Up to date for Android 9, Android Studio 3.2, & Kotlin 1.3

Kotlin Coroutines by Tutorials

FIRST EDITION

Mastering coroutines in Kotlin and Android

By Filip **Babić** & Nishant **Srivastava**

Table of Contents: Overview

About This Book Sample 4
What You Need 10
Book License 11
Book Source Code & Forums 12
Chapter 1: What Is Asynchronous Programming? ... 13
Chapter 15: Coroutines on Android: Part 1 28
Where to Go From Here? 68

Table of Contents: Extended

About This Book Sample 4
What You Need 10
Book License 11
Book Source Code & Forums 12
Chapter 1: What Is Asynchronous Programming? ... 13
Providing feedback 13
Why multithreading? 15
Interacting with the UI thread from the background 16
Handling work completion using callbacks 19
Indentation hell 21
Using reactive extensions for background work 22
Diving deeper into the complexity of Rx 23
A blast from the past 24
Explaining coroutines: The inner workings 25
Variations through history 25
Key points 26
Where to go from here? 27
Chapter 15: Coroutines on Android: Part 1 28
Getting started 29
Does Android really need coroutines? 31
Coroutines 62
Introducing Anko 64
Key points 65
Where to go from here? 67
Where to Go From Here? 68

About This Book Sample

Kotlin Coroutines by Tutorials will give you the tools you need to solve common programming problems using asynchronous programming.

The importance of concurrency is discovered quite early on by people who start with Android development. Android is inherently asynchronous and event-driven, with strict requirements as to on which thread certain things can happen. Add to this the often-cumbersome Java callback interfaces, and you will be trapped in spaghetti code pretty quickly (aptly termed as "Callback Hell"). No matter how many coding patterns you use to avoid that, you will have to encounter the state change across multiple threads in one way or the other.

The only way to create a responsive app is by leaving the UI thread as free as possible, letting all the hard work be done asynchronously by background threads.

We are pleased to offer you this sample from the full *Kotlin Coroutines by Tutorials* book that will introduce you to these concepts and give you a chance to practice them in our hands-on By Tutorials style.

The chapters that follows introduce you to the foundational concepts of asynchronous programming and how to get started with coroutines on Android.

This sample includes:

- **Chapter 1: What Is Asynchronous Programming?**: Before getting you into the magic of coroutines, we'll help you understand what problem coroutines will solve for you. You'll learn what it means to be asynchronous and how to escape from an "indentation hell." This is a fundamental chapter in order to understand the basics of multi-threading and concurrent programming.

- **Chapter 15: Coroutines on Android: Part 1**: Learn how you can manage concurrency and multi-threading in Android and why coroutines can help you simplify and optimize code in many ways.

The book is ready for purchase at:

- https://store.raywenderlich.com/products/kotlin-coroutines-by-tutorials.

Enjoy!

The *Kotlin Coroutines by Tutorials* Team

Kotlin Coroutines by Tutorials

By Filip Babić and Nishant Srivastava

Notice of Rights

Notice of Liability

Trademarks

Dedications

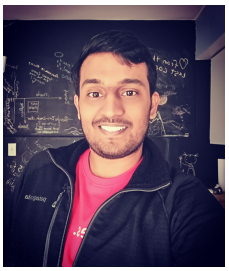
"To my friends and family. And mostly to my loved one. Thank you for being patient and understanding, when I couldn't grab a cup of coffee or tea and catch up. Huge thanks to everyone who's supported me throughout the entire process, with positive and motivational encouragement. This wouldn't have gone as nearly as smooth without you."

— Filip Babić

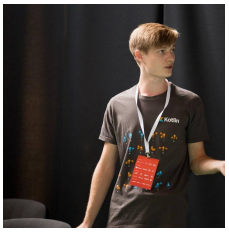
"I would like to thank the many people who have made this book possible. To my father, who gave me the desire to be a curious soul and learn more. To my mom, who has supported me all along whenever I have had doubts about my own capabilities as a writer. To my friends, Saachi Chawla and Kirti Dohrey, who have always believed in me during my ups and downs. To people who have directly or indirectly been my mentor and helped me through understanding technology at a deeper level whenever I found myself stuck. And lastly, to the team at raywenderlich.com, my co-author, editors and everyone involved in making this book a reality."

— Nishant Srivastava

About the Authors



Nishant Srivastava is an author on this book. Nishant is a Sr.Android Engineer at Soundbrenner in Berlin, Germany and an open source enthusiast who spends his time doodling when not hacking on Android. He is a caffeine-dependent life-form and can be found either talking about android libraries or advocating that coffee is the elixir of life at community gatherings. He has been part of two startups in the past (Founding Team Member at OmniLabs, Inc. and one of the first employees at Silverpush) with experience in Android SDK Engineering and Audio Digital Signal Processing(DSP) on Android. While working at his past company (Silverpush), he developed the company's patented UAB (Unique Audio Beacon) Technology.



Filip Babić is an author of this book. He is an experienced Android developer from Croatia, working at the Five Agency, building world-known applications, such as the RosettaStone language-learning application and AccuWeather, the globally known weather reporting app. Previously he worked at COBE d.o.o., a German-owned mobile agency, which is partners with the biggest German media company. He's enthusiastic about the Android ecosystem, focusing extensively on applying Kotlin to Android applications, and building scalable, testable and user-friendly applications. Passionately building up good spirit in local development groups in Croatia, focusing on lectures, education, and engagement of new, aspiring developers in the Croatian IT community. But also pursuing global conferences, meetups, and IT fests. Altruistic when it comes to consulting and mentoring, trying to give help to everyone, whenever possible, motivated by the ideology that the Android ecosystem we live in is only as good as we make it.

About the Editors

Eric Crawford is a tech editor of this book. Eric is a Senior Software Developer at John Deere, where he bounces between iOS and Android development. Before coming to Deere he did freelance mobile development and serverside web development utilizing Java. In his free time he likes to dabble into other platforms like IOT and cloud computing.

Massimo Carli is the final pass editor of this book. Massimo has been working with Java since 1995 when he co-founded the first Italian magazine about this technology (http://www.mokabyte.it). After many years creating Java desktop and enterprise application, he started to work in the mobile world. In 2001 he wrote his first book about J2ME. After many J2ME and Blackberry applications, he then started to work with Android in 2008. The same year he wrote the first Italian book about Android; best seller on Amazon.it. That was the first of a series of 8 books. he worked at Yahoo and Facebook and he's actually Engineering Tech Lead at Lloyds. He's a musical theatre lover and a supporter of the soccer team S.P.A.L.

Manda Frederick is an editor of this book. She has been involved in publishing for over ten years through various creative, educational, medical and technical print and digital publications, and is thrilled to bring her experience to the raywenderlich.com family as Managing Editor. In her free time, you can find her at the climbing gym, backpacking in the backcountry, hanging with her dog, working on poems, playing guitar and exploring breweries.

About the Artist

Vicki Wenderlich is the designer and artist of the cover of this book. She is Ray's wife and business partner. She is a digital artist who creates illustrations, game art and a lot of other art or design work for the tutorials and books on raywenderlich.com. When she's not making art, she loves hiking, a good glass of wine and attempting to create the perfect cheese plate.

What You Need

To follow along with this book, you'll need the following:

- **IntelliJ IDEA Community Edition 2018.2**: Available at https://www.jetbrains.com/idea/. This is the environment in which you'll develop most of the sample code in this book.
- **Jave SE Development Kit 8.**: Most of the code in this book will be run on the Java Virtual Machine or JVM, for which you need a Java Development Kit or JDK. The JDK can be downloaded from Oracle at http://www.oracle.com/technetwork/java/javase/downloads/index.html.
- **Android Studio 3.x.**: For the examples about Android described in Section 3, you can the IDE available at https://developer.android.com/studio/.

If you haven't installed the latest versions of IntelliJ IDEA Community Edition and JDK 8, be sure to do that before continuing with the book. Chapter 2: "Setting Up Your Build Environments" will show you how to get started with IntelliJ IDEA to run Kotlin coroutines code on the JVM.

Book License

By purchasing *Kotlin Coroutines by Tutorials*, you have the following license:

- You are allowed to use and/or modify the source code in *Kotlin Coroutines by Tutorials* in as many apps as you want, with no attribution required.
- You are allowed to use and/or modify all art, images and designs that are included in *Kotlin Coroutines by Tutorials* in as many apps as you want, but must include this attribution line somewhere inside your app: "Artwork/images/designs: from *Kotlin Coroutines by Tutorials*, available at www.raywenderlich.com."
- The source code included in *Kotlin Coroutines by Tutorials* is for your personal use only. You are NOT allowed to distribute or sell the source code in *Kotlin Coroutines by Tutorials* without prior authorization.
- This book is for your personal use only. You are NOT allowed to sell this book without prior authorization, or distribute it to friends, coworkers or students; they would need to purchase their own copies.

All materials provided with this book are provided on an "as is" basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

All trademarks and registered trademarks appearing in this guide are the properties of their respective owners.

B Book Source Code & Forums

This book comes with the source code for the starter and completed projects for each chapter. These resources are shipped with the digital edition you downloaded from store.raywenderlich.com.

We've also set up an official forum for the book at forums.raywenderlich.com. This is a great place to ask questions about the book or to submit any errors you may find.

Chapter 1: What Is Asynchronous Programming?

By Filip Babić

The **UI (user interface)** is a fundamental part of almost every application. It's what users see and interact with in order to do their tasks. More often than not, the applications do **complex work**, such as talking to external services or processing data from a database. Then, when the work is done, it shows a **result**, mostly in some form of a message.

The UI must be **responsive**. If the work at hand takes a lot of time to complete, it's necessary to provide **feedback** to the users so that they don't feel like the application has frozen, for example, or that they didn't click a button properly — or perhaps that the feature doesn't work at all.

In this chapter, you'll learn how to provide useful information to the users about what's happening in the application and what different mechanisms exist for working with multiple tasks. You'll see what problems arise while trying to do complex and long-running synchronous operations and how asynchronous programming comes to rescue.

You'll start off by analyzing the flow of a function that deals with data processing and provides feedback to the user.

Providing feedback

Suppose you have an application that needs to upload content to a network. When the user selects the Upload button, loading bars or spinners appear to indicate that something is ongoing and the application hasn't stop working. This information is crucial for a good user experience since no one likes unresponsive applications. But what does providing feedback look like in code?

Consider the following task wherein you want to upload an image but must wait for the application to complete the upload:

```
fun uploadImage(image: Image) {
  showLoadingSpinner()
  // Do some work
  uploadService.upload(image)
  // Work's done, hide the spinner
  hideLoadingSpinner()
}
```

At first glance, the code gives you an idea of what's happening:

- You start by showing a spinner.
- You then upload an image.
- When complete, you hide the spinner.

Unfortunately, it's not exactly that simple because the spinner contains an animation, and there must be code responsible for that. The `showLoadingSpinner` function must then contain code such as this:

```
fun showLoadingSpinner() {
  showSpinnerView()
  while(running) {
    rotateSpinnerImage()
    delay()
  }
}
```

The `showSpinnerView` displays the actual `View` component, and the following cycle manages the image rotation. But when does this function actually return?

In the `uploadImage` code, you assumed that the spinner animation was running even after the completion of the `showLoadingSpinner` function, so that the uploading of the image could start. Looking at the previous code, this is not possible. If the spinner is animating, it means that the `showLoadingSpinner` has not completed. If the `showLoadingSpinner` has completed, and so the `upload` has started, it means that the spinner is not animating anymore. This is happening because when you invoke the `showLoadingSpinner` function you're making a **blocking call**.

Blocking calls

A **blocking call** is essentially a function that only returns when it has completed. In the example above, the `showLoadingSpinner` function prevents the upload of an image because it keeps the **main thread** of execution busy until it returns. But, when it returns, because the `running` variable becomes `false`, the spinner stops rotating.

So how can you solve this problem and animate the spinner even while the `upload` function is executing?

Simply put, you need additional threads on which to execute your long-running tasks.

The **main thread** is also known as **UI thread**, because it's responsible for rendering everything on the screen, and this should be **the only thing it does**. This means that it should manage the rotation of the spinner but not the upload of the image — that has nothing to do with UI. But if the **main thread** cannot do it because that isn't its job, what can execute the upload task? Well, quite simply, you need **a new thread** on which to execute your long-running tasks!

Computers nowadays are far more advanced than they were 10 or 15 years ago. Back in the day computers could only have one thread of execution, making them freeze up often, if you tried to do multiple things at once. But because of technological advancements, your applications support a mechanism known as **multi-threading**. It's the art of having multiple threads, where each can process a piece of work, collectively finishing the needed tasks.

Why multithreading?

There's always been a hardware limit on how fast computers could be — that's not really about to change. Moreover, the number of operations a single processor in a computer can complete is reaching the law of diminishing returns.

Because of that, technology has steered in the direction of increasing the number of cores each processor has, and the number of threads each core can have running **concurrently**. This way, you could logically divide any number of tasks between different threads, and the cores could prioritize work by organizing them. And, by doing so, multithreading has drastically improved how computer systems optimize work and the speed of execution.

