Up to date for Kotlin 1.3

Kotlin Coroutines by Tutorials

SECOND EDITION

Mastering Coroutines in Kotlin and Android

By the raywenderlich Tutorial Team

Filip Babić & Nishant Srivastava

Table of Contents: Overview

Table of Contents: Extended

About This Book Sample

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

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

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

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

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

This sample includes:

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

The book is ready for purchase at:

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

Enjoy!

The *Kotlin Coroutines by Tutorials* Team

Kotlin Coroutines by Tutorials

By Filip Babić and Nishant Srivastava

Notice of Rights

Notice of Liability

Trademarks

Dedications

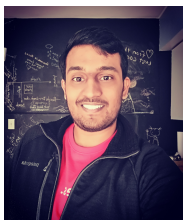
"To my friends and family. And mostly to my loved one. Thank you for being patient and understanding, when I couldn't grab a cup of coffee or tea and catch up. Huge thanks to everyone who's supported me throughout the entire process, with positive and motivational encouragement. This wouldn't have gone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

— Filip Babić

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

— Nishant Srivastava

About the Authors



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

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

About the Editors

Eric Crawford is a technical 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.

Kevin Moore is a technical editor for the book. He has been developing Android apps for over 9 years and at many companies. He's written several articles at www.raywenderlich.com and created the "Programming in Kotlin" video series. He enjoys creating apps for fun and teaching others how to write Android apps.In addition to programming, he loves playing Volleyball and running the sound system at church.

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.

About the Artist

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

What You Need

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

- **IntelliJ IDEA Community Edition 2019.1.x**: 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 use the IDE available at https://developer.android.com/studio/.

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

Book License

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

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

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

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

B Book Source Code & Forums

If you bought the digital edition

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

If you bought the print version

You can get the source code for the print edition of the book here:

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

Forums

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.

Digital book editions

We have a digital edition of this book available in both ePUB and PDF, which can be handy if you want a soft copy to take with you, or you want to quickly search for a specific term within the book.

Buying the digital edition version of the book also has a few extra benefits: free updates each time we update the book, access to older versions of the book, and you can download the digital editions from anywhere, at anytime.

Visit our *Kotlin Coroutines by Tutorials* store page here:

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

And if you purchased the print version of this book, you're eligible to upgrade to the digital editions at a significant discount! Simply email support@razeware.com with your receipt for the physical copy and we'll get you set up with the discounted digital edition version of the book.

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, applications do **complex work**, such as talking to external services or processing data from a database. Then, when the work is done, they show 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 user so that they don't feel like the application has frozen, that they didn't click a button properly — or perhaps that a feature doesn't work at all.

In this chapter, you'll learn how to provide useful information to 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 the 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 stopped 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. `showLoadingSpinner()` must then contain code such as this:

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

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

In `uploadImage()`, you assumed that the spinner animation was running even after the completion of `showLoadingSpinner()`, 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 `showLoadingSpinner()` has not completed. If `showLoadingSpinner()` has completed, then the `upload` has started. This means that the spinner is not animating anymore. This is happening because when you invoke `showLoadingSpinner()` 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, `showLoadingSpinner()` prevents the upload of an image because it keeps the **main thread** of execution busy until it returns. But when it returns (because `running` 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 the **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 the UI. But if the **main thread** cannot do this 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 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 their 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 the 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**. This 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 used to control their collaboration and synchronization, is called **concurrency**.

Given this, you can rethink how `uploadImage()` should work. `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 **non-blocking call** and can return immediately, allowing the image upload to start 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 with 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 for rotating the spinner image needs to communicate with the main thread in order to refresh the UI at each frame. The worker thread is responsible for the actual upload and needs to communicate with the UI thread which handles the animation when it completes in order to stop it, and to hide the spinner. All of 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 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, which is one of the main benefits of well written concurrency code.

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 a **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 to as **immutable**. Immutable objects are always thread safe.

As a practical 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 and make a mess. As one person started making a mocha latte and another wanted 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 to 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 it also 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. 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 restaurant. 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 consumes it and leaves the queue. You could also look at the customers as produced work, which the workers need to consume and serve, but the idea stays the same.

