task after the previous one has finished. For simple operations, this is fine. However, there are some tasks that might take a long time to finish, such as the following:

- Fetching data from or saving data to a database
- Getting, adding, or updating data to a network
- Processing text, images, videos, or other files
- Complicated computations

The app will look frozen and unresponsive to the users while it is performing these tasks. They won't be able to do anything else in the app until the tasks are finished.

Asynchronous programming solves this problem. You can run a task that may be processed indefinitely on a background thread (in parallel to the main thread) without freezing the app. This will allow the users to still interact with the app or the UI while the original task is running. When the task has finished or if an error was encountered, you can then inform the user using the main thread.

A visual representation of asynchronous programming is shown in the following figure:

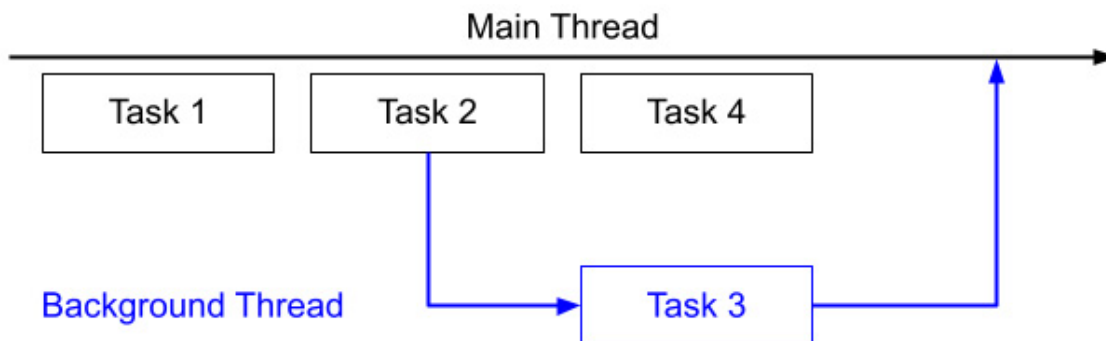


Figure 1.1 – Asynchronous programming

Task 1 and **Task 2** are running on the main thread. **Task 2** starts **Task 3** on the background thread. While **Task 3** is running, the main thread can continue to perform other tasks, such as **Task 4**. After **Task 3** is done, it will return to the main thread.

Asynchronous programming is an important skill for developers to have, especially for mobile app development. Mobile devices have limited capabilities and not all locations have a stable network connection.

In Android, if you run a task on the main thread and it takes too long, the app can become unresponsive or look frozen. The app can also crash unexpectedly. You will likely get an **Application Not Responding (ANR)** error, as shown in the following screenshot:

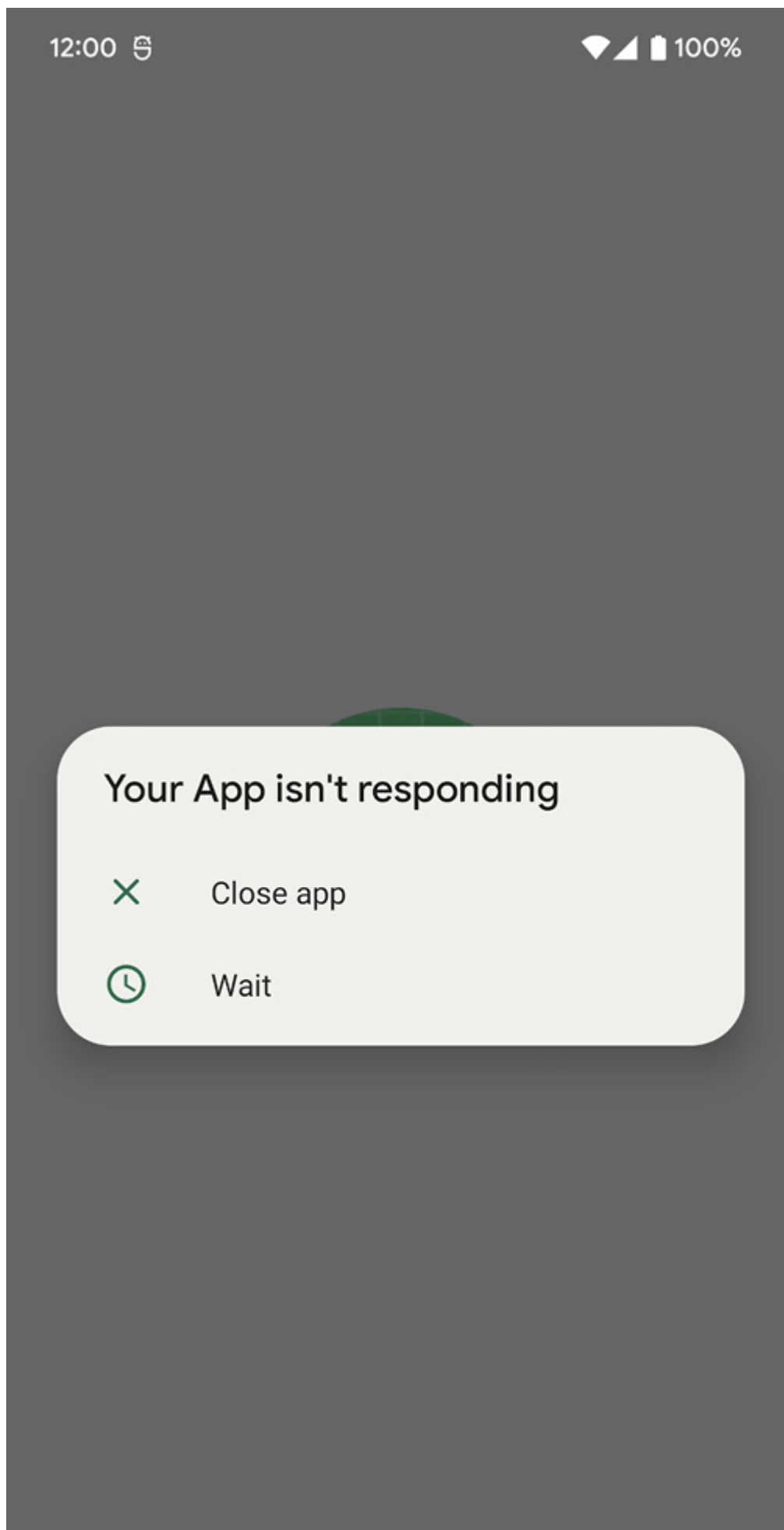




Figure 1.2 – An ANR dialog

Starting with Android 3.0 (Honeycomb), running a network operation on the main thread will cause `android.os.NetworkOnMainThreadException`, which will crash your app.

ANR dialogs and crashes can annoy your users. If they happen all the time, they might stop using your app altogether and choose another app. To prevent them in your app, you must run tasks that can take a long period of time on the background thread.

In this section, you revisited the concept of asynchronous programming and how you can use it to run long-running tasks without freezing the app. You will explore various approaches for using asynchronous programming in Android in the next section.

Exploring threads, AsyncTasks, and Executors

There are many ways you can run tasks on the background thread in Android. In this section, you are going to explore various ways of doing asynchronous programming in Android, including using threads, `AsyncTask`, and **Executors**. You will learn how to start a task on the background thread and then update the main thread with the result.

Threads

A thread is a unit of execution that runs code concurrently. In Android, the UI thread is the main thread. You can perform a task on another

thread by using the **java.lang.Thread** class:

```
private fun fetchTextWithThread() {  
  
    Thread {  
  
        // get text from network  
  
        val text = getTextFromNetwork()  
  
    }.start()  
  
}
```

To run the thread, call **Thread.start()**. Everything that is inside the braces will be performed on another thread. You can do any operation here, except updating the UI, as you will encounter **NetworkOnMainThreadException**.

To update the UI, such as displaying the text fetched in a **TextView** from the network, you would need to use **Activity.runOnUiThread()**. The code inside **runOnUiThread** will be executed in the main thread, as follows:

```
private fun fetchTextWithThread() {  
  
    Thread {  
  
        // get text from network  
  
        val text = getTextFromNetwork()  
  
        runOnUiThread {  
  
            // Display on UI  
  
            displayText(text)  
  
        }  
  
    }  
  
}
```

```
    }.start()

}
```

runOnUiThread will perform the **displayText(text)** function on the main UI thread.