You can apply the same idea to modern applications. For example, rather than spending large amounts of money on servers with better hardware, you can speed up the entire system using **multithreading** and a smart application of **concurrency**.

Comparing the main and worker threads

The **main thread**, or the **UI thread**, is the thread responsible for managing the UI. Every application can only have one main thread in order to avoid a classical problem called **deadlock** that can happen when many threads access the same resources — in

this case, UI components — in a different order. The other threads, which are not responsible for rendering the UI, are called **worker threads** or **background threads**. The ability to allow the execution of multiple threads of control is called **multithreading**, and the set of techniques in order to control their collaboration and synchronization, is called **concurrency**.

Given this, you can rethink how the `uploadImage` function should work. The `showLoadingSpinner` starts a new thread that is responsible for the rotation of the spinner image, which interacts with the main thread just to notify a refresh in the UI. Starting a new thread, the function is now a **no blocking call** and can returns immediately allowing the image upload to start into its own worker thread. When completed, this background thread will notify the main thread to hide the spinner.

Once the program launches a background thread, it can either forget about it or expect some result. You will see how background threads process the result, and communicate it to the main thread, in the following section.

Interacting with the UI thread from the background

The upload image example demonstrates how important managing threads is. The thread responsible to rotate the spinner image needs to communicate with the main thread in order to refresh the UI at each frame. The worker thread responsible for the actual upload needs, when it completes, to communicate with the previous in order to stop the animation and with the main thread for hiding the spinner. All this must happen without any type of blocks. Knowing how threads *communicate* is key to achieving the full potential of concurrency.

Sharing data

In order to communicate, different threads need to share some data. For instance, the thread responsible for the rotation of the spinner image needs to notify the main thread that a new image is ready to be displayed. Sharing data is not simple, and it needs some sort of synchronization that is the main reason for concurrency.

What happens, for instance, if the main thread receives a notification that a new image is available and, before displaying it, the image is replaced? In this case, the application would skip a frame and a **race condition** would happen. You then need some sort of **thread safe** data structure. This means that the data structure should work correctly even if accessed by multiple threads at the same time.

Accessing the same data from multiple threads maintaining the correct behavior and good performance, is the real challenge of **concurrent programming**.

There are special cases, however. What if the data is only accessed and never updated? In this case, multiple threads can read the same data without any race condition, and your data structure is referred as **immutable**. Immutable objects are always thread safe.

As a pratical example, take a coffee machine in an office. If two people shared it, and it wasn't thread safe, they could easily make bad coffee or spill it around and make a mess. As one person started making a mocha latte, but another would want a black coffee, they would ultimately ruin the machine — or worse, the coffee.

What are the data structures that you can use in order to safely share data in a thread? The most important data structures are **queues** and, as a special case, **pipelines**.

Queues

Threads usually communicate using **queues**, and they can act on them as **producers** or **consumers**. A producer is a thread that puts information into the queue, and the consumer is the one that reads and uses them. You can think of a queue as a list in which producers append data at the end, and then consumers read data from the top, following a logic called FIFO (First In First Out). Threads usually put data into the queue as objects called **messages**, which encapsulate the information to share.

A queue is not just a **container**, but provides synchronization in order to allow a thread to consume a message only if it is available, otherwise, it waits if the message is not available. Depending if the queue is a **blocking queue**, the consumer can block and wait for a new message — or just retry later.

The same can happen for the producer if the queue is full. Queues are thread safe, so it is possible to have multiple producers and multiple consumers.

A great real-life example of queues are fast food lines.

Imagine having three lines at a fast food restaraunt. The first line has no customers, so the person working the line is blocked until someone arrives. The second has customers, so the line is slowly getting smaller as the worker serves customers. However, the last line is full of customers, but there's no one to serve them; this, in turn, blocks the line until help arrives.

In this example customers form a queue waiting to consume what the fast food workers are preparing for them. When the food is available, the customer consume it and leaves the queue.

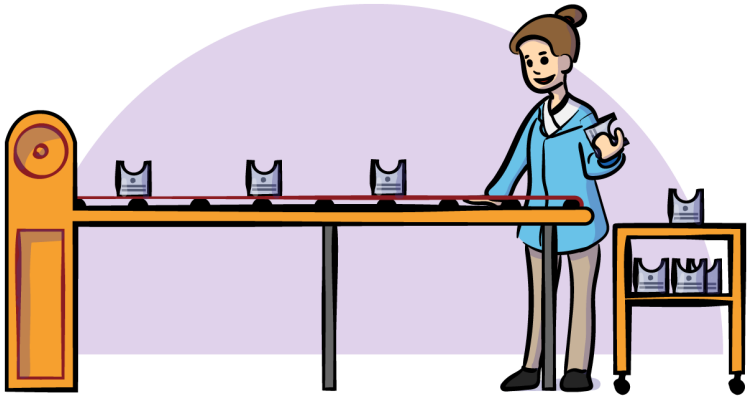
Pipelines

If you think about pipes, or faucets, and how they work, it's a fairly simple concept. When you release the pressure by turning the valve, you're actually **requesting** water. On the other side of that request, there's a system that **regulates the flow** of water. As soon as you make a request, it is blocked until the water comes running — just like a blocking call.

The same process is used for **pipelines** or **pipes** in programming. There's a pipe that allows **streams of data** to flow, and there are **listeners**. The data is usually a stream of bytes, which the listeners parse into something more meaningful.

As an example, you can also think about factory lines. Just like in a factory line, if there's too much product, the line has to stop until you process everything. That is, if there's **too much** data that you haven't yet processed, the pipeline is blocked until you consume some of the data and make room for more to flow. And, alternatively, if there's not enough product, the person processing it sits and waits until something comes up.

In other words, if there's **not enough** data to flow — the pipe is empty — you're blocked until some data emerges. Because you're either trying to send data to an overflowed stream, or trying to get data from an empty stream, the mechanism doesn't know how to react but to block until the conditions are met.



You can think of pipes as blocking queues wherein you don't have messages, but chunks of bytes.

Handling work completion using callbacks

Out of all the asynchronous programming mechanisms, **callbacks** are the most often used. This consists in the creation of objects that encapsulate code that somebody else can execute later — when a specific task completes, for example. This approach can be also used in real life when you ask somebody to push a button when they have completed some task you have assigned to them. When using **callbacks**, the button is analogous to code for them to execute; the person executing the task is a **no blocking function**.

How can you put some code into an object to pass around? One way is by using **interfaces**. You can create the interface in this way:

```
interface OnUploadCallback {

  fun onUploadCompleted()
}
```

With this, you are passing an implementations to the function that is executing the long-running task. At completion, this function will invoke `onUploadCompleted` on the object. The function doesn't know what that implementations does, and it's not suppose to know it.

In modern programming languages, like Kotlin, which supports functional programming features, you can do the same with a **lambda** expression. In the previous example, you could pass the lambda to the upload function as a callback. The lambda would then contain the code to execute when the upload task completes:

```
fun uploadImage(image: Image) {
  showLoadingSpinner()

  uploadService.upload(image) { hideLoadingSpinner() }
}
```

Looking back at the first snippet, not much has changed. You still show a loading spinner, call the `upload` method and hide the spinner when the upload is done. The core difference, though, is that you're not calling `hideLoadingSpinner` right after the upload. That function is now part of the **lambda block**, passed as parameter to the `upload`, which will execute it at completion.

Doing so, you can call the wrapped function anytime you're done with the connected task. And the lambda block can do pretty much anything, not just hide a loading spinner.

In case some value is returned, it has passed down the lambda block, so that you can use it from within. Of course, the inner implementation of the `uploadService` depends on the service and the library that you're using. Generally, each library has its own type of callbacks. However, even though callbacks are one of the most popular ways to deal with asynchronicity, they have become notorious over the years. You'll see how in the next section.

Indentation hell

Callbacks are simpler than building your own mechanisms for thread communication. They syntax is also fairly readable, when it comes to simple functions. However, it's often the case that you have **multiple function calls**, which need to be **connected or combined** somehow, mapping the results into more complex objects.

In these cases, the code becomes extremely difficult to write, maintain and reason. Since you can't return a value from a callback, but have to pass it down the lambda block itself, you have to **nest callbacks**. It's similar to nesting `forEach` or `map` statements on collections, where each operation has its lambda parameter.

When nesting callbacks, or lambdas, you get a large number of braces '`{}`', each forming a **local scope**. This, in turn, creates a structure called **indentation hell** — or **callback hell**, when it's specific to callbacks. A good example, following our example so far, would be fetching, resizing and uploading images:

```
fun uploadImage(imagePath: String) {
  showLoadingSpinner()

  loadImage(imagePath) { image ->
      resizeImage(image) { resizedImage ->
          uploadImage(resizedImage) {
              hideLoadingSpinner()
          }
      }
  }
}
```

You show the upload spinner before the upload itself, as before. But, after you load the image from a file, you proceed to resize it. Next, when you've resized the image successfully, you start uploading it. Finally, once you manage to upload it, you hide the loading spinner.

The first thing you notice is the amount of braces and indentation that form a **stairs-like** code structure. This makes the code *very hard* to read, and it's not even a complex operation. When building services on the web, nesting can easily reach 10 levels, if not more. Not only is the code hard to read, but it's also extremely hard to maintain such code. Because of the structure, you suffer from cognitive load, making it harder to reason about the functionality and flow. Trying to add a step in between, or change the lambda-result types, will break *all* the subsequent levels.

Additionally, some people find callbacks really hard to grasp at first. Their steep learning curve, combined with the cognitive load and the lack of extensibility, make people look elsewhere for a solution to asynchronous programming.

This is where **reactive extensions** come to life. You'll see how they solve the nesting problem in the next section.

Using reactive extensions for background work

The most significant issue of a callback-based approach is passing the data from one function to another. This results in **nested callbacks**, which are tough to read and maintain. If you think about the queues and pipes, they operate with **streams** of data, wherein you can **listen** to the data as long as you need.

Rx was built upon the idea of having asynchronous operations wrapped up in streams of events.

Rx incorporates the **observer pattern** into helpful constructs. Furthermore, there are a large number of **operators**, extending the behavior of **observable streams**, allowing for clean and expressive data processing. You can **subscribe** to a **stream of events**, map, filter, reduce and combine the events in numerous ways, as well as handle errors in the entire **chain of operations**, using a *single* lambda function.

The previous example of loading, uploading and resizing an image, using Rx, can be represented:

```
fun uploadImage(imagePath: String) {
  loadImage(imagePath)
          .doOnSubscribe(::showLoadingSpinner)
          .flatMap(::resizeImage)
          .flatMap(::uploadImage)
          .subscribe(::hideLoadingSpinner, ::handleError)
}
```

At first, the code might look weird. In reality, it's a stream of data modified by using a bunch of operators. It begins with the `flatMap` operator, which takes some data — the image from `loadImage` function — and passes it to another function, creating a new stream. Then, the new stream sends events in the form of `resizedImage` value, which gets passed to the `uploadImage`, again using `flatMap`, and operator chaining.

Finally, the `uploadImage` stream doesn't pass data but, rather, **completion events**, which tell you to hide the loading spinner, since the upload has finished.

These streams of data and operations don't actually get executed until someone **subscribes** to them, using the `subscribe(onComplete, onError)` function.

Additionally, the `doOnSubscribe` function takes an action that the stream executes whenever you subscribe to it. There are also functions like `doOnSuccess` and `doOnError`, which propagate their respective events.

Further, it's important to know that, if any error or exception occurs in any of the operations in a chain, *it's not thrown*, and the application doesn't crash. Instead, the stream passes it down the chain, finally reaching the `onError` lambda. Callbacks do not have this behavior; they just throw the exception and you have to handle it yourself, using `try/catch` blocks.

Reactive extensions are cleaner than callbacks when it comes to asynchronous programming, but they also have a steeper learning curve. With dozens of operators, different types of streams and a lot of edge cases about switching between threads, it takes a larger amount of time to fully understand them.

The learning curve, and a few other issues, will be discussed in the next section.

Diving deeper into the complexity of Rx

Since this book isn't about Rx, you'll only have a narrow overview of its positive and negative features. As seen before, Rx makes asynchronous programming clean and readable. Further, compared with the operators that allow for data processing, Rx is a powerful mechanism. Moreover, the error handling concept of streams adds extra safety to applications.

But Rx is not perfect. It has its problems like any other framework, or paradigm, some of which are really coming up in the programming community lately.

To start, there is the learning curve. When you start learning Rx, you have to learn a number of additional concepts, such as the **observer pattern** and **streams**. You will also find that Rx is not just a framework; it brings a completely new paradigm called **reactive programming**. Because of this, it's very hard to start working with Rx. But it's even harder to grasp the finesse of its operators. The amount of **operators**, types of **thread scheduling**, and the combinations between the two, creates so many options that it's nearly impossible to know the full extent of Rx.

Another problematic issue with using Rx is the *hype*. Over the years, people have moved towards Rx as a silver bullet for asynchronous operations.