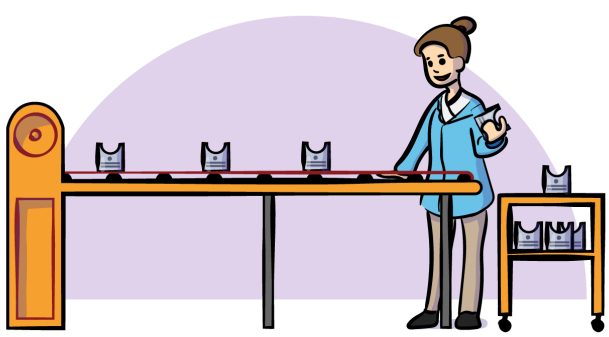
Pipelines

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 of the creation of objects that encapsulate code that somebody else can execute later, like when a specific task completes . This approach can also be 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 **non-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 implementation of the interface 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 supposed to know.

In modern programming languages like Kotlin, which support 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 very first snippet, not much has changed. You still show a loading spinner, call `upload()` 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 a parameter to `upload()`, which will be executed 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 is passed down into 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 types 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. Their 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 about. 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 own ambda 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 would be the 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 **stair-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. Reactive extensions, or Rx, are 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** that extend 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 as:

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

At first, this 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()` — and passes it to another function, creating a new stream. Then, the new stream sends events in the form of `resizedImage`, which gets passed to `uploadImage()`, using `flatMapCompletable()`, and operator chaining.

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

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

Additionally, `doOnSubscribe()` 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 with switching between threads, it takes a large 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, in addition to 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 problems like any other framework, or paradigm, some of which are showing 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 using 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 back to the 1960's, depicting a unique way of handling asynchronous programming. The concept revolves around the use of **suspension points**, **suspendable 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 is fetched before it is 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 of **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 a call *without* using callbacks, threads or streams. Think of it this way, once you start a coroutine, or call a suspendable function, 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 works

It's not really black magic — only a smart way of using low-level processing. `getUser()` is marked as a **suspendable 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 for further commands. Once you're ready to eat the taco and you request the result, the program can block until you get a ready-to-go snack, or suspend and wait for it within the coroutine.

Knowing this, the program can skip over 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 `getUser()` and if it hasn't already, suspends 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 suspendable functions are asynchronous and treats their execution sequentially, you can write understandable and clean code. This makes your code very extensible and easy to maintain.

Since writing asynchronous code is so simple with coroutines, you can easily combine multiple requests or transformations of 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 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 versions 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 a 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 support for the `yield` statement, like in Python, but also for `async` and `await` calls, like this:

```
MyResult result = await AsyncMethodThatReturnsAResult();

await AsyncMethodWithoutAResult();
```

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. By including them in the language itself, it allows you to make asynchronous calls without including a third-party framework. Many other programming languages utilize external libraries in order to support programming with coroutines. Kotlin also has coroutine support in its standard library. Additionally, the way Kotlin coroutines are built using global and extension functions with receivers, makes them very **extensible**. You can also create your own APIs 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 a **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 behind asynchronous programming and coroutines.

If you're excited about seeing some code and Kotlin's coroutines, in the next section of the book you'll learn about **suspendable 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, learn 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!

Where to Go From Here?

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

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

- **Chapter 1: What Is Asynchronous Programming?**: In this very first chapter, you'll learn what asynchronous programming means and why a modern developer should understand it. You'll see the basics of multithreading like queue and shared memory and you'll understand how to solve the "Indentation Hell Problem".
- **Chapter 2: Setting Up Your Build Environments**: Learning through example is one of the most efficient ways to gain more skills. To do this, you need to set up your build environment and learn how to load the starting projects with IntelliJ or Android Studio. This chapter describes all you need to start writing your code.
- **Chapter 3: Getting Started with Coroutines**: This is the chapter where you'll learn the main concepts about coroutines like builders, scope and context. You'll see for the first time the Job object and learn how to manage dependencies between coroutines. You'll understand and write code to manage one of the most important features of asynchronous tasks: cancellations.
- **Chapter 4: Suspending Functions**: To understand how to use coroutines you need to learn what a suspending function is and how to implement it. In this chapter, you'll learn all you need to create and use your suspending functions. You'll also learn how to change your existing functions to use them in a coroutine.

