Learning Concurrency in Kotlin

Build highly efficient and robust applications

Packt>

www.packt.com

By Miguel Angel Castiblanco Torres

Learning Concurrency in Kotlin

Build highly efficient and robust applications

Miguel Angel Castiblanco Torres

BIRMINGHAM - MUMBAI

Learning Concurrency in Kotlin

Commissioning Editor: Aaron Lazar
Acquisition Editor: Sandip Mishra
Content Development Editor: Anugraha Arunagiri
Technical Editor: Jijo Maliyekal
Copy Editor: Safis Editing
Project Coordinator: Ulhas Kambali
Proofreader: Safis Editing
Indexer: Tejal Daruwale Soni
Graphics: Tania Dutta
Production Coordinator: Arvindkumar Gupta

First published: July 2018

Production reference: 1270718

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78862-716-0

www.packtpub.com

mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

PacktPub.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Miguel Angel Castiblanco Torres is a software engineer living in the United States. He works as a full-stack technical leader and software designer at Globant, where he has led many successful projects for a Forbes' Top Ten World's Most Valuable Brand and Top Five Regarded Company.

Passionate about what's next, Miguel was an early adopter of Kotlin, writing about Kotlin's concurrency primitives from the first beta release of coroutines. He always keeps an eye on the new and upcoming features of Kotlin.

About the reviewer

Peter Sommerhoff helps you to always keep learning, master new skills, and, ultimately, achieve your goals. Does that sound like something you'd like to do? If yes, ensure that you enroll for one of his courses and learn about programming languages, software design, productivity, or anything else you're interested in! He will ensure that you have a great learning experience. You can always ask him personally if you have anything at all that he can help you with. So, check out his courses, and he'll see you there!

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface

The era when the speed of a single core in a processor would duplicate every few years is long over. Transistors have reached unbelievably small sizes, yet heating, power consumption, and other technical and practical challenges have made it difficult to keep increasing their speed in a user-friendly, affordable way. Even more so if you consider not only desktop computers but also mobile devices such as smartphones and tablets, which are universal nowadays. So, in recent years, the focus has been put on equipping these computing devices with more than one core – or more than one processor – running with a balanced configuration of speed, heating, and power consumption.

The challenge becomes using those cores efficiently and effectively. An increase in the speed of a core will directly improve the execution time of the software running on it – for CPU-bound operations – but that is not the case when another core or processor is added. For an application to be able to take advantage of multiple cores, it needs to be written with that objective in mind. It needs to be *concurrent*.

However, concurrency not only benefits applications that will run in multiple cores. A concurrent application will usually work faster than a single-threaded application for IO-bound operations, even if running in a single core, because it allows the application to use the processor while other tasks are waiting for IO devices. For example, while waiting for a network operation.

So it's in your hands, as the developer, the person reading this book, to write the software that will work efficiently both in a single-core device and in a multi-core one. It's in my hands, the writer, to provide you with enough information, expressed in an understandable way, to achieve this goal. With this in mind, I have written this book, expecting nothing but basic Kotlin knowledge from you. I have tried to distill everything, from the very concepts of concurrency to the very specifics of how coroutines are compiled, for you. I have done it so that we both can do our part in bringing our users – be it humans or other systems – a fast yet reliable experience.

This book covers various topics in great detail, using tools such as diagrams, examples, and more often than not, real-life use cases. For that reason, most of the chapters follow the creation and modification of an RSS reader for Android, completely written in Kotlin and using only the tools and primitives provided by the language and the coroutines library. I encourage you to follow along the creation of the application and to type the code instead of copying it – especially if you have only the printed version of the book – so that you get to interiorize how to do it. I encourage you even more to go off-script and try some things for yourself; if you start wondering whether something can be accomplished in a certain way, it means that your brain is beginning to get a grasp of how to apply the concepts. And there's nothing better than trying something to know if it works.

One thing that you should be aware of is that while coroutines are being developed for all the platforms that Kotlin supports – JVM, JavaScript, and Kotlin/Native – at the time of writing this book, the most complete implementation is that of JVM. For that reason, I have written this book centered around what is available and how things work for the JVM. However, know that many of the concepts and primitives work in a similar fashion for JavaScript, and I believe they will work the same way in Kotlin/Native once implemented.

There are still many topics that I wasn't able to write about, but I do believe that the contents of this book are enough to get you to a point from where you can confidently move forward on you own, and I also believe that you will be able to write many of your concurrent applications based on what will be covered in these pages.

> *This puts the future squarely in the hands of those who know computers not for what they are, but the potential of what they can be*
>
> *- Gordon*

> *Computers aren't the thing, they are the thing that gets us to the thing*
>
> *- Joe*

Who this book is for

This book is aimed at any Kotlin developer who is interested in learning about concurrency in general, or applied to Kotlin in specific. I have written this book so that only basic knowledge of Kotlin is expected, and everything else is explained as needed.

What this book covers

`Chapter 1`, *Hello, Concurrent World!*, is a friendly introduction to concurrency. Starting with concepts such as processes, threads, and coroutines, and then diving into the differences between concurrency and parallelism, you will learn about the common challenges of concurrent code, Kotlin's approach to concurrency, and some of the terminology that will be needed throughout the book.

`Chapter 2`, *Coroutines in Action*, takes you through a hands-on first experience with coroutines. We will create an empty Android project and work our way to doing networking using coroutines. We will also discuss different approaches to asynchronous operations with their advantages and my recommendations.

`Chapter 3`, *Lifecycle and Error Handling*, explains the lifecycle of coroutines and how to handle errors when using them. Then, we will put the theory in practice by adding error handling to the RSS reader created in the previous chapter.

`Chapter 4`, *Suspending Functions and the Coroutine Context*, covers the essential coroutine context, the one piece of configuration of your coroutine that allows you to define and modify its behavior. We will cover the usage of contexts to define the thread of a coroutine and for error handling.

`Chapter 5`, *Iterators, Sequences, and Producers*, dives into some primitives that allow data sourcing and data processing to be made suspending. We will talk, in detail, about how and when to use them, and we'll put this knowledge to test by implementing content loading on demand.

`Chapter 6`, *Channels, Share Memory by Communicating*, explains the really important concept of channels, starting with their use cases, covering the different types and their differences, and making it to a real-life application. We will use channels to perform concurrent searches in a group of RSS feeds.

`Chapter 7`, *Thread Confinement, Actors, and Mutexes*, lists three different tools that are provided by the coroutines framework to avoid errors such as atomicity violations and race conditions. In this chapter, you will learn how to use thread confinement, actors, and mutual exclusions, which are fundamental when writing safe concurrent code.

`Chapter 8`, *Testing and Debugging Concurrent Code*, focuses on how to make the most out of your tests. We talk about functional tests, the three premises for testing concurrent code, and the configuration of your project to make the most out of your logs. We will also cover how to debug your concurrent code so that you avoid noise from other threads or coroutines.

`Chapter 9`, *The Internals of Concurrency in Kotlin*, analyzes how suspending functions, thread enforcing, and exception handling work during execution. It gives you a low-level understanding by both describing what the compiler does and by exploring the current implementation of many of the classes, both in the stdlib and in the coroutines library.

To get the most out of this book

1. You should have basic knowledge of Kotlin. While all the concepts, primitives, and code related to concurrency are explained in great detail, some basic knowledge is assumed. For example, you should know how to create variables, classes, what a lambda is, and such like.
2. No knowledge of concurrency is assumed. This book aims to give you an introduction to concurrency, plus how to approach it from a pure Kotlin perspective.
3. Read the code samples with attention to detail, and ensure that you follow along the creation of the Android application. It will be easier for you to learn if you really dedicate the time to putting things in practice.
4. An installation of IntelliJ IDEA Community 18.01 or a more recent version is expected. We will cover the installation of Android Studio 3.1 during `Chapter 2`, *Coroutines in Action*, so that you can follow the development of the RSS reader.