This eventually led to such programming being Rx-driven, introducing even more complexity to existing applications. Finding workarounds and using numerous design patterns, just to make Rx work, introduced new layers of unwanted complexity. Because

of this, in Android, the Rx community has been debating if programmers should represent things like network requests as streams of data versus just a single event that they could handle using callbacks or something even simpler. The same debate transitions to navigation events, as an example. Should programmers represent clicks as streams of events, too? The community opinion is very divided on this topic.

So, with all this in mind, is there a better or simpler way to deal with asynchronicity? Oddly enough, there's a concept dating back decades, which has recently become a hot topic.

A blast from the past

This is a book about **coroutines**. They're a mechanism dating to the 1960's, depicting a unique way of handling asynchronous programming. The concept revolves around the use of **suspension points**, **functions** and **continuations** as first-class citizens in a language.

They're a bit abstract, so it's better to show an example:

```
fun fetchUser(userId: String) {
  val user = userService.getUser(userId) // 1

  print("Fetching user") // 2
  print(user.name) // 3
  print("Fetched user") // 4
}
```

Using the above code snippet, and revisiting what you learned about blocking calls, you'd say that the execution order was `1, 2, 3` and `4`. If you carefully look at the code, you realize that this is not the only possible logical sequence. For instance, the order between `1` and `2` is not important, nor is the order between `3` and `4`. What is important is that the user data are fetched before they are displayed; `1` must happen before `3`. You can also delay the fetching of the user data to a convenient time before the user data is actually displayed. Managing these issues in a transparent way is the black magic called **coroutines**!

They're a part-thread, part-callback mechanism, which use the system's power of scheduling and suspending work. This way, you can immediately return a result from the call *without* using callbacks, threads or streams. Think of it this way, once you start a coroutine, or call a suspension functions, it gets nicely wrapped up and prepared like a taco. But, until you want to eat the taco, the code inside might not get executed.

Explaining coroutines: The inner workings

It's not really magic — only a smart way of using low-level processing. The `getUser` function is marked as a **suspension function**, meaning the system prepares the call in the background, and you get an unfinished, wrapped taco. But it might not execute the function yet. The system moves it to a **thread pool**, where it waits further command. Once you're ready to eat the taco and you call the result, the program blocks until you get a ready-to-go snack.

Knowing this, the program can skip over to the rest of the function code, until it reaches the first line of code on which it uses the `user`. This is called **awaiting** the result. At that point, it executes the `getUser` function, if it hasn't already, blocking the program.

This means you can do as much processing as you want, in between the call itself and using its result. Because the compiler knows suspension points and functions are asynchronous, but treats their execution sequentially, you can write understandable and clean code, which is very extensible and easy to maintain.

Since writing asynchronous code is so simple with coroutines, you can easily combine multiple requests or transform the data. No more staircases, strange stream mapping to pass the data around, or complex operators to combine or transform the result. All you need to do is mark functions as suspendable, and call them in a coroutine block.

Another, extremely important thing to note about coroutines is that they're not threads. They are a low-level mechanism that utilizes **thread pools** to shuffle work between multiple, existing threads. This allows you to create millions of coroutines, without overflowing the memory; a million threads would take so much memory, even today's state-of-the-art computers would crash.

Although many languages support coroutines, each has a different implementation.

Variations through history

As mentioned, coroutines are a dated but powerful concept. Throughout the years, several programming languages have evolved their version of the implementation. For example, in languages like Python and Smalltalk, coroutines are first-class citizens, and can be used without an external library.

A **generator** in Python would look like this:

```
def coroutine():
    while True:
        value = yield
        print('Received a value:', value)
```

This code defines a function, which loops forever, listening and printing any arguments you send to it. The concept of an infinite loop, which listens for data is called a generator. The keyword `yield` is what triggers the generator, receiving the value. As you can see, there's a `while True` statement in the function. In regular code, this would create an standard infinite loop, effectively blocking the program, since there's no exit condition. But this is a coroutine-powered call, so it waits in the background until you send some value to the function, which is why it doesn't block.

Another language with first-class coroutines is C#. In C#, there's a support for the `yield` statement, like in Python, but also for `async` and `await` calls, like this:

```
MyResult result = await AsyncMethodThatReturnsAResult();

await AsyncMethodThatReturnsAResult()
```

Here, by adding the `await` keyword, you can return an asynchronous result, using normal, sequential code. It's pretty much what you saw in the example above, where you first learned about coroutines.

Both Python and C# have first-class support for coroutines. Many other programming languages utilize external libraries in order to support programming with coroutines. Kotlin also has coroutines support in its standard library. By including them in the language itself, it allows you to make asynchronous calls without including a third-party framework. Additionally, the way Kotlin coroutines are built — global functions with receivers, makes them very **extensible**, you can create your own API by building on top of the existing functions.

You'll see how to do this in the next chapters of the book.

Key points

- **Multithreading** allows you to run multiple tasks in parallel.
- **Asynchronous programming** is a common pattern for thread communication.
- There are different **mechanisms** for sharing data between threads, some of which are queues and pipelines.

- Most mechanisms rely on **push-pull** tactic, blocking threads when there is too much, or not enough data, to process
- **Callbacks** are a complex, hard-to-maintain and cognitive-load-heavy mechanism.
- It's easy to reach **callback hell** when doing complex operations using callbacks.
- **Reactive extensions** provide clean solutions for data transformation, combination and error handling.
- Rx can be too complex, and doesn't fit all applications.
- **Coroutines** are an established, and reliable concept, based on low-level scheduling.
- Too many **threads** can take up a lot of memory, ultimately crashing your program or computer.
- **Coroutines** don't always create new threads, they can reuse existing ones from thread pools.
- It's possible to have asynchronous code, written in a clean, **sequential** style, using coroutines.

Where to go from here?

Well that was a *really brief* overview of the history and theory beind asynchronous programming and coroutines.

If you're excited about some code and Kotlin's coroutines, in the next section of the book you'll learn about **suspending functions** and **suspension points**. Moreover, you'll see how coroutines are created in Kotlin, using **coroutine builders**. Next, you'll build asynchronous calls, which return some data with the **async** function, and see how you await the result. And, finally, you'll learn about **jobs** and their children, in coroutines.

You'll cover the entire base API for Kotlin Coroutines, learning how to wrap asynchronous calls into async blocks, how to combine multiple operations and how to build Jobs which have multiple layers of coroutines.

But before that, you have to set up your build environment, so let's get going!

Chapter 15: Coroutines on Android: Part 1

By Nishant Srivastava

The importance of concurrency is discovered quite early on by people who start with Android development. Android is inherently asynchronous and event-driven, with strict requirements as to on which thread certain things can happen. Add to this the often-cumbersome Java callback interfaces, and you will be trapped in spaghetti code pretty quickly (aptly termed as "Callback Hell"). No matter how many coding patterns you use to avoid that, you will have to encounter the state change across multiple threads in one way or the other.

The only way to create a responsive app is by leaving the UI thread as free as possible, letting all the hard work be done asynchronously by background threads.

> **Note**: You've already met the term "Callback Hell" in the first chapter. It's the situation in which you have to serially execute and process the results of asynchronous services by nesting callback, often several layers deep.

The purpose of coroutines is to take care of the complications in working with asynchronous programming. You write code sequentially, like you usually would, and then leave the hard asynchronous work up to coroutines.

Using coroutines in Android provides some of the following benefits:

- Coroutines are a language feature provided out of the box by Kotlin and, thus, they can be updated independently from the Android platform releases.
- Coroutines make asynchronous code look synchronous, making the code more readable. Also, since a synchronous sequence of steps is much easier to manage — as opposed to asynchronous code — coroutines enable greater confidence in changing the flow when needed.

- Thanks to coroutines, getting rid of any callbacks and the need to pass around state information is fairly easy, i.e., storing temporary state in Presenter/ViewModel is simplified and state is not passed across multiple methods any longer.
- Coroutines enable better, concise and testable code.

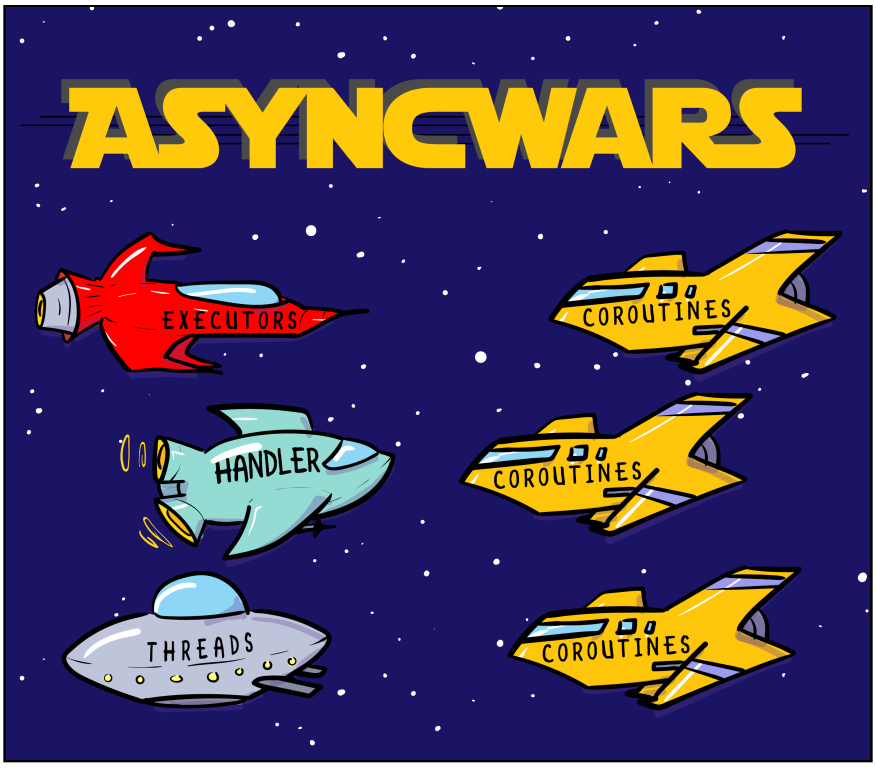
In this chapter, you'll learn about what different mechanisms already exist for asynchronous programming on the Android platform and why coroutines are a much better replacement for all of them. You'll see what Kotlin coroutines bring to the table and how they simplify various facets of Android development.

Getting started

For this chapter, you will use a basic app called **Async Wars** in order to learn about various async primitives in Android and coroutines at a high level. If you have already downloaded the starter project, then import it into Android Studio.

The project consists of some pre-written utility classes under the package `utils`. Let's go over them one by one:

1. `DownloaderUtil`: A **singleton** which has a method called `downloadImage()` that fetches an image from a pre-setup URL returning a `Bitmap`. This is done on the main thread and it will be your goal to execute this method on a background thread, and then you will display the image on the screen.
2. `ImageDownloadListener`: Interface which is used as a listener for images being downloaded.
3. `BroadcasterUtil`: A **singleton** which is used to abstract away the calls made using `LocalBroadcastManager`.
4. `MyBroadcastReceiver`: Implementation of `BroadcastReceiver` class used as an **adapter** between the sender and an `ImageDownloadListener`.
5. `Extensions.kt`: Utility Kotlin extension methods.



Async Wars

Under the package `async`, you will find `GetImageAsyncTask` and `MyIntentService` classes, which will be used and discussed at a later stage in this chapter.

Apart from that, there is `MainActivity` class wherein everything is wired up for making calls to download images using various async constructs in Android, and to display them in the UI. Almost all the code is pre-written to make it easier for you to switch between these async constructs and see the results. There are two important sections inside `MainActivity` class that you should take note of:

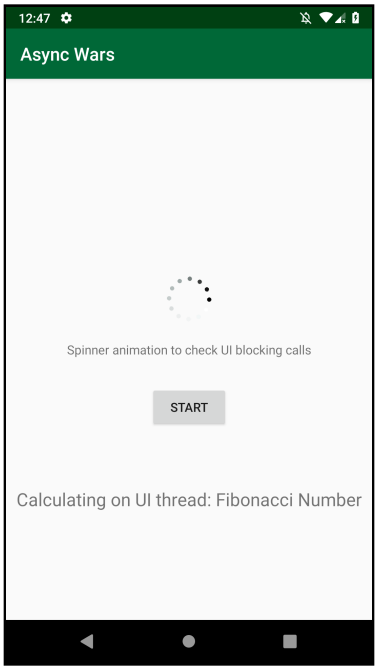
1. `MethodToDownloadImage`: This is an enum class defined inside the `MainActivity` class, which enumerates all the various types of async construct types in Android.
2. Inside the `onCreate()` is a code region marked to be modified:

```
//region
val doProcessingOnUiThread = true
val methodToUse = MethodToDownloadImage.Thread
//endregion
```

This is where you will mostly make the changes to trigger the right kind of async construct for downloading an image and displaying it in the UI. Here, when working with async constructs, you will have to set `doProcessingOnUiThread = false`. After that, the value of `methodToUse`, which will be one of the items from the

`MethodToDownloadImage` enum class, will be used later to trigger the specific async method. When not dealing with async constructs, simply set back to `doProcessingOnUiThread = true`.