- **Chapter 5: Async/Await**: In multithreading and asynchronous development in Java, you often use Runnable, Callable and Future. With coroutines, you can use Deferred instead. These are objects that you can manage using the async/await functions. In this chapter, you'll write code to understand when and how to use this pattern most effectively.
- **Chapter 6: Coroutine Context**: This chapter is about one of the most important concepts about coroutines: Coroutine Context. You'll learn what it is and how this is related to the dependencies between different coroutine jobs. You'll also learn how to create your context.
- **Chapter 7: Context Switch & Dispatching**: In this chapter, you'll learn how to run different Jobs into the proper thread. You'll learn how to configure and use the proper thread to display information on the UI or to invoke different services on the network.
- **Chapter 8: Exception Handling**: Using functions with a callback is not difficult only because of the indentation hell problem but also for error and exception handling. In this very important chapter, you'll learn, with several examples, all the techniques you can use to handle exceptions.
- **Chapter 9: Manage Cancellation**: One of the most important topics to master when you deal with multithreading is a cancellation. Starting a thread is very easy compared to the techniques used to cancel it leaving the system in a consistent state. In this very important chapter, you'll learn, with several examples, all the techniques you can use to manage cancellations.
- **Chapter 10: Building Sequences & Iterators with Yield**: Sequences are one of the most interesting features of Kotlin because they allow generating values lazily. When you implement a sequence you use the yield function which is a suspending function. In this chapter, you'll learn how to create sequences and how the yield function can be used to optimize performance.
- **Chapter 11: Channels**: Although experimental, channels are a very important API you can use with coroutines. In this chapter, you'll create examples to understand what a channel is and how to act as a producer or consumer for it synchronously and asynchronously. You'll understand how to use multiple channels in the case of multiple senders and receivers. You'll finally compare channels with Java's BlockingQueue
- **Chapter 12: Broadcast Channels**: In this chapter, you'll write many examples to experiment with using channels with multiple receivers and emitted items need to be shared by all of them.

- **Chapter 13: Producer & Actors**: In this chapter, you'll learn how coroutines can help implement a producer/consumer pattern using different types of producers and consumers. Another approach to running tasks in the background is to use the actors model. In the second part of this chapter, you'll learn what an Actor is and how you can use it with coroutines.
- **Chapter 14: Beginning with Coroutine Flow**: In this chapter, you'll learn what Coroutine Flow is and how to use them in your project.
- **Chapter 15: Testing Coroutines**: Testing is a fundamental part of the development process and coroutines are not different. In this chapter, you'll learn how to test coroutines using the main testing frameworks.
- **Chapter 16: Android Concurrency Before Coroutines**: The Android platform allows you to run background tasks in many different ways. In this chapter, you'll see and implement examples for all of them. You'll learn what Looper and Handler are and when to use an AsyncTask. You'll finally see how coroutines can make the code more readable and efficient.
- **Chapter 17: Coroutine on Android - Part 1**: The chapter covers using Kotlin Coroutines in an Android app, covering working with various context i.e. UI and background to simplify and manage code sequentially. It will cover converting async callbacks for long-running tasks, such as a database or network access into sequential tasks while also keeping track and handling of the app lifecycle.
- **Chapter 18: Coroutine on Android - Part 2**: The chapter covers fortifying the use of Kotlin Coroutines in an Android app i.e. Enabling logging, exception handling, debugging and testing of code that uses Kotlin Coroutines. Towards the end, Anko library will also be covered.

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

We hope you enjoy the book!

— The *Kotlin Coroutines by Tutorials* Team

Learn coroutines!

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

Who This Book Is For

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

Topics Covered in Kotlin Coroutines by Tutorials:

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

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

About the Tutorial Team

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

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