Download the example code files

You can download the example code files for this book from your account at `www.packtpub.com`. If you purchased this book elsewhere, you can visit `www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packtpub.com`.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Learning-Concurrency-in-Kotlin`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates the name of variables, functions, classes, packages, files, and URLs. Here is an example: "The `hcf()` function will freeze the system."

A block of code is set as follows:

```
fun main(args: Array<String>) = runBlocking {
    val time = measureTimeMillis {
        val name = async { getName() }
        val lastName = async { getLastName() }

        println("Hello, ${name.await()} ${lastName.await()}")
    }
    println("Execution took $time ms")
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
fun main(args: Array<String>) = runBlocking {
    val netDispatcher = newSingleThreadContext(name = "ServiceCall")

    val task = launch(netDispatcher) {
        printCurrentThread()
    }

    task.join()
}
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Please make sure that the **Include Kotlin Support** option is enabled."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com` and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packtpub.com`.

1
Hello, Concurrent World!

This chapter is an introduction to concurrency and how Kotlin approaches concurrency challenges. In this chapter, we will also take a first look at asynchronous code written using coroutines. It's important to mention that while the keywords, primitives, and functions related to concurrency in pure Kotlin will be explained through the book, some knowledge of Kotlin is required in order to be able to fully comprehend the code examples.

The following topics will be covered in this chapter:

- Processes, threads, coroutines, and their relationships
- Introduction to concurrency
- Concurrency versus parallelism
- CPU-bound and I/O-bound algorithms
- Why concurrency is often feared
- Concurrency in Kotlin
- Concepts and terminology

Processes, threads, and coroutines

When you start an application, the operating system will create a process, attach a thread to it, and then start the execution of that thread – commonly known as the main thread. In this section, we will detail the relationship between processes, threads, and coroutines. This is necessary in order to be able to understand and implement concurrency.

Processes

A process is an instance of an application that is being executed. Each time an application is started, a process is started for it. A process has a state; things such as handles to open resources, a process ID, data, network connections, and so on, are part of the state of a process and can be accessed by the threads inside that process.

An application can be composed of many processes, a common practice for example for internet browsers. But implementing a multi-process application brings challenges that are out of the scope of this book. For this book, we will cover the implementation of applications that run in more than one thread, but still in a single process.

Threads

A thread of execution encompasses a set of instructions for a processor to execute. So a process will contain at least one thread, which is created to execute the point of entry of the application; usually this entry point is the `main()` function of the application. This thread is called the main thread, and the life cycle of the process will be tied to it; if this thread ends, the process will end as well, regardless of any other threads in the process. For example:

```
fun main(args: Array<String>) {
    doWork()
}
```

When this basic application is executed, a main thread is created containing the set of instructions of the `main()` function. `doWork()` will be executed in the main thread, so whenever `doWork()` ends, the execution of the application will end with it.

Each thread can access and modify the resources contained in the process it's attached to, but it also has its own local storage, called thread-local storage.

Only one of the instructions in a thread can be executed at a given time. So if a thread is blocked, the execution of any other instruction in that same thread will not be possible until the blocking ends. Nevertheless, many threads can be created for the same process, and they can communicate with each other. So it is expected that an application will never block a thread that can affect negatively the experience of the user; instead, the blocking operations should be assigned to threads that are dedicated to them.

In Graphic User Interface (GUI) applications, there is a thread called a UI thread; its function is to update the User Interface and listen to user interactions with the application. Blocking this thread, obstructs the application from updating its UI and from receiving interactions from the user. Because of this, GUI applications are expected to never block the UI thread, in order to keep the application responsive at all times.

Android 3.0 and above, for example, will crash an application if a networking operation is made in the UI thread, in order to discourage developers from doing it, given that networking operations are thread-blocking.

Throughout the book, we will refer to the main thread of a GUI application both as a UI thread and as a main thread (because in Android, by default, the main thread is also the UI thread), while for command-line applications we will refer to it only as a main thread. Any thread different from those two will be called a background thread, unless a distinction between background threads is required, in which case each background thread will receive a unique identifier for clarity.

Given the way that the Kotlin has implemented concurrency, you will find that it's not necessary for you to manually start or stop a thread. The interactions that you will have with threads will commonly be limited to tell Kotlin to create or use a specific thread or pool of threads to run a coroutine – usually with one or two lines of code. The rest of the handling of threads will be done by the framework.

In `Chapter 3`, *Lifecycle and Error Handling*, we will talk about how to correctly run coroutines in a background thread in order to avoid blocking the main thread.

Coroutines

Kotlin's documentation often refers to coroutines as lightweight threads. This is mostly because, like threads, coroutines define the execution of a set of instructions for a processor to execute. Also, coroutines have a similar life cycle to that of threads.

A coroutine is executed inside a thread. One thread can have many coroutines inside it, but as already mentioned, only one instruction can be executed in a thread at a given time. This means that if you have ten coroutines in the same thread, only one of them will be running at a given point in time.

The biggest difference between threads and coroutines, though, is that coroutines are fast and cheap to create. Spawning thousands of coroutines can be easily done, it is faster and requires fewer resources than spawning thousands of threads.

Take this code as an example. Don't worry about the parts of the code you don't understand yet:

```
suspend fun createCoroutines(amount: Int) {
    val jobs = ArrayList<Job>()
    for (i in 1..amount) {
        jobs += launch {
            delay(1000)
        }
    }
    jobs.forEach {
        it.join()
    }
}
```

This function creates as many coroutines as specified in the parameter `amount`, delays each one for a second, and waits for all of them to end before returning. This function can be called, for example, with 10,000 as the amount of coroutines:

```
fun main(args: Array<String>) = runBlocking {
    val time = measureTimeMillis {
        createCoroutines(10_000)
    }

    println("Took $time ms")
}
```

`measureTimeMillis()` is an inline function that takes a block of code and returns how long its execution took in milliseconds. `measureTimeMillis()` has a sibling function, `measureNanoTime()`, which returns the time in nanoseconds. Both functions are quite practical when you want a rough estimate of the execution time of a piece of code.

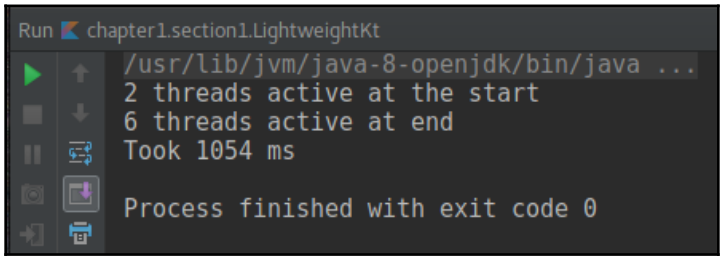
In a test environment, running it with an `amount` of 10,000 took around 1,160 ms, while running it with 100,000 took 1,649 ms. The increase in execution time is so small because Kotlin will use a pool of threads with a fixed size, and distribute the coroutines among those threads – so adding thousands of coroutines will have little impact. And while a coroutine is suspended – in this case because of the call to `delay()` – the thread it was running in will be used to execute another coroutine, one that is ready to be started or resumed.

How many threads are active can be determined by calling the `activeCount()` method of the `Thread` class. For example, let's update the `main()` function to do so:

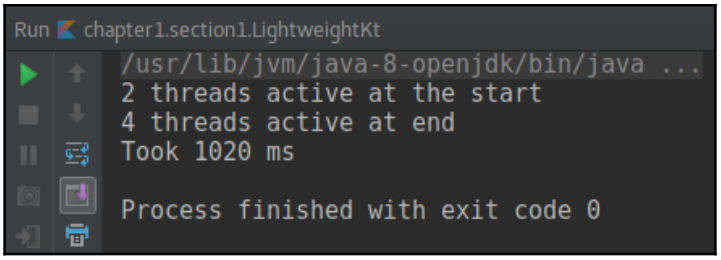
```
fun main(args: Array<String>) = runBlocking {
    println("${Thread.activeCount()} threads active at the start")
```