Run the app. You will see a UI like below with a button and an animating spinner. The spinner is there to show the impact of calls on the UI thread while a widget is animating. The button will trigger a calculation of a Fibonacci sequence number on the main thread when the flag `doProcessingOnUiThread` is set to `true`.



Starter Project

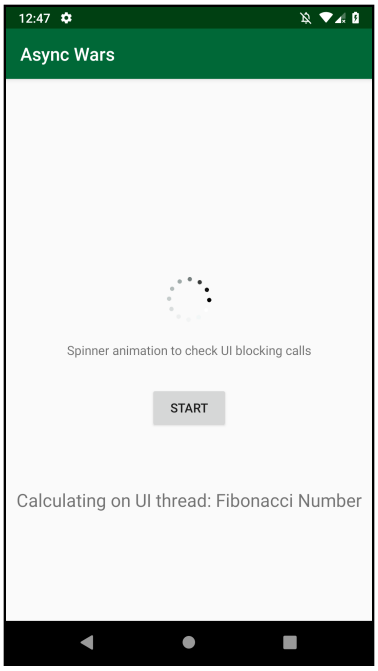
Does Android really need coroutines?

When you start an Android application, the first thread spawned by its process is the main thread, also known as the UI thread. This is the most important thread of an application. It is responsible for handling all the user interface logic, user interaction and also tying the application's moving parts together.

Android takes this very seriously; if your UI thread is stuck working on a task for more than a few seconds, the Android framework will throw an **Application Not Responding** (ANR) error and the app will crash. Most importantly, even small work on the UI/Main thread can lead to your UI freezing, i.e., animations will stop, and the UI will become non-responsive to the user interaction; everything will stop until the work is finished.

To demonstrate this behavior, inside the `MainActivity.kt` of the starter app, make sure that the value of the flag `doProcessingOnUiThread` is set to `true`. If it is, then simply run the app.

You will see the below app state:



UI blocking processing

Now, click the Start button in the UI. This will trigger a call to `runUiBlockingProcessing()` method. Here is the method definition:

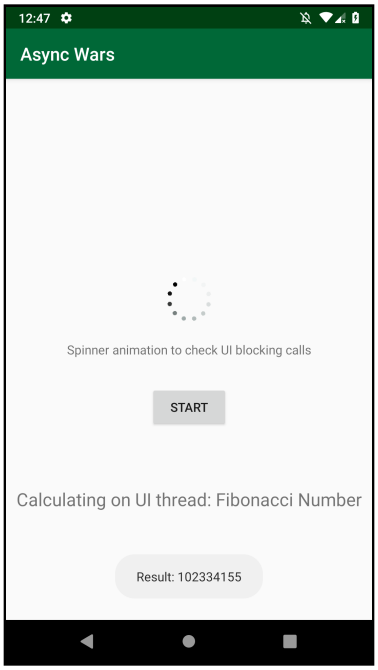
```
fun runUiBlockingProcessing() {
  // Processing
  showToast("Result: ${fibonacci(40)}")
}
```

Here, `fibonacci(number)` method is a helper method and has the below naive implementation:

```
// ----------- Helper Methods -----------//
fun fibonacci(number: Int): Long {
  return if (number == 1 || number == 2) {
    1
  } else fibonacci(number - 1) + fibonacci(number - 2)
}
```

Here, the `runUiBlockingProcessing()` method starts a calculation of the 40th Fibonacci sequence number. Since the processing is done on the UI thread, you will see that the animating spinner stops until the calculation has completed.

You will see a toast message with the result value when the calculation completes, after which the spinner start animating again.



UI blocking processing

Now, here is the problem: almost all code in an Android application will be executed on the UI thread by default. Since the tasks on a thread are executed sequentially, this means that your user interface could become unresponsive while it is processing some other work.

Long-running tasks called on the UI thread could be fatal to your application, leading to an ANR dialog, which gives the user the opportunity to force-quit the application. Even small tasks can compromise the user experience; hence, the correct approach is to move as much work off of the UI thread onto a background thread.

Android comes with some pre-built solutions to handle such situations, but, due to its design, it has proven to be really difficult for many. Using the low-level threading packages with Android means that you have to worry about a lot of tricky synchronization to avoid race conditions or, worse, deadlocks. The good news is that the folks working on the Android framework noticed this and provided better API to deal with such situations. **AsyncTask**, **IntentService**, **ExecutorService**, etc. are some of the very useful classes, as well as the **HaMeR** classes **Handler**, **Message** and **Runnable**. Each comes with its own pros and cons.

Let’s take a quick look at each one of them.

> **Note**: Before you continue with the chapter, from here onwards, inside the `MainActivity.kt` of the starter app, ensure that the value of the flag `doProcessingOnUiThread` is set to `false`. You will not be needed to set it to `true` anymore.

Threads

A thread is an independent path of execution within a program. Every thread in Java is created and controlled by a `java.lang.Thread` instance. A Java program can have many threads, and these threads can run concurrently, either asynchronously or synchronously.

Every Android developer, at one point or another, needs to deal with threads in their application. The main thread is responsible for dispatching events to the appropriate user-interface widget, as well as communicating with components from the Android UI toolkit. To keep your application responsive, it is essential to avoid using the main thread to perform operations that may last for long.

Network operations and database calls, as well as the loading of certain components, are common examples of operations that should not run in the main thread. When they are called in the main thread, they are called synchronously, which means that the UI will remain completely unresponsive until the operation completes. For this reason, they are usually performed in separate threads, which thereby avoids blocking the UI while they are being performed (i.e., they run asynchronously from the UI).

Sample usage:

You can create a thread in two ways:

1. Extending the `Thread` class:

```
// Creation
class MyThread : Thread() {

  override fun run() {
    doSomeWork()
  }
}

// Usage
val thread = MyThread()
thread.start()
```

2. Passing a `Runnable` interface implementation as `Thread` constructor parameter:

```
// Creation
class MyRunnable : Runnable {

  override fun run() {
    doSomeWork()
  }
}

// Usage
val runnable = MyRunnable()
val thread = Thread(runnable)
thread.start()
```

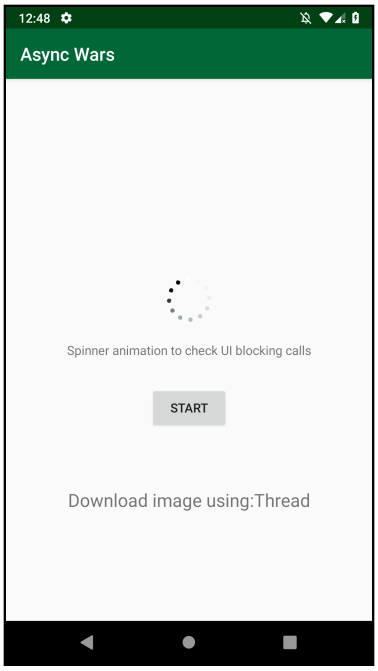
To see a working example, in your **MainActivity.kt** file under the `onCreate()` function, set `methodToUse = MethodToDownloadImage.Thread`. This makes sure that, when the button is clicked, the method `getImageUsingThread()` is called. Here is the method definition:

```
fun getImageUsingThread() {
  // Download image
  val thread = Thread(myRunnable)
  thread.start()
}
```

Where `myRunnable` has the below implementation:

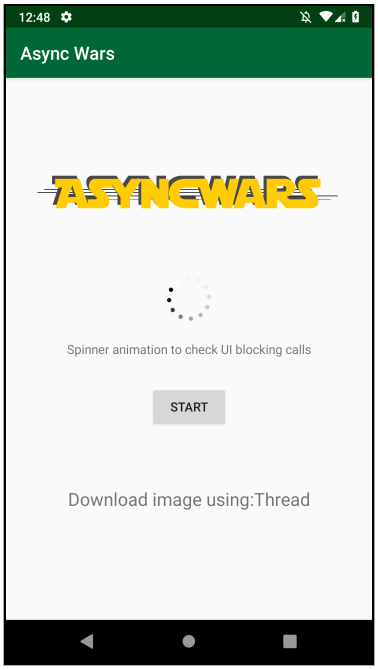
```
inner class MyRunnable : Runnable {
  override fun run() {
    // Download Image
    val bmp = DownloaderUtil.downloadImage()

    // Update UI on the UI/Main Thread with downloaded bitmap
    runOnUiThread {
      imageView?.setImageBitmap(bmp)
    }
  }
}
```

Run the app.



Download image using Thread

When you click the Start button, you will see that the image is downloaded and displayed in the `ImageView` without blocking the UI; the spinner animates while the image is being downloaded.



Download image using Thread

It's important to note how the downloaded image has been passed to the UI thread using the `runOnUiThread()` function that you inherit from the `Activity` class.

> **Note**: The animation will stop now just for a very short time. Passing an image from one thread to another never comes for free.

```
inner class MyRunnable : Runnable {
  override fun run() {
    // Download Image
    val bmp = DownloaderUtil.downloadImage()

    // Update UI on the UI/Main Thread with downloaded bitmap
    runOnUiThread {
      imageView?.setImageBitmap(bmp)
    }
  }
}
```

Interacting with UI components from a background thread would have caused an error like this:

```
E/AndroidRuntime: FATAL EXCEPTION: Thread-4
    Process: com.raywenderlich.android.asyncwars, PID: 3127
    android.view.ViewRootImpl$CalledFromWrongThreadException:
      Only the original thread that created a view hierarchy can touch
its views.
```

The operative system's scheduler is responsible for the management of the lifecycle of each thread. It can execute, suspend and resume threads depending on its state and some synchronization requirement. This is an expensive job and, if you try to launch a high number of threads — a million, for example — your processor will spend more time changing from one thread to another than executing the code you want it to execute. This is called **context switch**. Every `Thread` you instantiate in Java (or Kotlin) corresponds to a thread of the operating system (either physical or virtual), and, therefore, it is the scheduler of the operating system that is in charge of prioritizing which thread should be executed in every moment.

In a nutshell, threads might be:

- **Expensive**: Context switching and having upper limits in the number of threads that can be spawned.
- **Difficult**: Creating a multithreaded program is quite complex requiring a lot of ceremonies around how the code is referenced and executed across the threads.

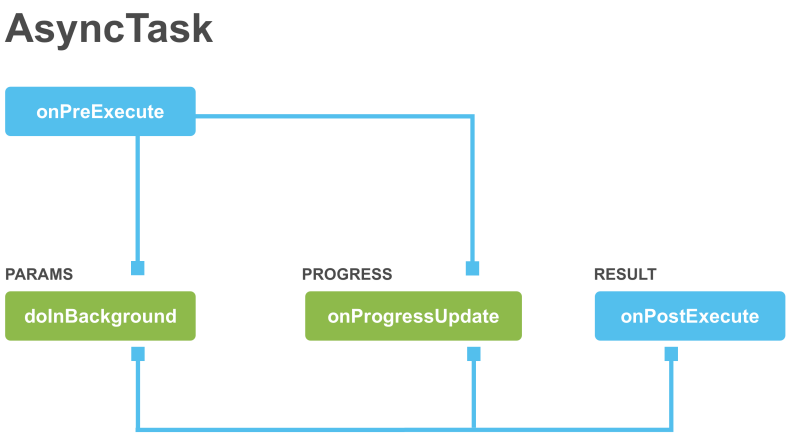
Taking that into account, engineers working on the Android framework came up with a solution to handle this scenario of doing work on the background thread to then publish it to the UI thread; it is called **AsyncTask**.

AsyncTask

In Java, you usually put the code you want to run asynchronously into the `run` method of a class, which implements the `Runnable` interface. This works well if all you need to do is offload work off to another thread. However, it becomes cumbersome when you need to relay the results of that thread back to the UI thread.

When Google adopted Java for Android, it released a new type of class called `AsyncTask` that made it easier to offload long-running tasks to a background thread, then update the UI thread with the result if there was one. Using `AsyncTask` instances certainly was easier than `Runnable`, but it came with its own set of issues.

AsyncTask is the most basic Android component for threading. It's simple to use and can be good for basic scenarios. The only important thing you should know here is that only one method of this class is running on another thread: `doInBackground`. The other methods are running on UI thread.