If you are not starting the thread from an activity, you can use handlers instead of **runOnUiThread** to update the UI, as seen in *Figure 1.3*:

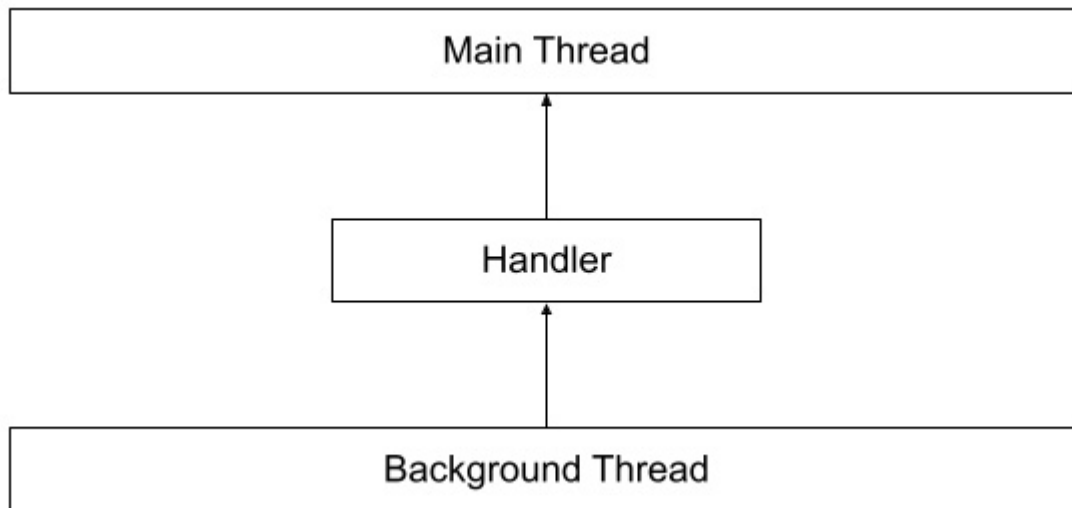


Figure 1.3 – Threads and a handler

A handler (**android.os.Handler**) allows you to communicate between threads, such as from the background thread to the main thread, as shown in the preceding figure. You can pass a looper into the Handler constructor to specify the thread where the task will be run. A looper is an object that runs the messages in the thread's queue.

To attach the handler to the main thread, you should use **Looper.getMainLooper()**, like in the following example:

```
private fun fetchTextWithThreadAndHandler() {

    Thread {

        // get text from network
```



```

        val text = getTextFromNetwork()

        Handler(Looper.getMainLooper()).post {

            // Display on UI

            displayText(text)

        }

    }.start()

}

```

Handler(Looper.getMainLooper()) creates a handler tied to the main thread and posts the **displayText()** runnable function on the main thread.

The **Handler.post (Runnable)** function enqueues the runnable function to be executed on the specified thread. Other variants of the post function include **postAtTime(Runnable)** and **postDelayed (Runnable, uptimeMillis)**.

Alternatively, you can also send an **android.os.Message** object with your handler, as shown in *Figure 1.4*:

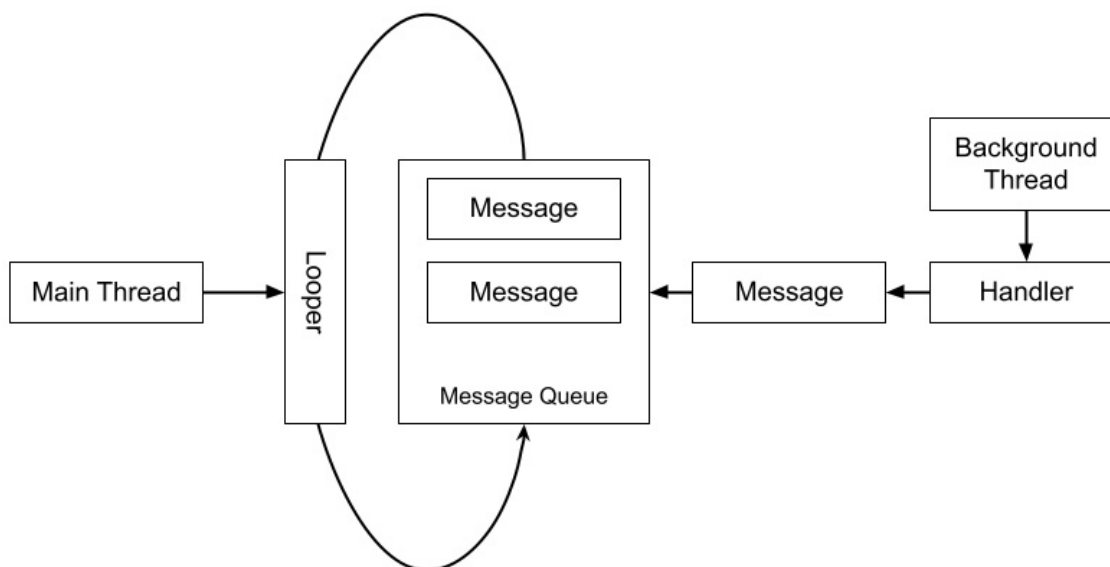


Figure 1.4 – Threads, handlers, and messages

A thread's handler allows you to send a message to the thread's message queue. The handler's loop will execute the messages in the queue.

To include the actual messages you want to send in your `Message` object, you can use `setData(Bundle)` to pass a single bundle of data. You can also use the public fields of the message class (`arg1`, `arg2`, and `what` for integer values, and `obj` for an object value).

You must then create a subclass of `Handler` and override the `handleMessage(Message)` function. There, you can then get the data from the message and process it in the handler's thread.

You can use the following functions to send a message:

`sendMessage(Message)`, `sendMessageAtTime(Message, uptimeMillis)`, and `sendMessageDelayed(Message, delayMillis)`. The following code shows the use of the `sendMessage` function to send a message with a data bundle:

```
private val key = "key"

private val messageHandler = object :

    Handler(Looper.getMainLooper()) {

        override fun handleMessage(message: Message) {

            val bundle = message.data

            val text = bundle.getString(key, "")

            //Display text

            displayText(text)

        }

    }

private fun fetchTextWithHandlerMessage() {
```

```
Thread {  
  
    // get text from network  
  
    val text = getTextFromNetwork()  
  
    val message = handler.obtainMessage()  
  
    val bundle = Bundle()  
  
    bundle.putString(key, text)  
  
    message.data = bundle  
  
    mHandler.sendMessage(message)  
  
}.start()  
  
}
```

Here, **fetchTextWithHandlerMessage()** gets the text from the network in a background thread. It then creates a message with a bundle object containing a string with a key of **key** to send that text. The handler can then, through the **handleMessage()** function, get the message's bundle and get the string from the bundle using the same key.

You can also send empty messages with an integer value (the what) that you can use in your **handleMessage** function to identify what message was received. These send empty functions are **sendEmptyMessage(int)**, **sendEmptyMessageAtTime(int, long)**, and **sendEmptyMessageDelayed(int, long)**.

This example uses **0** and **1** as values for what ("what" is a field of the **Message** class that is a user-defined message code so that the recipient can identify what this message is about): **1** for the case when the background task succeeded and **0** for the failure case:

```
private val emptymessageHandler = object :  
  
    Handler(Looper.getMainLooper()) {  
  
        override fun handleMessage(message: Message) {  
  
            if (message.what == 1) {  
  
                //Update UI  
  
            } else {  
  
                //Show Error  
  
            }  
  
        }  
  
    }  
  
}   
  
private fun fetchTextWithEmptyMessage() {  
  
    Thread {  
  
        // get text from network  
  
        ...  
  
        if (failed) {  
  
            emptymessageHandler.sendEmptyMessage(0)  
  
        } else {  
  
            emptymessageHandler.sendEmptyMessage(1)  
  
        }  
  
    }.start()
```

```
}
```

In the preceding code snippet, the background thread fetches the text from the network. It then sends an empty message of **1** if the operation succeeded and **0** if not. The handler, through the `handleMessage()` function, gets the `what` integer value of the message, which corresponds to the **0** or **1** empty message. Depending on this value, it can either update the UI or show an error to the main thread.

Using threads and handlers works for background processing, but they have the following disadvantages:

- Every time you need to run a task in the background, you should create a new thread and use `runOnUiThread` or a new handler to post back to the main thread.
- Creating threads can consume a lot of memory and resources.
- It can also slow down your app.
- Multiple threads make your code harder to debug and test.
- Code can become complicated to read and maintain.