```
        val time = measureTimeMillis {
            createCoroutines(10_000)
        }
        println("${Thread.activeCount()} threads active at the end")
        println("Took $time ms")
    }
```

In the same test environment as before, it was found that in order to create 10,000 coroutines, only four threads needed to be created:



But once the value of the amount being sent to `createCoroutines()` is lowered to one, for example, only two threads are created:



Notice how at the start the application, already had two threads. This is because of a thread called Monitor Control+Break, which is created when you run an application in IntelliJ IDEA. This thread is in charge of processing the hotkey *Control+Break*, which dumps the information of all the threads running. If you run this code from the command line, or in IntelliJ using debug mode, it will display just one thread at the start and five at the end.

It's important to understand that even though a coroutine is executed inside a thread, it's not bound to it. As a matter of fact, it's possible to execute part of a coroutine in a thread, suspend the execution, and later continue in a different thread. In our previous example this is happening already, because Kotlin will move coroutines to threads that are available to execute them. For example, by passing `3` as the `amount` to `createCoroutines()`, and updating the content of the `launch()` block so that it prints the current thread, we can see this in action:

```
suspend fun createCoroutines(amount: Int) {
    val jobs = ArrayList<Job>()
    for (i in 1..amount) {
        jobs += launch {
            println("Started $i in ${Thread.currentThread().name}")
            delay(1000)
            println("Finished $i in ${Thread.currentThread().name}")
        }
    }
    jobs.forEach {
         it.join()
    }
}
```

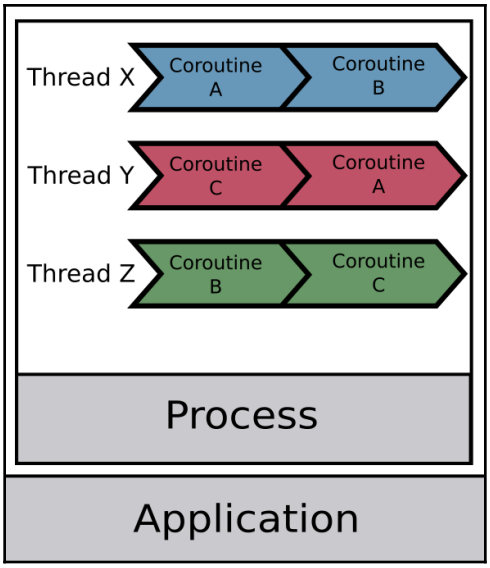
You will find that in many cases they are being resumed in a different thread:

A thread can only execute one coroutine at a time, so the framework is in charge of moving coroutines between threads as necessary. As will be explained in detail later, Kotlin is flexible enough to allow the developer to specify which thread to execute a coroutine on, and whether or not to confine the coroutine to that thread.

`Chapter 4`, *Suspending Functions and the Coroutine Context*, explains how to resume a coroutine in a thread different than the one in which it was started, while `Chapter 7`, *Thread Confinement, Actors, and Mutexes*, covers thread confinement in detail.

Putting things together

So far, we have learned that an application is composed of one or more processes and that each process has one or more threads. We have also learned that blocking a thread means halting the execution of the code in that thread, and for that reason, a thread that interacts with a user is expected to never be blocked. We also know that a coroutine is basically a lightweight thread that resides in a thread but is not tied to one. The following diagram encapsulates the content of this section so far. Notice how each coroutine is started in one thread but at some point is resumed in a different one:



In a nutshell, concurrency occurs when an application runs in more than one thread at the same time. So in order for concurrency to happen, two or more threads need to be created, and both communication and synchronization of those threads will most likely be needed for correct functioning of the application.

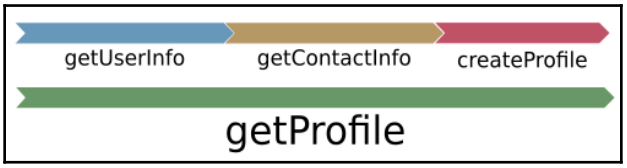
Later in this chapter, the differences between concurrency and parallelism will be discussed.

Introduction to concurrency

Correct concurrent code is one that allows for a certain variability in the order of its execution while still being deterministic on the result. For this to be possible, different parts of the code need to have some level of independence, and some degree of orchestration may also be required. The best way to understand concurrency is by comparing sequential code with concurrent code. Let's start by looking at some non-concurrent code:

```
fun getProfile(id: Int) : Profile {
    val basicUserInfo = getUserInfo(id)
    val contactInfo = getContactInfo(id)

    return createProfile(basicUserInfo, contactInfo)
}
```

If I ask you what is going to be obtained first, the user information or the contact information – assuming no exceptions – you will probably agree with me that 100% of the time the user information will be retrieved first. And you will be correct. That is, first and foremost, because the contact information is not being requested until the contact information has already been retrieved:



Timeline of getProfile

And that's the beauty of sequential code: you can easily see the exact order of execution, and you will never have surprises on that front. But sequential code has two big issues:

- It may perform poorly compared to concurrent code
- It may not take advantage of the hardware that it's being executed on

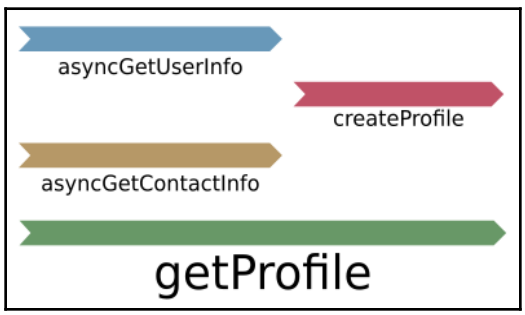
Let's say, for example, that both **getUserInfo** and **getContactInfo** call a web service, and each service will constantly take around one second to return a response. That means that **getProfile** will take not less than two seconds to finish, always. And since it seems like **getContactInfo** doesn't depend on **getUserInfo**, the calls could be done concurrently, and by doing so it would be possible can halve the execution time of **getProfile**.

Let's imagine a concurrent implementation of **getProfile**:

```
suspend fun getProfile(id: Int) {
    val basicUserInfo = asyncGetUserInfo(id)
    val contactInfo = asyncGetContactInfo(id)