AsyncTask Process Flow

Here is a sample usage:

```
class ExampleActivity : Activity() {

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    MyTask().execute(url)
  }
```

```
  private inner class MyTask : AsyncTask<String, Void, String>() {

    override fun doInBackground(vararg params: String): String {
      val url = params[0]
      return doSomeWork(url)
    }

    override fun onPostExecute(result: String) {
      super.onPostExecute(result)
      // do something with result
    }
  }
}
```

To see a working example, in your **MainActivity.kt** file under the `onCreate()` function, set `methodToUse = MethodToDownloadImage.AsyncTask`. This makes sure that, when the button is clicked, the method `getImageUsingAsyncTask()` is called. Here is the method definition:

```
fun getImageUsingAsyncTask() {
  // Download image
  val myAsyncTask = GetImageAsyncTask(imageDownloadListener)
  myAsyncTask.execute()
}
```

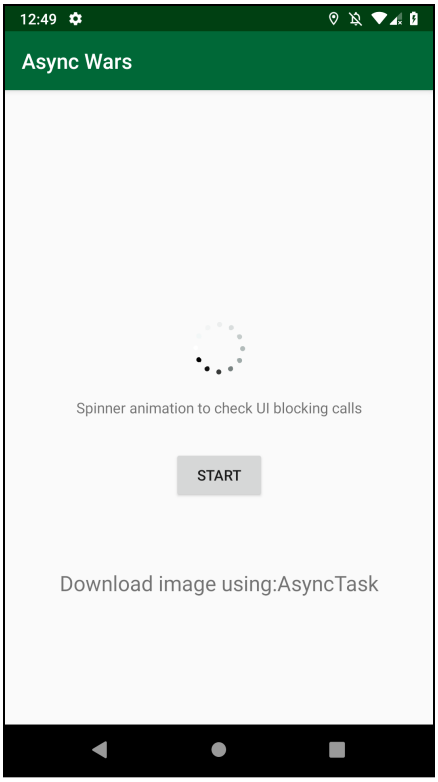
Here, `GetImageAsyncTask` has the below implementation:

```
class GetImageAsyncTask(val imageDownloadListener: ImageDownloadListener)
:
    AsyncTask<String, Void, Bitmap>() {


    // This executes on the background thread
    override fun doInBackground(vararg p0: String?): Bitmap? {
        // Download Image
        return DownloaderUtil.downloadImage()
    }

    // This executes on the UI thread
    override fun onPostExecute(bmp: Bitmap?) {
        super.onPostExecute(bmp)
        if (isCancelled) {
            return
        }

        // Pass it to the listener
        imageDownloadListener.onSuccess(bmp)

        // Cancel this async task after everything is done.
        cancel(false)
    }
}
```

`ImageDownloadListener` is used to setup a listener, which will return the bitmap once it is downloaded. In the `MainActivity.kt`, an instance of this is created and used inside the `getImageUsingAsyncTask()` method while creating the `GetImageAsyncTask`, which, in turn, is used to update the UI:

```
private val imageDownloadListener = object : ImageDownloadListener {
  override fun onSuccess(bitmap: Bitmap?) {
    // Update UI with downloaded bitmap
    imageView?.setImageBitmap(bitmap)
  }
}
```

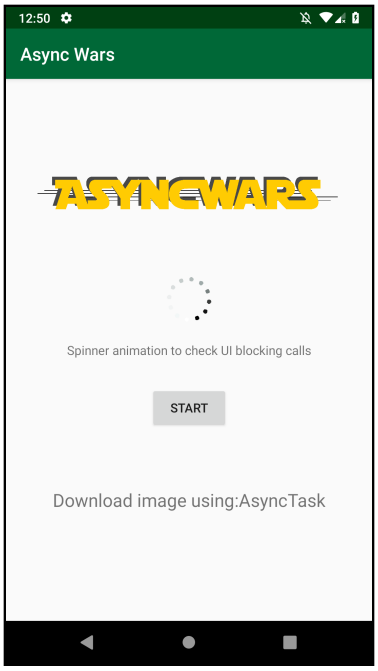
Run the app.



Download image using AsyncTask

When you click the Start button, you will see that the image is downloaded and displayed in the `ImageView` without blocking the UI; the spinner animates while the image is being downloaded.

The image downloads:



Download image using AsyncTask

The AsyncTask defines some callback methods in order to simplify the way cancellation and the progression of the task are communicated to the UI; however, it does not play out well when it comes to doing complex operations based on an Android component's lifecycle. It is actually unaware of Activity lifecycle; in other words, if the activity is destroyed, the AsyncTask doesn't know about it in the `onPostExecute()` method unless you tell it.

It is worth noting that even something as simple as screen rotation can cause the activity to be destroyed. Also, canceling an AsyncTask just puts it in a canceled state — it's up to you to check whether it's been canceled and halt operations.

Handlers

Handler is part of the **HaMeR Framework** (`Handler`, `Message` & `Runnable`), which is the recommended framework for communication between threads in Android. This is the one used, under the hood, by the `AsyncTask` class.

As you have seen in the previous chapters, threads can share data using queues, which usually have a **producer** and a **consumer**. The producer is the object that puts data into the queue, and the consumer is the object that reads those data from the queue when available.

If the producer runs into thread A and the consumer into thread B, you understand that you can use the queue as a communication channel between different threads. This is basically the idea behind the HaMeR framework. The queue is actually a `MessageQueue`, and the data you pass are encapsulated into a `Message` object. Each `Message` can contain some data or the reference to a `Runnable` implementation that defines the code to execute into the thread of the consumer.

If you had to implement the consumer of the queue on your own, you would probably implement it with a cycle that waits for a `Message` and, when available, reads and uses the information into it or else run the code into the `Runnable` object if available. That cycle would be into the `run` implementation of the related `Thread` class. Android defines this cycle into a class called `Looper`. It's important to note that you decide the destination thread putting the message into the related queue. This also implies that there is only one `Looper` per `Thread`.

What's the role of the `Handler` in all of this? Each `Handler` instance is associated with a specific `Thread` through its `Looper`. You can bind a `Looper` to a `Handler`, passing it as the constructor parameter or by simply creating the `Handler` instance into the `Looper`'s thread. You can then use a `Handler` in two different ways:

1. You can use it in order to put a `Message` into the queue that its `Looper` will read into the associated `Thread`.
2. You can also use `Handler` as the object containing the actual consumer logic. In this case, you usually override the `handleMessage(Message?)` method like this:

```
object handler: Handler(){
  override fun handleMessage(msg: Message?) {
    // Consume the message
  }
}
```

This is possible because, when a thread reads a message from its queue, it delegates the actual usage of the data to its handlers.

How can you use all this in order to send data from a background thread to the UI? You just need a `Handler` associated with the main looper that is available calling `Looper.getMainLooper()` and then post an action as a `Runnable`:

```
val runnable = Runnable {
    // update the ui from here
}

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

You can summarize the different objects responsibilities as:

- **Looper**: Runs a loop on its `Thread`, waiting for `Message` instances on its `MessageQueue`.
- **MessageQueue**: Holds a list of messages for a given `Thread`.
- **Handler**: Allows the sending and processing of `Message` and `Runnable` to the `MessageQueue`. It can be used to send and process messages between threads.
- **Message**: Contains a description and data that can be created and sent using a `Handler`.
- **Runnable**: Represents a task to be executed.

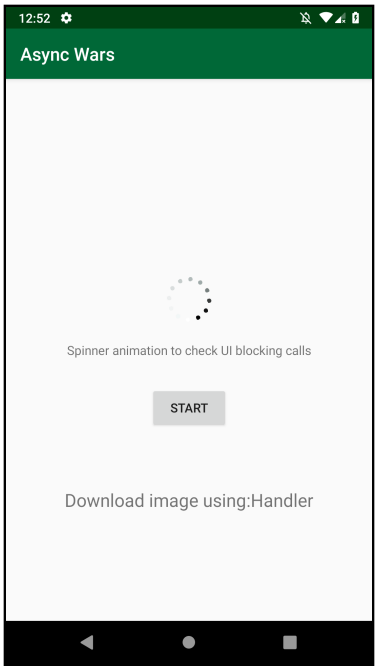
Handler is then the HaMeR workhorse. It's responsible for sending `Message` (data message) and post `Runnable` (task message) objects to the `MessageQueue` associated with a `Thread`.

After delivering the tasks to the queue, the handler receives the objects from the looper and processes the messages at the appropriate time. It can be used to send or post some message or runnable objects between threads, as long as such threads share the same process. Otherwise, it will be necessary to use an **Inter Process Communication** (IPC) mechanism, like the `Messenger` class or some **Android Interface Definition Language** (AIDL) implementation.

To see a working example, in your **MainActivity.kt** file under the `onCreate()` function, set `methodToUse = MethodToDownloadImage.Handler`. This makes sure that, when you click the button, the method `getImageUsingHandler()` is called. Here is the method definition:

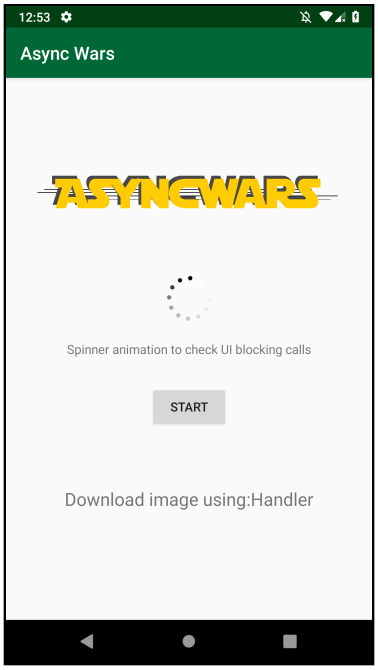
```
fun getImageUsingHandler() {
    // Create a Handler using the main Looper
    val uiHandler = Handler(Looper.getMainLooper())

    // Create a new thread
    Thread {
      // Download image
      val bmp = DownloaderUtil.downloadImage()

      // Using the uiHandler update the UI
      uiHandler.post {
        imageView?.setImageBitmap(bmp)
      }
    }.start()
  }
```

Run the app.



Download image using Handler

When you click the Start button, you will see that the image is downloaded and displayed in the `ImageView` without blocking the UI; the spinner animates while the image is being downloaded.



Download image using Handler

HandlerThreads

The UI thread already comes with a `Looper` and a `MessageQueue`. For other threads, you need to create the same objects if you want to leverage the HaMeR framework. You can do this by extending the `Thread` class as follow:

```
// Preparing a Thread for HaMeR
class MyLooperThread : Thread() {

  lateinit var handler: Handler

  override fun run() {
    // adding and preparing the Looper
    Looper.prepare()

    // the Handler instance will be associated with Thread's Looper
    handler = object : Handler() {
      override fun handleMessage(msg: Message) {
        // process incoming messages here

      }
    }

    // Starting the message queue loop using the Looper
    Looper.loop()
  }
}
```

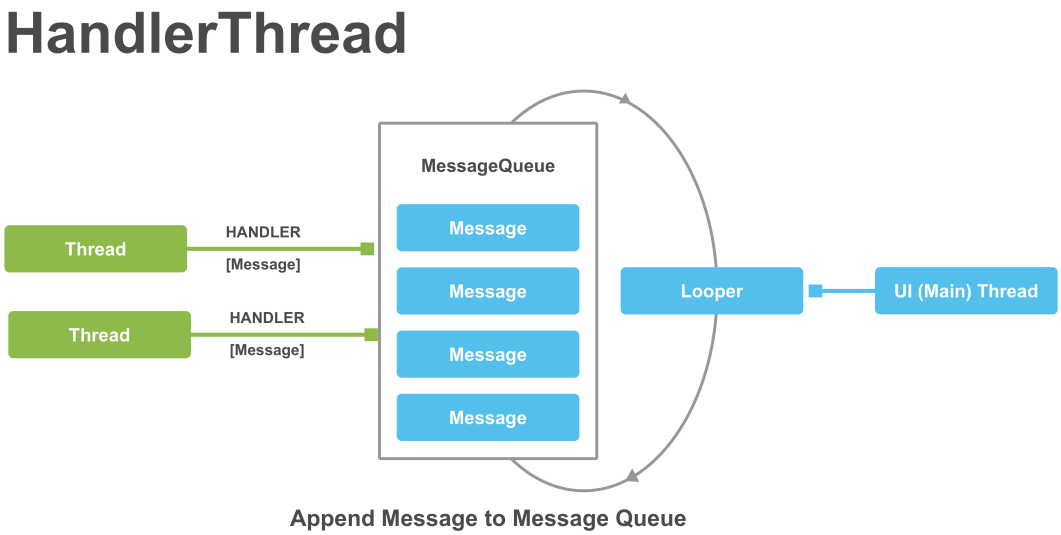
However, it's more straightforward to use a helper class called `HandlerThread`, which creates a `Looper` and a `MessageQueue` for you. Check out the implementation of `getImageUsingHandlerThread()` method inside `MainActivity.kt` of the starter app:

```
var handlerThread: HandlerThread? = null
fun getImageUsingHandlerThread() {
  // Download image
  // Create a HandlerThread
  handlerThread = HandlerThread("MyHandlerThread")

  handlerThread?.let{
    // Start the HandlerThread
    it.start()
    // Get the Looper
    val looper = it.looper
    // Create a Handler using the obtained Looper
    val handler = Handler(looper)
    // Execute the Handler
    handler.post {
      // Download Image
      val bmp = DownloaderUtil.downloadImage()

      // Send local broadcast with the bitmap as payload
      BroadcasterUtil.sendBitmap(applicationContext, bmp)
    }
  }
```

```
}

override fun onDestroy() {
  super.onDestroy()