Using threads makes it difficult to handle exceptions, which can lead to crashes.

As a thread is a low-level API for asynchronous programming, it is better to use the ones that are built on top of threads, such as executors and, until it was deprecated, `AsyncTask`. You can avoid it altogether by using Kotlin coroutines, which you will learn more about later in this chapter.

In the next section, you will explore callbacks, another approach to asynchronous Android programming.

Callbacks

Another common approach to asynchronous programming in Android is using callbacks. A callback is a function that will be run when the asyn-

chronous code has finished executing. Some libraries offer callback functions that developers can use in their projects.

The following is a simple example of a callback:

```
private fun fetchTextWithCallback() {  
  
    fetchTextWithCallback { text ->  
  
        //display text  
  
        displayText(text)  
  
    }  
  
}  
  
fun fetchTextWithCallback(onSuccess: (String) -> Unit) {  
  
    Thread {  
  
        val text = getTextFromNetwork()  
  
        onSuccess(text)  
  
    }.start()  
  
}
```

In the preceding example, after fetching the text in the background, the **onSuccess** callback will be called and will display the text on the UI thread.

Callbacks work fine for simple asynchronous tasks. They can, however, become complicated easily, especially when nesting callback functions and handling errors. This makes it hard to read and test. You can avoid this by avoiding nesting callbacks and splitting functions into subfunc-

tions. It is better to use coroutines, which you will learn more about shortly in this chapter.

AsyncTask

AsyncTask has been the go-to class for running background tasks in Android. It makes it easier to do background processing and post data to the main thread. With **AsyncTask**, you don't have to manually handle threads.

To use **AsyncTask**, you have to create a subclass of it with three generic types:

```
AsyncTask<Params?, Progress?, Result?>()
```

These types are as follows:

- **Params:** This is the type of input for **AsyncTask** or is void if there's no input needed.
- **Progress:** This argument is used to specify the progress of the background operation or Void if there's no need to track the progress.
- **Result:** This is the type of output of **AsyncTask** or is void if there's no output to be displayed.

For example, if you are going to create **AsyncTask** to download text from a specific endpoint, your **Params** will be the URL (**String**) and **Result** will be the text output (**String**). If you want to track the percentage of time remaining to download the text, you can use **Integer** for **Progress**. Your class declaration would look like this:

```
class DownloadTextAsyncTask : AsyncTask<String, Integer,  
String>()
```

You can then start **AsyncTask** with the following code:

```
DownloadTextAsyncTask().execute("https://example.com")
```

AsyncTask has four events that you can override for your background processing:

- **doInBackground**: This event specifies the actual task that will be run in the background, such as fetching/saving data to a remote server. This is the only event that you are required to override.
- **onPostExecute**: This event specifies the tasks that will be run in the UI thread after the background operation finishes, such as displaying the result.
- **onPreExecute**: This event runs on the UI thread before doing the actual task, usually displaying a progress loading indicator.
- **onProgressUpdate**: This event runs in the UI thread to denote progress on the background process, such as displaying the amount of time remaining to finish the task.

The diagram in *Figure 1.5* visualizes these **AsyncTask** events and in what threads they are run:

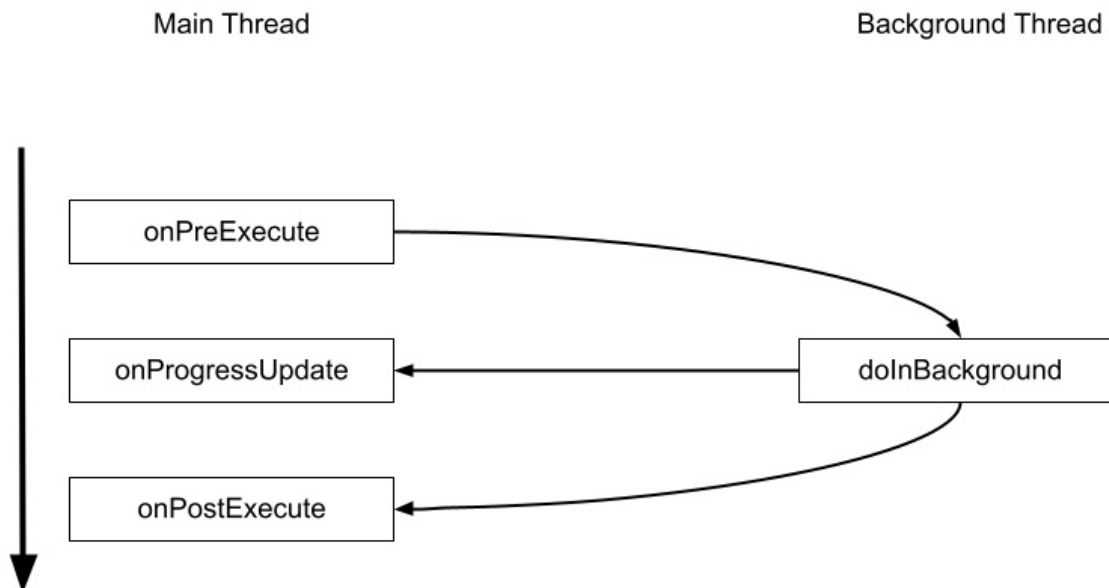


Figure 1.5 – AsyncTask events in main and background threads

The **onPreExecute**, **onProgressUpdate**, and **onPostExecute** functions will run on the main thread, while **doInBackground** executes on the background

thread.

Coming back to our example, your **DownloadTextAsync** class could look like the following:

```
class DownloadTextAsyncTask : AsyncTask<String, Void,  
    String>() {  
  
    override fun doInBackground(vararg params:  
        String?): String? {  
  
        val text = getTextFromNetwork(params[0] ?: "")  
  
        //get text from network  
  
        return text  
  
    }  
  
    override fun onPostExecute(result: String?) {  
  
        //Display on UI  
  
    }  
  
}
```

In **DownloadTextAsync**, **doInBackground** fetches the text from the network and returns it as a string. **onPostExecute** will then be called with that string that can be displayed in the UI thread.

AsyncTask can cause context leaks, missed callbacks, or crashes on configuration changes. For example, if you rotate the screen, the activity will be recreated and another **AsyncTask** instance can be created. The original instance won't be automatically canceled and when it finishes and returns to **onPostExecute()**, the original activity is already gone.

Using **AsyncTask** also makes your code more complicated and less readable. As of Android 11, **AsyncTask** has been deprecated. It is recommended to use **java.util.concurrent** or Kotlin coroutines instead.

In the next section, you will explore one of the **java.util.concurrent** classes for asynchronous programming, **Executors**.

Executors

One of the classes in the **java.util.concurrent** package that you can use for asynchronous programming is **java.util.concurrent.Executor**. An executor is a high-level Java API for managing threads. It is an interface that has a single function, **execute(Runnable)**, for performing tasks.

To create an executor, you can use the utility methods from the **java.util.concurrent.Executors** class.

Executors.newSingleThreadExecutor() creates an executor with a single thread.

Your asynchronous code with **Executor** will look like the following:

```
val handler = Handler(Looper.getMainLooper())

private fun fetchTextWithExecutor() {

    val executor = Executors.newSingleThreadExecutor()

    executor.execute {

        // get text from network

        val text = getTextFromNetwork()

        handler.post {

            // Display on UI
```

```
}  
  
}  
  
}
```

The handler with `Looper.getMainLooper()` allows you to communicate back to the main thread so you can update the UI after your background task has been done.

ExecutorService is an executor that can do more than just `execute(Runnable)`. One of its subclasses is **ThreadPoolExecutor**, an **ExecutorService** class that implements a thread pool that you can customize.

ExecutorService has `submit(Runnable)` and `submit(Callable)` functions, which can execute a background task. They both return a **Future** object that represents the result.

The **Future** object has two functions you can use, `Future.isDone()` to check whether the executor has finished the task and `Future.get()` to get the results of the task, as follows:

```
val handler = Handler(Looper.getMainLooper())  
  
private fun fetchTextWithExecutorService() {  
  
    val executor = Executors.newSingleThreadExecutor()  
  
    val future = executor.submit {  
  
        displayText(getTextFromNetwork())  
  
    }  
  
    ...  
  
    val result = future.get()
```

```
}
```

In the preceding code, the executor created with a new single thread executor was used to submit the runnable function to get and display text from the network. The **submit** function returns a **Future** object, which you can later use to fetch the result with **Future.get()**.