  // Quit and cleanup any instance of dangling HandlerThread
  handlerThread?.quit()
}
```

Here, you create an instance of the `HandlerThread`, passing a name that is very useful for debugging purposes. The `HandlerThread` extends the `Thread` class and you have to start it in order to use its `Looper`. You then access `looper` property and pass it as the constructor paramenter of the `Handler`. You can then use the handler that you have created for sending `Runnable` objects to the `HandlerThread`. All of the code you encapsulate into the `Runnable` object will be then executed into the `HandlerThread`.

Note: It is important that you call `quit()` on the `HandlerThread` instance when the work is done, possibly in the `onDestroy()` of the activity so as to release resources it would be holding.

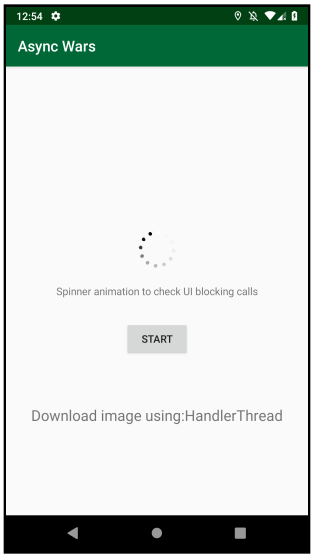


HandlerThread

Note: When the `Activity` is destroyed, it's important to terminate the `HandlerThread`. This also terminates the `Looper`.

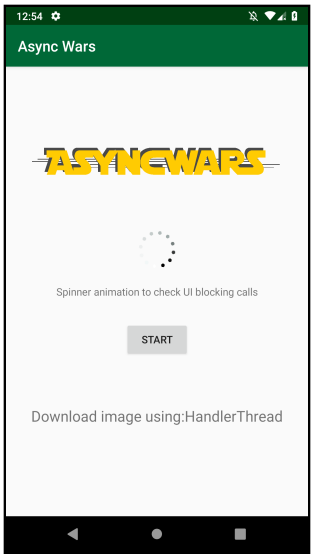
To see a working example, in your **MainActivity.kt** file under the `onCreate()` function, set `methodToUse = MethodToDownloadImage.HandlerThread`. This makes sure that, when you click the button, the method `getImageUsingHandlerThread()` is called.

Run the app.



Download image using HandlerThread

When you click the Start button, you will see that the image is downloaded and displayed in the `ImageView` without blocking the UI; the spinner animates while the image is being downloaded.



Download image using HandlerThread

Service

The definition of **component** implies the existence of a **container**. You usually describe all your components to the container using some document; the container will create, suspend, resume and destroy components depending on the state of the application or on the available resources on the device. You would say that the container is responsible for the component's lifecycle. You can apply the same concept to Android when you describe all your components to the system using the `AndroidManifest.xml` file.

In the example you've seen earlier, the component is an `Activity` whose lifecycle depends mainly on the application usage and on the available resources. For instance, when the user rotates the device, the activity is destroyed and then re-created — unless you don't configure differently. What happens when you start a task in the background from an `Activity` and then rotate the device? In the case of the `HandlerThread`, you should make it aware of the lifecycle and cancel the tasks, if any, and execute them again. This is not always the best solution — especially in cases of very long tasks like downloading a file.

For situations like these, Android provides a different component whose lifecycle doesn't depend on what's happening on the UI but that can only depend on the available resources: the **service**. It's an Android component and, as such, you have to declare it into the `AndroidManifest.xml` file, but it has lifecycle different from the activities lifecycle.

The `Service` is a component that you can use as the owner of a very long task because the system will change its state only if it really needs resources. You can think of it as a safe place to put your long-running code. It's important to note that a service does not create its own thread and does not run in a separate process unless you explicitly say so.

A sample usage:

```
class ExampleService : Service() {

  fun onStartCommand(intent: Intent, flags: Int, startId: Int): Int {
    doSomeLongProccesingWork()
    return START_NOT_STICKY
  }

  fun onBind(intent: Intent): IBinder? {
    return null
  }

  fun doSomeLongProccesingWork(){
    // Do some work
```

```
      // Stop service when required
      stopSelf()
    }
  }
```

It is your responsibility to stop a `Service` when its work is complete by calling either the `stopSelf()` or the `stopService()` method. The `Service` doesn't know what is going on in the code running in your thread or executor task — it is your responsibility to let it know when you've started and when you've finished.

A basic service can exist in two flavors:

- A **started** service is initiated by a component in your application and remains active in the background of the device, even if the original component is destroyed. When a started service finishes running its task, the service will stop itself. A standard started service is generally used for long-running background tasks that do not need to communicate with the rest of the app.
- A **bound** service provides a client/server communication paradigm. The service is usually thought of as the server and an Android context, usually an activity, is the client. This type of service is similar to a started service, and it also provides callbacks for various app components that can bind to it. When all bound components have unbound themselves from the service, the service will stop itself.

It is important to note that these two ways to run a service are not mutually exclusive so you can start a service that will run indefinitely and can have components bound to it.

However, since the **Api Level 26 (Android 8.0)**, the `Service` usage as you might know it today, has been deprecated. It is no longer allowed to fulfill its primary purpose, namely to execute the long-running task in the background. Calling `startService()` method when your app has been put in background throws an `IllegalStateException`. The only way one can use services now is as a **foreground service**.

Intent service

As stated previously, `Service` components, by default, are started in the main thread like any other Android component. If you need the service to run a task as a background task, then it's up to you to create a separate thread and move your work to that thread. The Android frameworks also offer sub-class of `Service` that can do all the threading work for you: `IntentService`.

It runs on a separate thread and stops itself automatically after it completes its work. `IntentService` is usually used for short tasks that don't need to be attached to any UI. Since `IntentService` doesn't attach to any activity and it runs on a non-UI thread, it serves the need perfectly. Moreover, `IntentService` stops itself automatically, so there is no need to manually manage it, either.

One of the biggest issues with a standard **started** service is that it cannot handle multiple requests at a time, but that is not the case with an `IntentService`. It creates a default worker thread for executing all intents that are received in `onStartCommand()`, so all operations can happen off the main thread. It then creates a work queue for sending each intent to `onHandleIntent()` one at a time so that you don't need to worry about multi-threading issues.

Essentially, there is always only one instance of your `IntentService` implementation at any given time and it has only one `HandlerThread`. This means that if you need more than one thing to happen at the same time, `IntentServices` may not be a good option.

Sample usage:

```
// Required constructor with a name for the service
class MyIntentService : IntentService("MyIntentService") {

  override fun onHandleIntent(intent: Intent?) {
    //Perform your tasks here
    doSomeWork();
  }
}
```

To see a working example, in your **MainActivity.kt** file under the `onCreate()` function, set `methodToUse = MethodToDownloadImage.IntentService`. This makes sure that when the button is clicked, the method `getImageUsingIntentService()` is called. Here is the method definition:

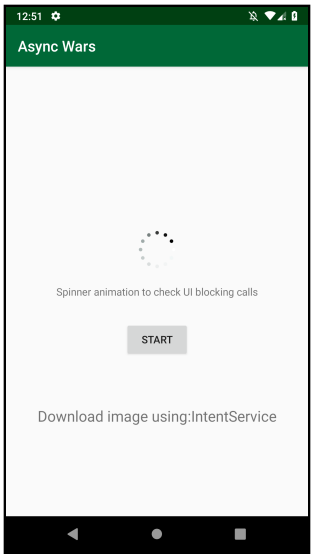
```
fun getImageUsingIntentService() {
  // Download image
  val intent = Intent(this@MainActivity, MyIntentService::class.java)
  startService(intent)
}
```

Here, `MyIntentService` has the below implementation:

```
// Required constructor with a name for the service
class MyIntentService : IntentService("MyIntentService") {

  override fun onHandleIntent(intent: Intent?) {
    // Download Image
    val bmp = DownloaderUtil.downloadImage()

    // Send local broadcast with the bitmap as payload
```

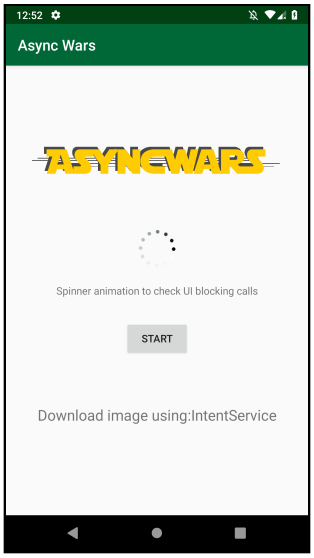
```
      BroadcasterUtil.sendBitmap(applicationContext, bmp)
    }
  }
```

Here, `BroadcasterUtil` is a utility class that internally uses LocalBroadcastManager. It is used here to easily send the image back to the UI thread. You will learn more about this process in the next section. Run the app.



Download image using IntentService

When you click the Start button, you will see that the image is downloaded and displayed in the `ImageView` without blocking the UI; the spinner animates while the image is being downloaded.



Download image using IntentService

Sending data from a Service to the UI

You learned that a started `Service` is an Android component that is not bound to the UI. If you need to send some data from a service to a different component, like an `Activity`, you need some other mechanisms like the `LocalBroadcastManager` that you used via the `BroadcasterUtil` in the previous example. You can see how to send data from a service in the `onHandleIntent()` method of the `MyIntentService` class:

```
override fun onHandleIntent(intent: Intent?) {
  // Download Image
  val bmp = DownloaderUtil.downloadImage()

  // Send local broadcast with the bitmap as payload
  BroadcasterUtil.sendBitmap(applicationContext, bmp)
}
```

Here, `sendBitmap(applicationContext, bmp)` is a method defined inside `BroadcasterUtil` class with the below implementation:

```
/**
  * Send local broadcast with the bitmap as payload
  * @param context Context
  * @param bmp Bitmap
  * @return Unit
  */
fun sendBitmap(context: Context, bmp: Bitmap?) {
    val newIntent = Intent()
    bmp?.let {
        newIntent.putExtra("bitmap", it)
        newIntent.action = MainActivity.FILTER_ACTION_KEY

LocalBroadcastManager.getInstance(context).sendBroadcast(newIntent)
    }
}
```

As you can see, it uses `LocalBroadcastManager` to send a broadcast using an intent, which has a payload of the passed bitmap. A `LocalBroadcastManager` needs a `BroadcastReceiver` to be registered via using the `registerReceiver()` method. In the starter app, there is an implementation for a `BroadcastReceiver` already provided named `MyBroadcastReceiver`, which has the below implementation:

```
class MyBroadcastReceiver(val imagdeDownloadListener:
ImageDownloadListener) : BroadcastReceiver() {
  override fun onReceive(context: Context, intent: Intent) {
      val bmp = intent.getParcelableExtra<Bitmap>("bitmap")

      // Pass it to the listener
      imagdeDownloadListener.onSuccess(bmp)
  }
}
```

`ImageDownloadListener` is used here to set up a listener, which will return the bitmap once it is downloaded. In the `MainActivity.kt`, you've already created an instance of this during the AsyncTask section of this chapter.

`BroadcasterUtil` abstracts register and unregister of `MyBroadcastReceiver` for the `LocalBroadcastManager` by defining helper methods:

```
/**
  * Register Local Broadcast Manager with the receiver
  * @param context Context
  * @param myBroadcastReceiver MyBroadcastReceiver
  * @return Unit
  */
fun registerReceiver(context: Context, myBroadcastReceiver:
MyBroadcastReceiver?) {
    myBroadcastReceiver?.let {
        val intentFilter = IntentFilter()
        intentFilter.addAction(MainActivity.FILTER_ACTION_KEY)
        LocalBroadcastManager.getInstance(context).registerReceiver(it,
intentFilter)
    }
}

/**
  * Unregister Local Broadcast Manager from the receiver
  * @param context Context
  * @param myBroadcastReceiver MyBroadcastReceiver
  * @return Unit
  */
fun unregisterReceiver(context: Context, myBroadcastReceiver:
MyBroadcastReceiver?) {
    myBroadcastReceiver?.let {
        LocalBroadcastManager.getInstance(context).unregisterReceiver(it)
    }
}
```

You use these helper methods later to register and unregister an instance of `MyBroadcastReceiver` to the `LocalBroadcastManager` in `onStart()` and `onStop()` respectively, of the `MainActivity`:

```
// ----------- Lifecycle Methods -----------//
override fun onStart() {
  super.onStart()
  BroadcasterUtil.registerReceiver(this, myReceiver)
}

override fun onStop() {
  super.onStop()
  BroadcasterUtil.unregisterReceiver(this, myReceiver)
}
```

Important points to note, here:

- If there's no `BroadcastReceiver` registered, there won't be any update in the UI.
- The thread that will perform the `ImageView` update is the UI thread.
- IntentService uses HandlerThread internally.

Executors

You've seen that you can encapsulate code into a `Runnable` implementation in order to eventually run it in some given `Thread`. Every object that can execute what's defined into a `Runnable` can be abstracted using the `Executor` interface, introduced in Java 5.0 as part of the concurrent APIs.