In this section, you learned some of the methods that you can use for asynchronous programming in Android. While they do work and you can still use them (except for the now-deprecated **AsyncTask**), nowadays, they are not the best method to use moving forward.

In the next section, you will learn the new, recommended way of asynchronous programming in Android: using Kotlin coroutines and flows.

The new way to do it – coroutines and flows

In this section, you will learn about the recommended approach for Android asynchronous programming: using coroutines and flows. Coroutines is a Kotlin library you can use in Android to perform asynchronous tasks. Coroutines is a library for managing background tasks that return a single value. Flows are built on top of coroutines that can return multiple values.

Kotlin coroutines

Coroutines is a Kotlin library for managing background tasks, such as making network calls and accessing files or databases, or performing long-running background tasks. Using Kotlin coroutines is Google's official recommendation for asynchronous programming on Android. Their Android Jetpack libraries, such as Lifecycle, WorkManager, and Room-KTX, now include support for coroutines. Other Android libraries, such as Retrofit, Ktor, and Coil, provide first-class support for Kotlin coroutines.

With Kotlin coroutines, you can write your code in a sequential way. A long-running task can be made into a **suspend** function. A **suspend** function is a function that can perform its task by suspending the thread without blocking it, so the thread can still run other tasks. When the suspending function is done, the current thread will resume execution. This makes the code easier to read, debug, and test. Coroutines follow a principle of structured concurrency.

You can add coroutines to your Android project by adding the following lines to your **app/build.gradle** file dependencies:

```
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-  
core:1.6.0"  
  
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-  
android:1.6.0"
```

kotlinx-coroutines-core is the main library for Kotlin coroutines, while **kotlinx-coroutines-android** adds support for the main Android thread (**Dispatchers.Main**).

To mark a function as a suspending function, you can add the **suspend** keyword to it; for example, here we have a function that calls the **fetchText()** function, which retrieves text from an endpoint and then displays it in the UI thread:

```
fun fetchText(): String {  
    ...  
}
```

You can make the **fetchText()** function a suspending function by prefixing the **suspend** keyword, as follows:

```
suspend fun fetchText(): String { ... }
```

Then, you can create a coroutine that will call the **fetchText()** suspending function and display the list, as follows:

```
lifecycleScope.launch(Dispatchers.IO) {  
  
    val fetchedText = fetchText()  
  
    withContext(Dispatchers.Main) {  
  
        displayText(fetchedText)  
  
    }  
  
}
```

lifecycleScope is the scope with which the coroutine will run. **launch** creates a coroutine to run in **Dispatchers.IO**, which is a thread for I/O or network operations.

The **fetchText()** function will suspend the coroutine before it starts the network request. While the coroutine is suspended, the main thread can do other work.

After getting the text, it will resume the coroutine.

withContext(Dispatchers.Main) will switch the coroutine context to the main thread, where the **displayText(text)** function will be executed (**Dispatchers.Main**).

In Android Studio, the **Editor** window identifies the **suspend** function calls in your code with a gutter icon next to the line number. As shown in lines 13 and 15 in *Figure 1.6*, the **fetchText()** and **withContext()** lines have the **suspend** function call gutter icon:



Figure 1.6 – Android Studio suspend function call gutter icon

You can learn more about Kotlin coroutines in [Chapter 2, Understanding Kotlin Coroutines](#).

In the next section, you will learn about Kotlin Flows, built on top of coroutines, which can return multiple sequences of values.

Kotlin Flows

Flow is a new Kotlin asynchronous stream library that is built on top of Kotlin coroutines. A flow can emit multiple values instead of a single value and over a period of time. Kotlin Flow is ideal to use when you need to return multiple values asynchronously, such as automatic updates from your data source.

Flow is now used in Jetpack libraries such as Room-KTX and Android developers are already using Flow in their applications.

To use Kotlin Flows in your Android project, you have to add coroutines. An easy way to create a flow of objects is to use the `flow{} builder`. With the `flow{} builder` function, you can add values to the stream by calling `emit`.