```
interface Executor {
    fun execute(command: Runnable)
}
```

You can execute a `Runnable` in many different ways. You can, for instance, simply invoke directly the `run()` method or pass the `Runnable` object as constructor parameter of the `Thread` class and start it, as seen previously. In the former case, you're executing the runnable code in the caller thread. In the latter, you're executing the same code into a different thread. This depends on the particular `Executor` implementation.

Creating a thread is very simple in the code but expensive in practice. Every time you create a `Thread` instance you need to request resources to the operative system and every time the thread completes its job — when its `run()` method ends — it must be collected as garbage. The typical solution, in this case, is the usage of a **pool of threads**, which, on the other hand, needs some kind of **lifecycle**.

The pool needs to be initialized with a minimum number of threads. When the application ends, the pools should shut down and release all the resources. Even when the pool is active, you can have a different policy for the minimum number of instances of thread to keep alive or how to manage the creation of new instances when needed. You could limit the number of threads forcing the client to wait, or create a new thread every time you need. This is something more than a simple `Executor` that concurrent APIs abstracts with the `ExecutorService` interface.

The `ExecutorService` is then the abstraction for specific `Executor`, which needs to be initialized and shut down to allow for the execution of `Runnable` objects in an efficient and optimized way. The way this is happening depends on the specific implementation. One of the most important is `ThreadPoolExecutor`. It manages a pool of worker threads and a queue of tasks to execute.

Depending on the configured policy, it reuses an available thread or creates a new one in order to consume the tasks from a queue.

The concurrent APIs provide different implementations that are available through some **static factory methods** of the `Executors` class. The most common are `Executors.newSingleThreadExecutor()`, which create an executor that will process a single task at a time, and `Executors.newFixedThreadPool(N)`, which creates an executor with an internal pool of N threads.

It's important to note that an `ExecutorService` also provides the option of executing `Callable<T>` implementations. While the `Runnable` interface defines a `run()` method, which returns `Unit`, a `Callable<T>` is a generic interface, which defines the `call()` method that returns an object of type `T`:

```
interface Callable<T> {
    fun call(): T
}
```

You can think of a `Callable<T>` as a `Runnable` that actually returns an object of type T at the end of the task. You can ask the `ExecutorService` to run the given `Callable<T>` using the `invoke()` method, getting a `Future<T>` in return. The `Future<T>` provides a `get()` method, which blocks until the result of type `T` is available or throws an exception in case of error or interruption.

Sample usage:

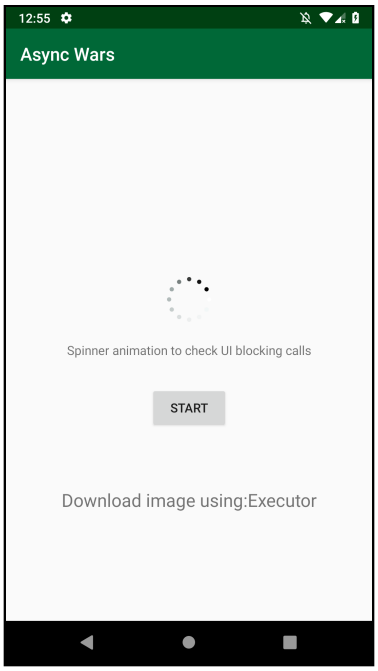
```
val executor = Executors.newFixedThreadPool(4)
(1..10).forEach {
  executor.submit {
    print("[Iteration $it] Hello from Kotlin Coroutines! ")
    println("Thread: ${Thread.currentThread()}")
  }
}
```

To see a working example, in your **MainActivity.kt** file under the `onCreate()` function, set `methodToUse = MethodToDownloadImage.Executor`. This makes sure that, when you click the button, the method `getImageUsingExecutors()` is called. Here is the method definition:

```
fun getImageUsingExecutors() {
  // Download image
  val executor = Executors.newFixedThreadPool(4)
  executor.submit(myRunnable)
}
```

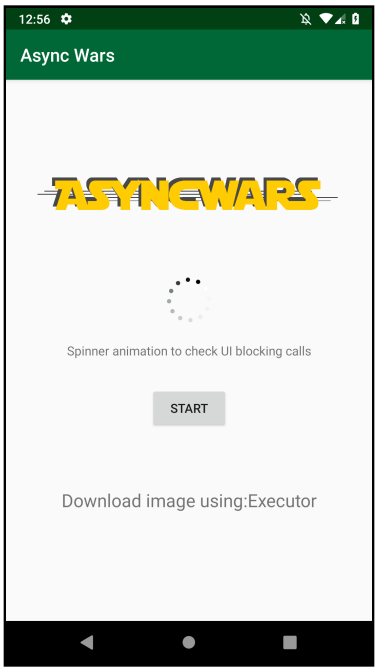
Here, `myRunnable` in the `MainActivity.kt` is a instance of `MyRunnable` class, which you've already created during the Thread section of this chapter.

Run the app.



Download image using Executor

When you click the Start button, you will see that the image is downloaded and displayed in the `ImageView` without blocking the UI; the spinner animates while the image is being downloaded.



Download image using Executor

The main advantages of using `ThreadPoolExecutor` in an Android application are:

- Powerful task execution framework as it supports task addition in a queue, task cancellation and task prioritization.
- Reduces the overhead associated with thread creation as it manages a required number of threads in its thread pool.
- Reduces boilerplate code as it abstracts most of the codebase behind factory method with sane defaults.

However, although `ExecutorService` implementations provide an optimized usage of threads in terms of creation and reuse, they don't solve the problems related to context switching between threads.

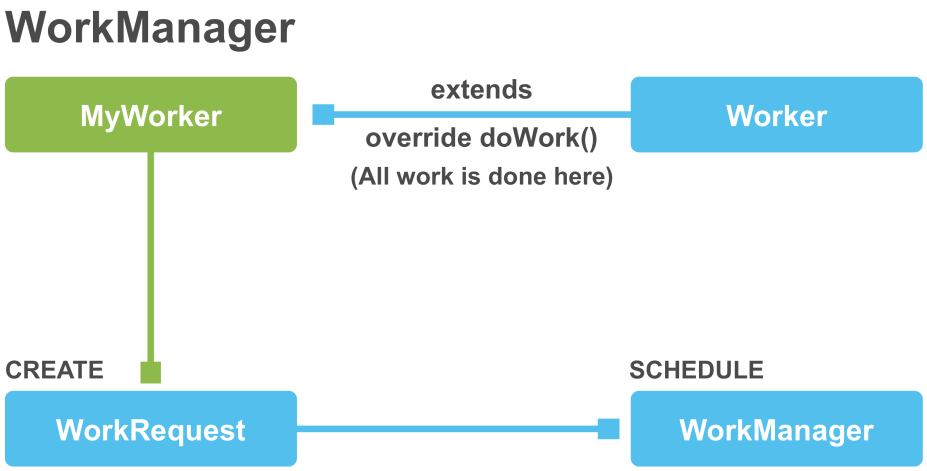
WorkManager

Announced at Google I/O 2018 as part of Jetpack, **WorkManager** aims to simplify the developer experience by providing a first-class API for system-driven background processing. The WorkManager API makes it easy to specify deferrable, asynchronous tasks and when they should run. It is intended for background jobs that should run even if the app is no longer in the foreground. Where possible, it delegates its work to a JobScheduler, Firebase JobDispatcher, or Alarm Manager + Broadcast receivers. If your app is in the foreground, it will even try to do the work directly in your process. The task is still guaranteed to run, even if your app is force-quit or the device is rebooted.

WorkManager chooses the appropriate way to run your task based on such factors as the device API level and the app state.

By default, WorkManager runs each task immediately, but you can also specify the conditions the device needs to fulfill before the task can proceed, including network conditions, charging status and the amount of storage space available on the device. If WorkManager executes one of your tasks while the app is running, it can run your task in a new thread in your app's process.

If your app is not running, WorkManager chooses an appropriate way to schedule a background task — depending on the device API level and included dependencies. You don't need to write device logic to figure out what capabilities the device has and choose an appropriate API; instead, you can just hand your task off to WorkManager and let it choose the best option.



WorkManager Process Flow

Sample usage:

```
// A simple Worker
class DoSomeWorker : Worker() {
    // This method will run in background thread and WorkManger
    // will take care of it
    override fun doWork() : WorkerRequest() {
        doSomeWork()
        return WorkResult.SUCCESS
    }
}

// Usage
// Create the request
val request : WorkRequest = OneTimeWorkRequestBuilder<DoSomeWorker>()
                            .build()
// Enqueue the request
val workManager : WorkManager = WorkManager.getInstance()
workManager.enqueue(request)
```

In short, the WorkManager is another library that is trying to solve the old problem of executing long-running jobs on the Android platform. It delegates the logic to different components that are available only on specific versions of the platform. If you accept to use this library, you also accept all the fallbacks and workarounds used to enable support for older platforms/APIs. WorkManager is seen as the third attempt by Google to solve the job management on Android Platform and probably not the last.

RxJava + RxAndroid

Reactive programming is an asynchronous programming paradigm concerned with data streams and the propagation of change. The essence of reactive programming is the **observer pattern**.

> **Note**: The observer pattern is a software design pattern wherein data sources or streams, called observables, emit data and one or more observers, who are interested in getting the data, subscribe to the observable.

In reactive programming, you are allowed to create data streams from anything including `Array`, `ArrayList`, etc. These data streams can be observed, modified, filtered or operated upon. You can use a stream as an input to another one. You can even use multiple streams as inputs to another stream. You can merge two streams. You can filter a stream to get another one that has only those events you are interested in. You can map data values from one stream to another one. A typical data stream can emit three different values: one on when the event occurs, one on when the error occurs or one on when the event is completed.

RxJava is a library that makes it easier for you to implement reactive programming principles on any JVM-based platform, including Android. To manage threads, RxJava has a helper class called `Schedulers`. Schedulers are how you tell where the observer and observables should run.

Some general use Schedulers to observe:

- `Schedulers.computation()`: Used for CPU intensive tasks.
- `Schedulers.io()`: Used for IO bound tasks.
- `Schedulers.from(Executor)`: Used with custom ExecutorService.
- `Schedulers.newThread()`: It always creates a new thread when a worker is needed.

This is where RxAndroid library comes into the picture, which plays a major role in supporting multi-threading concepts in Android applications. It provides a `Scheduler` that schedules on the main thread or any given `Looper`.

Sample usage:

```
Observable.just("Hello", "from", "RxJava")
        .subscribeOn(Schedulers.newThread())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(/* an Observer */);
```

This will execute the `Observable` on a new thread and emit results through `onNext()` on the main thread.

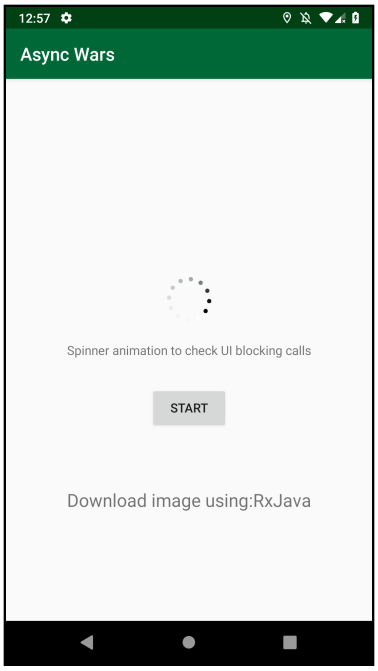
To see a working example, in your **MainActivity.kt** file under the `onCreate()` function, set `methodToUse = MethodToDownloadImage.RxJava`. This makes sure that when you click the button, the method `getImageUsingRx()` is called. Here is the method definition:

```
var single: Disposable? = null
fun getImageUsingRx() {
  // Download image
  single = Single.create<Bitmap> { emitter ->
    DownloaderUtil.downloadImage()?.let { bmp ->
      emitter.onSuccess(bmp)
    }
  }.observeOn(AndroidSchedulers.mainThread())
    .subscribeOn(Schedulers.io())
    .subscribe { bmp ->
      // Update UI with downloaded bitmap
      imageView?.setImageBitmap(bmp)
    }
}

override fun onDestroy() {
  super.onDestroy()

  // Cleanup disposable if it was created i.e not null
  single?.dispose()
}
```

Note: It is important that you call `dispose()` on the `Single` instance when the work is done, possibly in the `onDestroy()` of the activity so as to release resources it would be holding and close the stream.

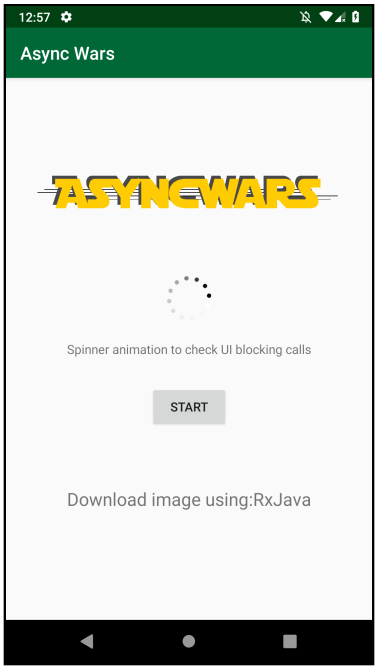
Also note that the topic of reactive extensions is pretty vast; covering the mechanics of its functionalities is out of the scope of this book. The example shown here is for comparison purpose only and you can learn more in "Chapter 14: Coroutines and RxKotlin Comparison."

Run the app.



Download image using RxJava

When you click the Start button, you will see that the image is downloaded and displayed in the `ImageView` without blocking the UI; the spinner animates while the image is being downloaded.



Download image using RxJava

Although reactive programming is a compelling tool and solves a lot of complex concurrency problems, the learning curve for RxJava is very steep and complex. It is a different approach towards programming and can lead to some confusion when programming larger apps.

Coroutines

Now that you have a clear idea about various ways of doing asynchronous work in Android, as well as the pros and cons, let's come back to Kotlin coroutines. Kotlin coroutines are a way of doing things asynchronously in a sequential manner. Creating coroutines is cheap versus creating threads.

> **Note**: Coroutines are completely implemented through a compilation technique (no support from the VM or OS side is required), and suspension works through code transformation.

Coroutines are based on the idea of suspending functions: functions that can stop the execution when they are called and make it continue once it has finished running their own task. Enabling Kotlin coroutines in Android involves just a few simple steps. To show how easy it is to enable coroutines, head back to the starter project and add the Android coroutine library dependency into your app's **build.gradle** file under `dependencies` block, replacing the line `// TODO: Add Kotlin Coroutine Dependencies here` with the following:

```
dependencies {
  ..
  // Coroutines
  final def coroutineVer = "1.0.1"
  implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:
$coroutineVer"
  implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:
$coroutineVer"
}
```

> **Note**: In order to use the stable Coroutines v1.0.1, the accompanying Koltin version should be v1.3.0 and above. Make sure that the Kotlin standard library is at least v1.3.0

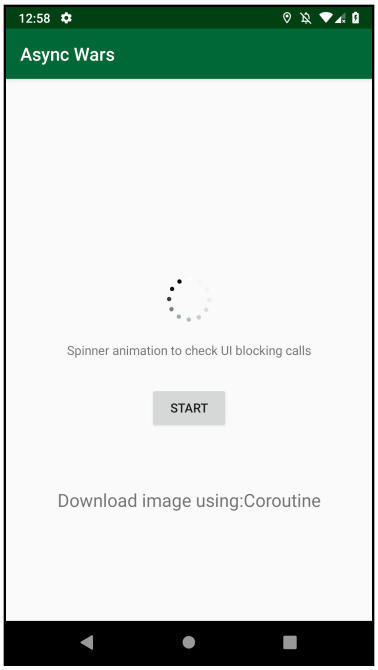
Next, inside your **MainActivity.kt** file, add the implementation for the method `getImageUsingCoroutines()` by replacing `// TODO: add implementation here` with the below code snippet:

```
GlobalScope.launch {
  // Download Image in background
  val deferredJob = async(Dispatchers.IO) {
    DownloaderUtil.downloadImage()
  }
  withContext(Dispatchers.Main) {
    val bmp = deferredJob.await()
    // Update UI with downloaded bitmap
    imageView?.setImageBitmap(bmp)
  }
}
```

To see a working example, in your **MainActivity.kt** file under the `onCreate()` function, set `methodToUse = MethodToDownloadImage.Coroutine`.

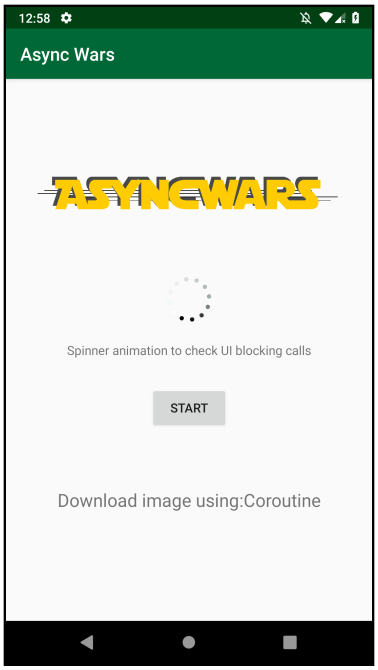
This makes sure that when you click the button, the method `getImageUsingCoroutines()` is called.

Run the app.



Download image using Coroutine

When you click the Start button, you will see that the image is downloaded and displayed in the `ImageView` without blocking the; the spinner animates while the image is being downloaded.



Download image using Coroutine

A lot has already been explained about the mechanics of Kotlin coroutines in the previous chapters; in the subsequent chapters, you will mostly cover the usage of Kotlin coroutines in the Android apps.

Introducing Anko

While Kotlin does remove much of the verbosity and complexity typically associated with Java, no programming language is perfect and, thus, libraries that build on top of the language are born. Anko is one such library that uses Kotlin and provides a lot of extension functions to make your Android development easier.

> **Note**: That's how Anko got its name: (An)droid (Ko)tlin.

Anko was originally designed as a single library. As the project grew, adding Anko as a dependency began to have a significant impact on the size of the APK (Android Application Package).

Today, Anko is split across several modules:

- **Commons**: Helps you perform the most common Android tasks, including displaying dialogs and launching new Activities.
- **Layouts**: Provides a Domain Specific Language (DSL) for defining Android layouts.
- **SQLite**: A query DSL and parser that makes it easier to interact with SQLite databases.
- **Coroutines**: Supplies utilities based on the `kotlinx.coroutines` library.

You can see the differences in a sample comparison, below.

Using language provided coroutines:

```
button.setOnClickListener {
  launch(UI){
    val userId = fetchUserString("user_id_1").await()
    val user = deserializeUser(userId).await()
    showUserData(user)
  }
}
```

Using an Anko-provided coroutine helper:

```
button.onClick {
  val userId= bg { fetchUserString("user_id_1").await() }
  val user = bg { deserializeUser(userId).await() }
  showUserData(user)
}
```

`onClick` and `bg` are some of many helper functions Anko provides for making the process of handling coroutines even simpler, which will be covered in depth in later chapters.

Key points

- Android is inherently **asynchronous and event-driven**, with strict requirements as to which thread certain things can happen on.
- The **UI thread** — a.k.a. main thread — is responsible for interacting with the UI components and is the most important thread of an Android application.
- Almost all code in an Android application will be executed on the **UI thread** by default; blocking it would result in a non-responsive application state.

- **Thread** is an independent path of execution within a program allowing for asynchronous code execution, but it is highly complex to maintain and has limits on usage.
- **AsyncTask** is a helper class which simplifies asynchronous programming between UI thread and background threads on Android. It does not work well with complex operations based on Android Lifecycle.
- **Handler** is another helper class provided by Android SDK to simplify asynchronous programming, but requires a lot of moving parts to set up and get running.
- **HandlerThread** is typically a thread that is ready to receive a `Handler` because it has a `Looper` and a `MessageQueue` built into it.
- **Service** is a component that is useful for performing long (or potentially long) operations without any UI, and it runs in the main thread of its hosting process.
- **IntentService** is a service that runs on a separate thread and stops itself automatically after it completes its work; however, it cannot handle multiple requests at a time.
- **Executors** is a manager class that allows running many different tasks concurrently while sharing limited CPU time, used mainly to manage thread(s) in an efficent manner.
- **WorkManager** is a fairly new API developed as part of JetPack libraries provided by Google, which makes it easy to specify deferrable, asynchronous tasks and when they should run.
- **RxJava + RxAndroid** are libraries that make it easier to implement reactive programming principles in the Android platform.
- **Coroutines** make asynchronous code look like synchronous and work pretty well with Android platform out of the box.
- **Anko** is a library that uses Kotlin and provides a lot of extension functions to make our Android development easier.

Where to go from here?

Phew! That was a lot of background on asynchronous programming in Android! But the good thing is that you made it!

In the upcoming chapters, you dive deeper into how you can leverage coroutines in Android apps to handle async operations while keeping in sync with various nuances of the Android platform, such as respecting lifecycles of an app and efficient context switching to facilitate the various use cases of apps to fetch-process-display data.

Where to Go From Here?

We hope you enjoyed this sample of *Kotlin Coroutines by Tutorials*!

If you enjoyed this sample, be sure to check out the full book, which will contain the following chapters:

1. **What Is Asynchronous Programming?**: Before getting you into the magic of coroutines, we'll help you understand what problem coroutines will solve for you. You'll learn what it means to be asynchronous and how to escape from an "indentation hell." This is a fundamental chapter in order to understand the basics of multi-threading and concurrent programming.
2. **Setting Up Your Build Environments**: This tutorial-style chaptre will get your hands dirty as soon as possible — this is where you will set your environment and get ready to write your first coroutine.
3. **Getting Started with Coroutines**: Learn the core concepts of coroutines and how you can launch them using a builder. You'll also learn about jobs and how to manage dependencies between them.
4. **Suspending Functions**: This is the main concept around coroutines. Here, you'll learn how to define a suspending function and how to manage its results.
5. **Async/Await**: Synchronization is a fundamental part of every concurrent framework, and coroutines aren't any different. In this chapter, you'll learn how to master the async and away functions in order to achieve an efficient synchronization between tasks.

6. **Building Sequences & Iterators with Yield**: Functional programming is one of the coolest concept you can use in Kotlin and, in this chapter, you'll see how you can use coroutines with sequences and iterators in order to manage theoretically infinite collections of data.

7. **Coroutine Contexts & Dispatchers**: Long and expensive tasks that run in the background and want to display the results on the main thread is a typical scenario in programming. In this chapter, you'll understand how to achieve this through Context and Dispatchers.

8. **Exception Handling & Cancellation**: Running a task is a relatively simple operation. Problems arise when errors occur or when you need to cancel the task because it becomes obsolete. In this chapter, you'll learn how to do handle these scenarios and how coroutines can manage dependencies in these cases.

9. **Coroutines as State Machines**: Every time you use a framework, it's important to understand how it works under the hood in order to fix unusual problems or to extend the way it works. In this chapter, you'll learn what is a state machine and how it's used by coroutines in order to do its magic.

10. **Channels**: Think about a box: Somebody can put objects into it and somebody else can remove them. Imagine, then, that the first actor, the producer, suspends its job if the box is full and that the second actor, the consumer, does the same if the box is empty. This is the logic behind channels, which you'll learn about in this chapter.

11. **Producers & Actors**: This has nothing to do with Hollywood! Here, you'll learn a different concurrent model that will allow you to manage the state of your application in a convenient and type-safe way.

12. **Broadcast Channels**: The channel you've covered in Chapter 10 is usually between a single sender and a single receiver. In this chapter, you'll learn what is happening exactly and what you can do if you have multiple receivers.

13. **Coroutine Operators**: Learn the most important operators that you can use in order to elaborate and combine streams, as you usually do with Rx.

14. **Coroutines & RxKotlin Comparison**: Coroutines are not the silver bullet of concurrent programming. Here, you'll learn what is the difference between coroutines and RxKotlin and how to understand which is the best for you.

15. **Coroutines on Android: Part 1**: Learn how you can manage concurrency and multi-threading in Android and why coroutines can help you simplify and optimize code in many ways.

16. **Coroutines on Android: Part 2**: Learn how to use different contexts in order to run long tasks in the background returning data to the main thread. You'll learn how to use async callbacks for long-running tasks, such as a database or network access into sequential tasks, while also keeping track of and handling app lifecycles.
17. **Coroutines on Android: Part 3**: Learn how to use Kotlin coroutines in an Android app with logging, exception handling, debugging and testing of code. Anko library will also be covered.

You can find the book on the raywenderlich.com store here: https://store.raywenderlich.com/products/kotlin-coroutines-by-tutorials

We hope you enjoy the book!

— The *Kotlin Coroutines by Tutorials* Team

Learn Coroutines in Kotlin!

Executing background tasks has always been a big challenge in every environment and, in particular, on mobile devices where resources are limited. Kotlin has simplified the way you can write code improving your productivity with a new programming paradigm, enhancing object-oriented and functional programming with with simple, powerful and new constructs. Coroutines are one of these!

Who This Book Is For

This book is for intermediate Kotlin or Android developers who already know the basics of UI development but want to learn coroutine API in order to simplify and optimise their code.

Topics Covered in Kotlin Coroutines by Tutorials:

- **Asynchronous programming:** Learn what asynchronous programming means and how to achieve it using not blocking calls.
- **Configuration:** Learn how to configure IntelliJ and Android Studio in order to use Coroutine APIs
- **Coroutine principles:** Learn what coroutines and launching builders are and how to manage Job dependencies.
- **Suspending functions:** This is the main concept around coroutines. Learn how to declare a suspending function and how to deal with results.
- **Sequences and Iterators:** Learn how to manage theoretically infinite collections of data in an efficient way using Sequences, Iterators and the yield function.
- **Thread communication techniques:** Learn how different tasks can communicate using Channels, Actors, and specific coroutine operators.
- **And much more,** including benchmarks, Broadcast Channels, and State machines!

One thing you can count on: After reading this book, you'll be prepared to take advantage of all the improvements coroutines have to offer!

About the iOS Tutorial Team

The Tutorial Team is a group of app developers and authors who write tutorials at the popular website **raywenderlich.com.** We take pride in making sure each tutorial we write holds to the highest standards of quality. We want our tutorials to be well written, easy to follow, and fun.

If you've enjoyed the tutorials we've written in the past, you're in for a treat. The tutorials we've written for this book are some of our best yet — and this book contains detailed technical knowledge you simply won't be able to find anywhere else.