Let's say in your Android app you have a `getTextFromNetwork` function that fetches text from a network endpoint and returns it as a `String` object:

```
fun getTextFromNetwork(): String { ... }
```

If we want to create a flow of each word of the text, we can do it with the following code:

```
private fun getWords(): Flow<String> = flow {  
  
    getTextFromNetwork().split(" ").forEach {  
  
        delay(1_000)  
  
        emit(it)  
  
    }  
  
}
```

Flow does not run or emit values until the flow is collected with any terminal operators, such as `collect`, `launchIn`, or `single`. You can use the `collect()` function to start the flow and process each value, as follows:

```
private suspend fun displayWords() {  
  
    getWords().collect {  
  
        Log.d("flow", it)  
  
    }  
  
}
```



```
}
```

A visual representation of this flow is shown in the following figure:

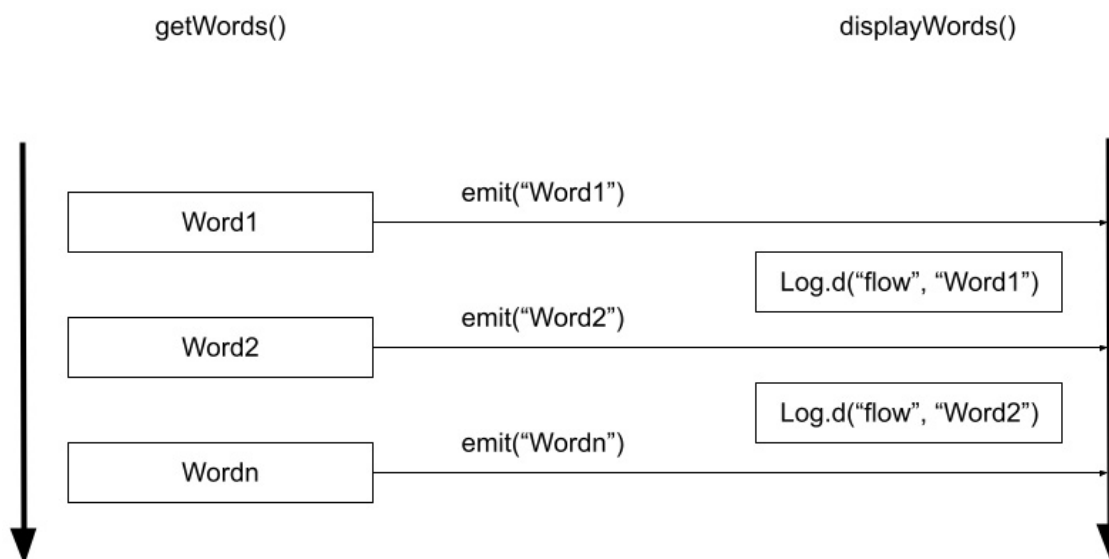


Figure 1.7 – Kotlin Flow visual representation

As you can see in *Figure 1.7*, as soon as the **getWords()** flow emits a string, the **displayWords** function collects the string and displays it immediately on the logs.

You will learn more about Kotlin Flows in [Chapter 5, Using Kotlin Flows](#).

In this section, you learned about Kotlin coroutines and flows, the recommended way to carry out asynchronous programming in Android.

Coroutines is a Kotlin library for managing long-running tasks in the background. Flow is a new Kotlin asynchronous stream library, built on top of coroutines, that can emit multiple values over a period of time.

Summary

In this chapter, you revisited the concept of asynchronous programming. We learned that asynchronous programming helps you execute long-running tasks in the background without freezing the app and annoying your users.

You then learned about various ways you can do asynchronous programming in Android, including with threads, AsyncTask, and **Executors**. We also learned that they allow you to perform tasks in the background and update the main thread. AsyncTask is already deprecated, and threads and **Executors** are not the best ways to carry out asynchronous programming in Android.

Finally, you were introduced to the new, recommended way to carry out asynchronous programming in Android: with Kotlin's Coroutines and Flow. We learned that Coroutines is a Kotlin library that you can use to easily perform asynchronous, non-blocking, and long-running tasks in the background. Flow, built on top of Coroutines, allows you to handle functions that return multiple values over time.

In the next chapter, you will dive deeper into Kotlin coroutines and learn how to use them in your Android project.

Further reading

This book assumes that you have experience and skills in Android development with Kotlin. If you would like to learn more about this, you can read the book *How to Build Android Apps with Kotlin* (Packt Publishing, 2021, ISBN 9781838984113).

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

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