    createProfile(basicUserInfo.await(), contactInfo.await())
}
```

In this updated version of the code, `getProfile()` is a suspending function – notice the `suspend` modifier in its definition – and the implementation of `asyncGetUserInfo()` and `asyncGetContactInfo()` are asynchronous – which will not be shown in the example code to keep things simple.

Because `asyncGetUserInfo()` and `asyncGetContactInfo()` are written to run in different threads, they are said to be concurrent. For now, let's think of it as if they are being executed at the same time – we will see later that it's not necessarily the case, but will do for now. This means that the execution of `asyncGetContactInfo()` will not depend on the completion of `asyncGetUserInfo()`, so the requests to the web services could be done around at the same time. And since we know that each service takes around one second to return a response, `createProfile()` will be called around one second after `getProfile()` is started, sooner than it could ever be in the sequential version of the code, where it will always take at least two seconds to be called. Let's take a look at how this may look:



Concurrent timeline for **getProfile**

But in this updated version of the code, we don't really know if the user information will be obtained before the contact information. Remember, we said that each of the web services takes *around one second*, and we also said that both requests will be started at around the same time.

This means that if **asyncGetContactInfo** is faster than **asyncGetUserInfo**, the contact information will be obtained first; but the user information could be obtained first if **asyncGetUserInfo** returns first; and since we are at it, it could also happen that both of them return the information at the same time. This means that our concurrent implementation of **getProfile**, while possibly performing twice as fast as the sequential one, has some variability in its execution.

That's the reason there are two `await()` calls when calling `createProfile()`. What this is doing is suspending the execution of `getProfile()` until both `asyncGetUserInfo()` and `asyncGetContactInfo()` have completed. Only when both of them have completed `createProfile()` will be executed. This guarantees that regardless of which of the concurrent call ends first, the result of `getProfile()` will be deterministic.

And that's where the tricky part of concurrency is. You need to guarantee that no matter the order in which the semi-independent parts of the code are completed, the result needs to be deterministic. For this example, what we did was suspend part of the code until all the moving parts completed, but as we will see later in the book, we can also orchestrate our concurrent code by having it communicate between coroutines.

Concurrency is not parallelism

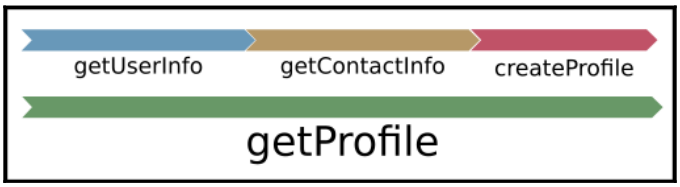
There's common confusion as to what the difference between concurrency and parallelism is. After all, both of them sound quite similar: two pieces of code running at the same time. In this section, we will define a clear line to divide both of them.

Let's start by going back to our non-concurrent example from the first section:

```
fun getProfile(id: Int) : Profile {
    val basicUserInfo = getUserInfo(id)
    val contactInfo = getContactInfo(id)

    return createProfile(basicUserInfo, contactInfo)
}
```

If we go back to the timeline for this implementation of `getProfile()`, we will see that the timelines of `getUserInfo()` and `getContactInfo()` don't overlap.

The execution of `getContactInfo()` will happen after `getUserInfo()` has finished, always:

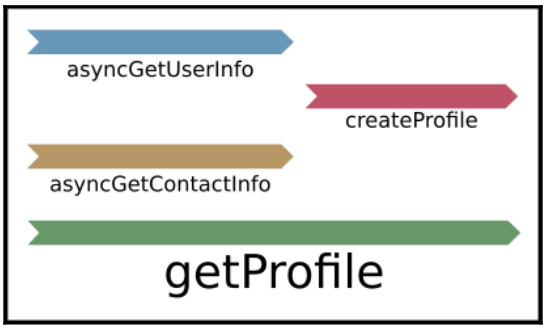


Timeline for the sequential implementation of getProfile

Let's now look again at the concurrent implementation:

```
suspend fun getProfile(id: Int) {
    val basicUserInfo = asyncGetUserInfo(id)
    val contactInfo = asyncGetContactInfo(id)

    createProfile(basicUserInfo.await(), contactInfo.await())
}
```

A timeline for the concurrent execution of this code would be something like the following diagram. Notice how the execution of `asyncGetUserInfo()` and `asyncGetContactInfo()` overlaps, whereas `createProfile()` doesn't overlap with either of them:

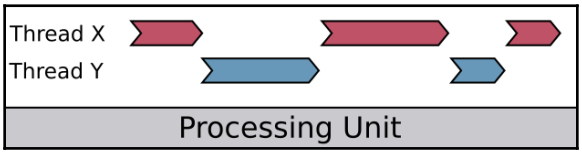


Timeline for a concurrent implementation of getProfile

A timeline for the parallel execution would look exactly the same as the one above. The reason why both concurrent and parallel timelines look the same is because this timeline is not granular enough to reflect what is happening at a lower level.

The difference is that concurrency happens when the timeline of different sets of instructions in the same process overlaps, regardless of whether they are being executed at the exact same point in time or not. The best way to think of this is by picturing the code of `getProfile()` being executed in a machine with only one core. Because a single core would not be able to execute both threads at the same time, it would interleave between `asyncGetUserInfo()` and `asyncGetContactInfo()`, making their timelines overlap – but they would not be executing simultaneously.

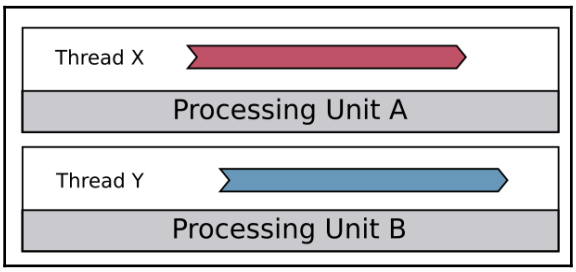
The following diagram is a representation of concurrency when it happens in a single core – it's concurrent but not parallel. A single processing unit will interleave between threads *X* and *Y*, and while both of their overall timelines overlap, at a given point in time only one of them is being executed:



Lower-level representation of concurrency

Parallel execution, on the other hand, can only happen if both of them are being executed at the exact same point in time. For example, if we picture `getProfile()` being executed in a computer with two cores, one core executing the instructions of `asyncGetUserInfo()` while a second core is executing those of `asyncGetContactInfo()`.

The following diagram is a representation of concurrent code being executed in parallel, using two processing units, each of them executing a thread independently. In this case, not only the timelines of thread *X* and *Y* are overlapping, but they are indeed being executed at the exact same point in time:



Lower-level representation of parallelism

Here is a summarized explanation:

- Concurrency happens when there is overlapping in the timeline of two or more algorithms. For this overlap to happen, it's necessary to have two or more active threads of execution. If those threads are executed in a single core, they are not executed in parallel, but they are executed concurrently nonetheless, because that single core will interleave between the instructions of the different threads, effectively overlapping their execution.
- Parallelism happens when two algorithms are being executed at the exact same point in time. For this to be possible, it's required to have two or more cores and two or more threads, so that each core can execute the instructions of a thread simultaneously. Notice that parallelism implies concurrency, but concurrency can exist without parallelism.

As we will see in the next section, this difference is not just a technicality, and understanding it, along with understanding your code, will help you to write code that performs well.

It's worth mentioning that parallelism's requirement for more than one core doesn't need to be local. For example, you could have an application run distributed work in many different computers in a network. Such implementations are called distributed computing, and are a form of parallelism.

CPU-bound and I/O-bound

Bottlenecks are the most important thing to understand when it comes to optimizing the performance of applications, because they will indicate the points in which any type of throttling occurs. In this section we will talk about how concurrency and parallelism can affect the performance of an algorithm based on whether it's bound to the CPU or to I/O operations.

You aren't always going to need or even benefit from writing concurrent code. Understanding the bottlenecks of your code, how threads and coroutines work, and the differences between concurrency and parallelism is required to be able to make a correct assessment of when and how to implement concurrent software.

CPU-bound

Many algorithms are built around operations that need only the CPU to be completed. The performance of such algorithms will be delimited by the CPU in which they are running, and solely upgrading the CPU will usually improve their performance.

Let's think, for example, of a simple algorithm that takes a word and returns whether it's a palindrome or not:

```
fun isPalindrome(word: String) : Boolean {
    val lcWord = word.toLowerCase()
    return lcWord == lcWord.reversed()
}
```

Now, let's consider that this function is called from another function, `filterPalindromes()`, which takes a list of words and returns the ones that are palindromes:

```
fun filterPalindromes(words: List<String>) : List<String> {
    return words.filter { isPalindrome(it) }
}
```

Finally, `filterPalindromes()` is called from the main method of the application where a list of words has been already defined:

```
val words = listOf("level", "pope", "needle", "Anna", "Pete", "noon",
"stats")

fun main(args: Array<String>) {
    filterPalindromes(words).forEach {
        println(it)
    }
}
```

In this example, all the parts of the execution depend on the CPU. If the code is updated to send hundreds of thousands of words, `filterPalindromes()` will take longer. Also, if the code is executed in a faster CPU, the performance will be improved without code changes.

I/O-bound

I/O-bound, on the other hand, are algorithms that rely on input/output devices, so their execution times depend on the speed of those devices, for example, an algorithm that reads a file and passes each word in the document to `filterPalindromes()` in order to print the palindromes in the document. Changing a few lines of the previous example will do:

```
fun main(args: Array<String>) {
    val words = readWordsFromJson("resources/words.json")
    filterPalindromes(words).forEach {
        println(it)
    }
}
```

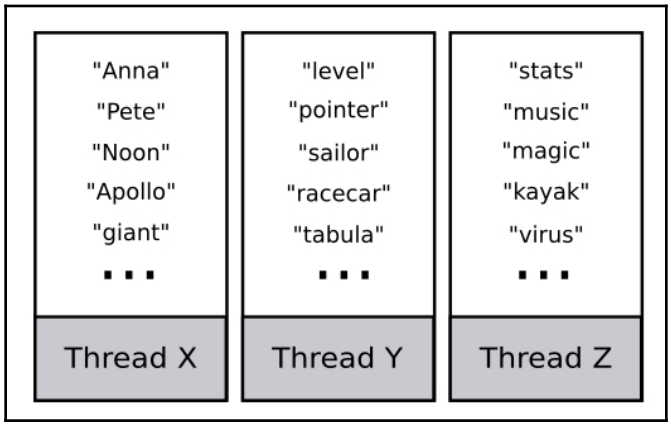
The `readWordsFromJson()` function will read the file from the filesystem. This is an I/O operation that will depend on the speed at which the file can be read. If the file is stored in a hard drive, the performance of the application will be worse than if the file is stored in an SSD, for example.

Many other operations, such as networking or receiving input from the peripherals of the computer, are also I/O operations. Algorithms that are I/O-bound will have performance bottleneck based on the I/O operations, and this means that optimizations are dependent on *external* systems or devices.

Many I/O-bound, high-performance applications, such as databases, may end up being as fast as the access to the storage of the machine they are running on. Networking-based applications, like many phone apps, will perform based on the speed of their connectivity to the internet.

Concurrency versus parallelism in CPU-bound algorithms

CPU-bound algorithms will benefit from parallelism but probably not from single-core concurrency. For example, the previous algorithm that takes a list of words and filters the palindromes could be modified so that a thread is created per each 1,000 words received. So, if `isPalindrome()` receives 3,000 words, the execution could be represented like this:



isPalindrome creating a thread per each 1,000 words

Single-core execution

If this is executed in a single core, that core will then interleave between the three threads, each time filtering some of the words before switching to the next one. This interleaving process is called context switching.

Context switching adds overhead to the overall process, because it requires saving the state of the current thread and then loading the state of the next one. This overhead makes it likely that this multi-threaded implementation of `isPalindrome()` will take longer in a single-core machine when compared to the sequential implementation seen before. This happens because the sequential implementation will have one core do all the work but will avoid the context switch.

Parallel execution

If parallel execution is assumed, where each thread is executed in one dedicated core, then the execution of `isPalindrome()` could be around one third of that of the sequential implementation. Each core will filter its 1,000 words without interruption, reducing the total amount of time needed to complete the operation.

It's important to consider creating a reasonable amount of threads for CPU-bound algorithms, making this decision based on the amount of cores of the current device.. This can be leveraged by using Kotlin's `CommonPool`, which is a pool of threads created to run CPU-bound algorithms.

`CommonPool`'s size is the amount of cores on the machine minus one. So, for example, it will be of size three in a machine with four cores.

Concurrency versus parallelism in I/O-bound algorithms

As seen before, I/O-bound algorithms are constantly waiting on something else. This constant waiting allows single-core devices to use the processor to do other useful tasks while waiting. So concurrent algorithms that are I/O-bound will perform similarly regardless of the execution happening in parallel or in a single core.

It is expected that I/O-bound algorithms will always perform better in concurrent implementations than if they are sequential. So it's recommended for I/O operations to be always executed concurrently. As mentioned before, in GUI applications it's particularly important to not block the UI thread.

Why concurrency is often feared

Writing correct concurrent code is traditionally considered difficult. This is not only because of it being difficult, but also because many programming languages make it more difficult than it should be. Some languages make it too cumbersome, while others make it inflexible, reducing its usability. With that in mind, the Kotlin team tried to make concurrency as simple as possible while still making it flexible enough so that it can be adjusted to many different use cases. Later in the book, we will cover many of those use cases and will use many of the primitives that the Kotlin team has created, but for now let's take a look at common challenges presented when programming concurrent code.

As you can probably guess by now, most of the time it comes down to being able to synchronize and communicate our concurrent code so that changes in the flow of execution don't affect the operation of our application.

Race conditions

Race conditions, perhaps the most common error when writing concurrent code, happen when concurrent code is written but is expected to behave like sequential code. In more practical terms, it means that there is an assumption that concurrent code will always execute in a particular order.

For example, let's say that you are writing a function that has to concurrently fetch something from a database and call a web service. Then it has to do some computation once both operations are completed. It would be a common mistake to assume that going to the database will be faster, so many people may try to access the result of the database operation as soon as the web service operation is done, thinking that by that time the information from the database will always be ready. Whenever the database operation takes longer than the webservice call, the application will either crash or enter an inconsistent state.

A race condition happens, then, when a concurrent piece of software requires semi-independent operations to complete in a certain order to be able to work correctly. And this is not how concurrent code should be implemented.

Let's see a simple example of this:

```
data class UserInfo(val name: String, val lastName: String, val id: Int)

lateinit var user: UserInfo

fun main(args: Array<String>) = runBlocking {
```

```
    asyncGetUserInfo(1)
    // Do some other operations
    delay(1000)

    println("User ${user.id} is ${user.name}")
}

fun asyncGetUserInfo(id: Int) = async {
    user = UserInfo(id = id, name = "Susan", lastName = "Calvin")
}
```

The `main()` function is using a background coroutine to get the information of a user, and after a delay of a second (simulating other tasks), it prints the name of the user. This code will work because of the one second delay. If we either remove that delay or put a higher delay inside `asyncGetUserInfo()`, the application will crash. Let's replace `asyncGetUserInfo()` with the following implementation:

```
fun asyncGetUserInfo(id: Int) = async {
    delay(1100)
    user = UserInfo(id = id, name = "Susan", lastName = "Calvin")
}
```

Executing this will cause `main()` to crash while trying to print the information in `user`, which hasn't been initialized. To fix this race condition, it is important to explicitly wait until the information has been obtained before trying to access it.

Atomicity violation

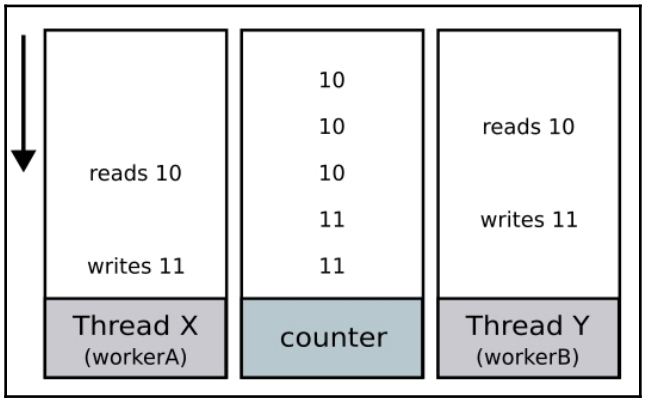
Atomic operations are those that have non-interfered access to the data they use. In single-thread applications, all the operations will be atomic, because the execution of all the code will be sequential – and there can't be interference if only one thread is running.

Atomicity is wanted when the state of an object can be modified concurrently, and it needs to be guaranteed that the modification of that state will not overlap. If the modification can overlap, that means that data loss may occur due to, for example, one coroutine overriding a modification that another coroutine was doing. Let's see it in action:

```
var counter = 0
fun main(args: Array<String>) = runBlocking {
    val workerA = asyncIncrement(2000)
    val workerB = asyncIncrement(100)
    workerA.await()
    workerB.await()
    print("counter [$counter]")
```

```
}

fun asyncIncrement(by: Int) = async {
    for (i in 0 until by) {
        counter++
    }
}
```

This is a simple example of atomicity violation. The previous code executes the `asyncIncrement()` coroutine twice, concurrently. One of those calls will increment `counter` 2,000 times, while the other will do it 100 times. The problem is that both executions of `asyncIncrement()` may interfere with each other, and they may override increments made by the other instance of the coroutine. This means that while *most* executions of `main()` will print `counter [2100]`, many other executions will print values lower than 2,100:



In this example, the lack of atomicity in `counter++` results in two iterations, one of `workerA` and the other of `workerB`, increasing the value of `counter` by only one, when those two iterations should increase the value a total of two times. Each time this happens, the value will be one less than the expected 2,100.

The overlapping of the instructions in the coroutines happens because the operation `counter++` is not atomic. In reality, this operation can be broken into three instructions: reading the current value of `counter`, increasing that value by one, and then storing the result of the addition back into `counter`. The lack of atomicity in `counter++` makes it possible for the two coroutines to read and modify the value, disregarding the operations made by the other.

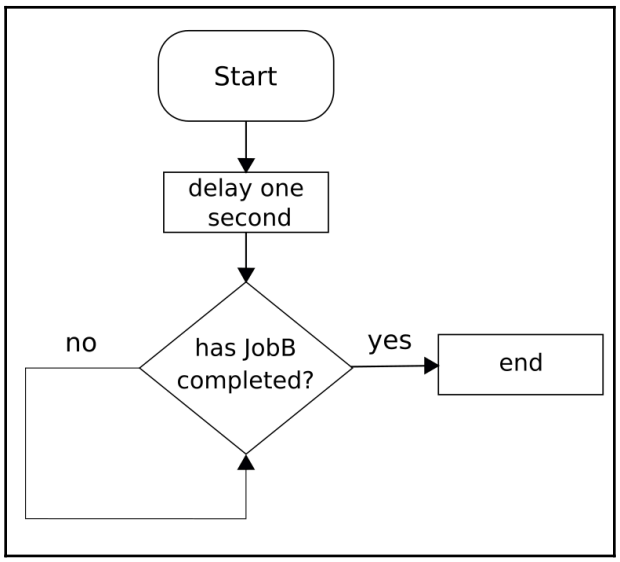
To fix this scenario, it's important to make sure that only one of the coroutines is executing `counter++` at a time, and this can be accomplished in many ways. Throughout the book, we will cover different approaches to guarantee atomic operations when needed.

Deadlocks

Often, in order to guarantee that concurrent code is synchronized correctly, it's necessary to suspend or block execution while a task is completed in a different thread. But due to the complexity of these situations, it isn't uncommon to end up in a situation where the execution of the complete application is halted because of circular dependencies:

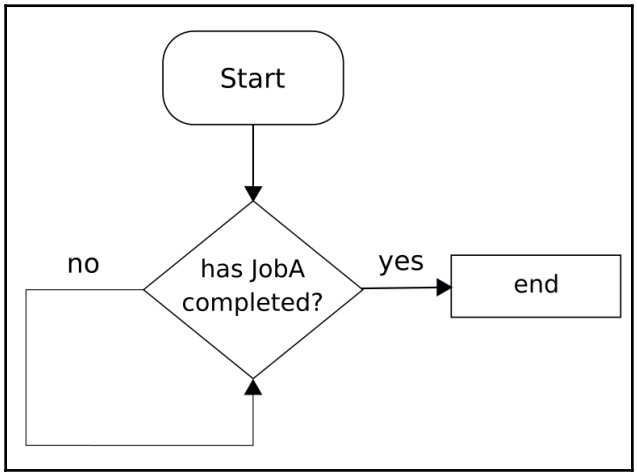
```
lateinit var jobA : Job
lateinit var jobB : Job

fun main(args: Array<String>) = runBlocking {
    jobA = launch {
        delay(1000)
        // wait for JobB to finish
        jobB.join()
     }

    jobB = launch {
        // wait for JobA to finish
        jobA.join()
    }

    // wait for JobA to finish
    jobA.join()
    println("Finished")
}
```

Let's take a look at a simple flow diagram of `jobA`.



Flow chart of jobA

In this example, `jobA` is waiting for `jobB` to finish its execution; meanwhile `jobB` is waiting for `jobA` to finish. Since both are waiting for each other, none of them is ever going to end; hence the message `Finished` will never be printed:



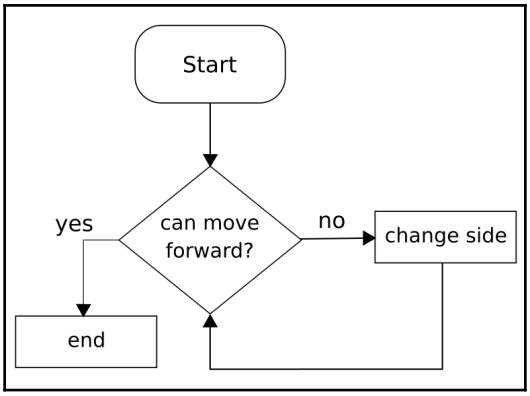
Flow chart of jobB

This example is, of course, intended to be as simple as possible, but in real-life scenarios deadlocks are more difficult to spot and correct. They are commonly caused by intricate networks of locks, and often happen hand-in-hand with race conditions. For example, a race condition can create an unexpected state in which the deadlock can happen.

Livelocks

Livelocks are similar to deadlocks, in the sense that they also happen when the application can't correctly continue its execution. The difference is that during a livelock the state of the application is constantly changing, but the state changes in a way that further prevents the application from resuming normal execution.

Commonly, a livelock is explained by picturing two people, Elijah and Susan, walking in opposite directions in a narrow corridor. Both of them try to avoid the other by moving to one side: Elijah moves to the left while Susan moves to the right, but since they are walking in opposite directions, they are now blocking each other's way. So, now Elijah moves to the right, just at the same time that Susan moves to the left: once again they are unable to continue on their way. They continue moving like this, and thus they continue to block each other:



Flow chart for the example of a livelock

In this example, both Elijah and Susan have an idea of how to recover from a deadlock—each blocking the other—but the timing of their attempts to recover further obstructs their progress.

As expected, livelocks often happen in algorithms designed to recover from a deadlock. By trying to recover from the deadlock, they may in turn create a livelock.

Concurrency in Kotlin

Now that we have covered the basics of concurrency, it's a good time to discuss the specifics of concurrency in Kotlin. This section will showcase the most differentiating characteristics of Kotlin when it comes to concurrent programming, covering both philosophical and technical topics.

Non-blocking

Threads are heavy, expensive to create, and limited—only so many threads can be created—So when a thread is blocked it is, in a way, being wasted. Because of this, Kotlin offers what is called Suspendable Computations; these are computations that can suspend their execution without blocking the thread of execution. So instead of, for example, blocking thread *X* to wait for an operation to be made in a thread *Y*, it's possible to suspend the code that has to wait and use thread *X* for other computations in the meantime.

Furthermore, Kotlin offers great primitives like channels, actors, and mutual exclusions, which provide mechanisms to communicate and synchronize concurrent code effectively without having to block a thread.

`Chapter 6`, Channels – Share Memory by Communicating, `Chapters 7`, *Thread Confinement, Actors, and Mutexes*, and `Chapter 8`, *Testing Concurrent Code*, will focus on the correct usage of channels, actors, mutexes, and more to correctly communicate and synchronize your concurrent code.

Being explicit

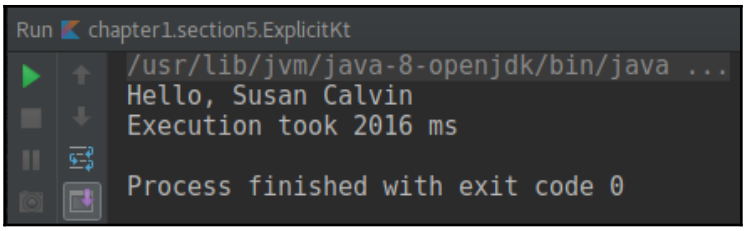
Concurrency needs to be thought about and designed for, and because of that, it's important to make it explicit in terms of when a computation should run concurrently. Suspendable computations will run sequentially by default. Since they don't block the thread when suspended, there's no direct drawback:

```
fun main(args: Array<String>) = runBlocking {
    val time = measureTimeMillis {
        val name = getName()
        val lastName = getLastName()
        println("Hello, $name $lastName")
    }
    println("Execution took $time ms")
}
```

```
suspend fun getName(): String {
    delay(1000)
    return "Susan"
}

suspend fun getLastName(): String {
    delay(1000)
    return "Calvin"
}
```

In this code, `main()` executes the suspendable computations `getName()` and `getLastName()` in the current thread, sequentially.

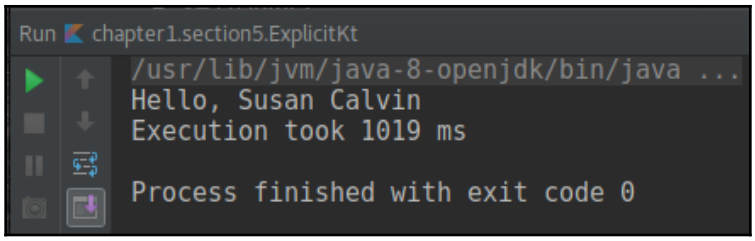
Executing `main()` will print the following:



This is convenient because it's possible to write non-concurrent code that doesn't block the thread of execution. But after some time and analysis, it becomes clear that it doesn't make sense to have `getLastName()` wait until after `getName()` has been executed since the computation of the latter has no dependency on the former. It's better to make it concurrent:

```
fun main(args: Array<String>) = runBlocking {
    val time = measureTimeMillis {
        val name = async { getName() }
        val lastName = async { getLastName() }

        println("Hello, ${name.await()} ${lastName.await()}")
    }
println("Execution took $time ms")
}
```

Now, by calling `async {...}` it's clear that both of them should run concurrently, and by calling `await()` it's requested that `main()` is suspended until both computations have a result:



Readable

Concurrent code in Kotlin is as readable as sequential code. One of the many problems with concurrency in other languages like Java is that often it's difficult to read, understand, and/or debug concurrent code. Kotlin's approach allows for idiomatic concurrent code:

```
suspend fun getProfile(id: Int) {
    val basicUserInfo = asyncGetUserInfo(id)
    val contactInfo = asyncGetContactInfo(id)

    createProfile(basicUserInfo.await(), contactInfo.await())
}
```

By convention, a function that is going to run concurrently by default should indicate this in its name, either by starting with `async` or ending in `Async`.

This suspend method calls two methods that will be executed in background threads and waits for their completion before processing the information. Reading and debugging this code is as simple as it would be for sequential code.

In many cases, it is better to write a `suspend` function and call it inside an `async {}` or `launch {}` block, rather than writing functions that are already async. This is because it gives more flexibility to the callers of the function to have a `suspend` function; that way the caller can decide when to run it concurrently, for example. In other cases, you may want to write both the concurrent and the suspend function.

Leveraged

Creating and managing threads is one of the difficult parts of writing concurrent code in many languages. As seen before, it's important to know when to create a thread, and almost as important to know how many threads are optimal. It's also important to have threads dedicated to I/O operations, while also having threads to tackle CPU-bound operations. And communicating/syncing threads is a challenge in itself.

Kotlin has high-level functions and primitives that make it easier to implement concurrent code:

- To create a thread it's enough to call `newSingleThreadContext()`, a function that only takes the name of the thread. Once created, that thread can be used to run as many coroutines as needed.
- Creating a pool of threads is as easy, by calling `newFixedThreadPoolContext()` with the size and the name of the pool.
- `CommonPool` is a pool of threads optimal for CPU-bound operations. Its maximum size is the amount of cores in the machine minus one.
- The runtime will take charge of moving a coroutine to a different thread when needed .
- There are many primitives and techniques to communicate and synchronize coroutines, such as channels, mutexes, and thread confinement.

Flexible

Kotlin offers many different primitives that allow for simple-yet-flexible concurrency. You will find that there are many ways to do concurrent programming in Kotlin. Here is a list of some of the topics we will look at throughout the book:

- **Channels**: Pipes that can be used to safely send and receive data between coroutines.
- **Worker pools**: A pool of coroutines that can be used to divide the processing of a set of operations in many threads.
- **Actors**: A wrapper around a state that uses channels and coroutines as a mechanism to offer the safe modification of a state from many different threads.

- **Mutual exclusions (Mutexes)**: A synchronization mechanism that allows the definition of a *critical zone* so that only one thread can execute at a time. Any coroutine trying to access the critical zone will be suspended until the previous coroutine leaves.
- **Thread confinement**: The ability to limit the execution of a coroutine so that it always happens in a specified thread.
- **Generators (Iterators and sequences)**: Data sources that can produce information on demand and be suspended when no new information is required.

All of these are tools that are at your fingertips when writing concurrent code in Kotlin, and their scope and use will help you to make the right choices when implementing concurrent code.

Concepts and terminology

To end this chapter, we will cover some basic concepts and terminology when talking about concurrency in Kotlin; these are important in order to more clearly understand the upcoming chapters.

Suspending computations

Suspending computations are those computations that can suspend their execution without blocking the thread in which they reside. Blocking a thread is often inconvenient, so by suspending its own execution, a suspending computation allows the thread to be used for other computations until they need to be resumed.

Because of their nature, suspending computations can only be invoked from other suspending functions or from coroutines.

Suspending functions

Suspending functions are suspending computations in the shape of a function. A suspending function can be easily identified because of the modifier `suspend`. For example:

```
suspend fun greetAfter(name: String, delayMillis: Long) {
    delay(delayMillis)
    println("Hello, $name")
}
```

In the previous example, the execution of `greetAfter()` will be suspended when `delay()` is called – `delay()` is a suspending function itself, suspending the execution for a give duration. Once `delay()` has been completed, `greetAfter()` will resume its execution normally. And while `greetAfter()` is suspended, the thread of execution may be used to do other computations.

In Chapter 9, *The Internals of Concurrency in Kotlin*, we will talk about how this works behind the scenes.

Suspending lambdas

Similar to a regular lambda, a suspending lambda is an anonymous local function. The difference is that a suspending lambda can suspend its own execution by calling other suspending functions.

Coroutine dispatcher

In order to decide what thread to start or resume a coroutine on, a coroutine dispatcher is used. All coroutine dispatchers must implement the `CoroutineDispatcher` interface:

- `DefaultDispatcher`: Currently it is the same as `CommonPool`. In the future, this may change.
- `CommonPool`: Executes and resumes the coroutines in a pool of shared background threads. Its default size makes it optimal for use in CPU-bound operations.
- `Unconfined`: Starts the coroutine in the current thread (the thread from which the coroutine was invoked), but will resume the coroutine in any thread. No thread policy is used by this dispatcher.

Along with these dispatchers, there are a couple of builders that can be used to define pools or threads as needed:

- `newSingleThreadContext()` to build a dispatcher with a single thread; the coroutines executed here will be started and resumed in the same thread, always.
- `newFixedThreadPoolContext()` creates a dispatcher with a pool of threads of the given size. The runtime will decide which thread to use when starting and resuming the coroutines executed in the dispatcher.

Coroutine builders

A coroutine builder is a function that takes a suspending lambda and creates a coroutine to run it. Kotlin provides many coroutine builders that adjust to many different common scenarios, such as:

- `async()`: Used to start a coroutine when a result is expected. It has to be used with caution because `async()` will capture any exception happening inside the coroutine and put it in its result. Returns a `Deferred<T>` that contains either the result or the exception.
- `launch()`: Starts a coroutine that doesn't return a result. Returns a `Job` that can be used to cancel its execution or the execution of its children.
- `runBlocking()`: Created to bridge blocking code into suspendable code. It's commonly used in `main()` methods and unit tests. `runBlocking()` blocks the current thread until the execution of the coroutine is completed.

Here is an example of `async()`:

```
val result = async {
    isPalindrome(word = "Sample")
}
result.await()
```

In this example, `async()` is executed in the default dispatcher. It is possible to manually specify the dispatcher:

```
val result = async(Unconfined) {
    isPalindrome(word = "Sample")
}
result.await()
```

In this second example, `Unconfined` is used as the dispatcher of the coroutine.

Summary

This chapter was a detailed introduction to many important concepts and tools involved in concurrency, and acts as a foundation for the upcoming chapters. So here's a summary to help you remember some of the most important points that we discussed:

- An application has one or more processes; each of them has at least one thread, and coroutines are executed inside a thread.
- A coroutine can be executed in a different thread each time it's resumed, but can also be confined to a specific thread.
- An application is concurrent when its execution happens in more than one overlapping thread.
- To write correct concurrent code, it's necessary to learn how to communicate and synchronize the different threads, which in Kotlin implies the communication and synchronization of coroutines.
- Parallelism happens when, during the execution of a concurrent application, at least two threads are effectively being executed simultaneously.
- Concurrency can happen without parallelism because modern processing units will interleave between threads, effectively overlapping threads.
- There are many challenges when it comes to writing concurrent code, most of them related to correct communication and the synchronization of threads. Race conditions, atomicity violations, deadlocks, and livelocks are examples of the most common challenges.
- Kotlin has taken a modern and fresh approach to concurrency. With Kotlin, it's possible and encouraged to write non-blocking, readable, leveraged, and flexible concurrent code.

In the next chapter, we will work with coroutines. We will start by configuring Android Studio and creating a project that supports coroutines. Then we will write and run concurrent code in an Android emulator, using coroutines for real-world scenarios such as REST calls. We will look at practical examples of suspending functions.

2
Coroutines in Action

It's time to start coding. In this chapter, we will go through the process of enabling support for coroutines in an Android project using Android Studio, and we will use coroutines to solve a common scenario for mobile apps: calling a REST service and displaying some of the response without blocking the UI thread.

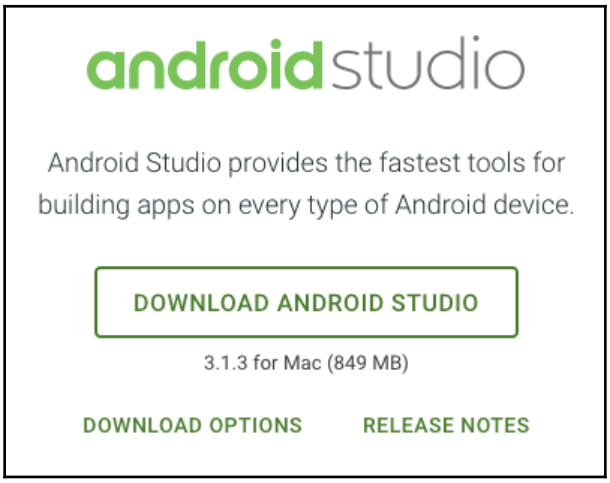
Topics that will be covered in this chapter include:

- Android Studio configuration for Kotlin and projects using coroutines
- Android's UI thread
- REST calls in a background thread using coroutines
- Coroutine builders `async()` and `launch()`
- Introduction to coroutine dispatchers

Downloading and installing Android Studio

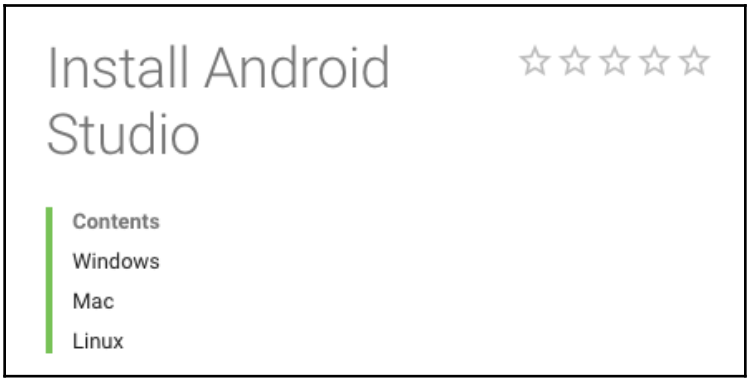
The first step for this chapter is to install Android Studio. For this, you can head to `https://developer.android.com/studio/index.html` and download the installer for your platform.

Kotlin support was added to Android Studio in version 3.0. If you already have Android Studio installed, please check that the version is at least 3.0. It's recommended that you update to the newest stable version of Android Studio if you have an earlier version.



Downloading Android Studio from the official web page

Once you have finished the download, follow the steps on the official guide at `https://developer.android.com/studio/install.html` to complete the installation. By default, the instructions are going to be according to the operating system detected by the web page; if you wish to see the instructions for a different OS, please select the relevant option in the menu on the right of the page, shown as follows:



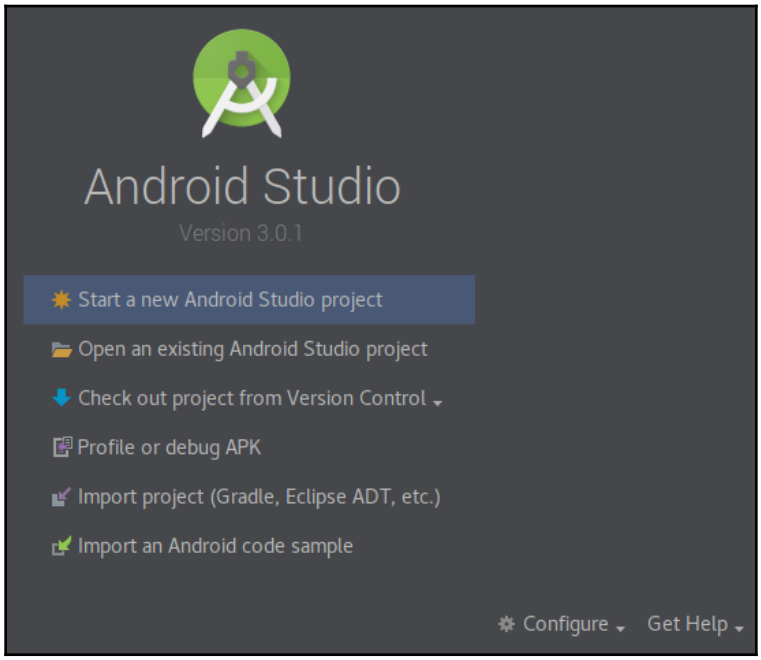
You can switch the instructions to other operating systems if you want

Once the installation is complete, you will be able to create Android projects with Kotlin support out of the box.

Keep in mind that the preceding screenshots are according to the current design of the page. Google updates both pages frequently, so they may look slightly different when you are downloading and installing Android Studio, but the premise should be the same: download and follow the steps according to your operating system.

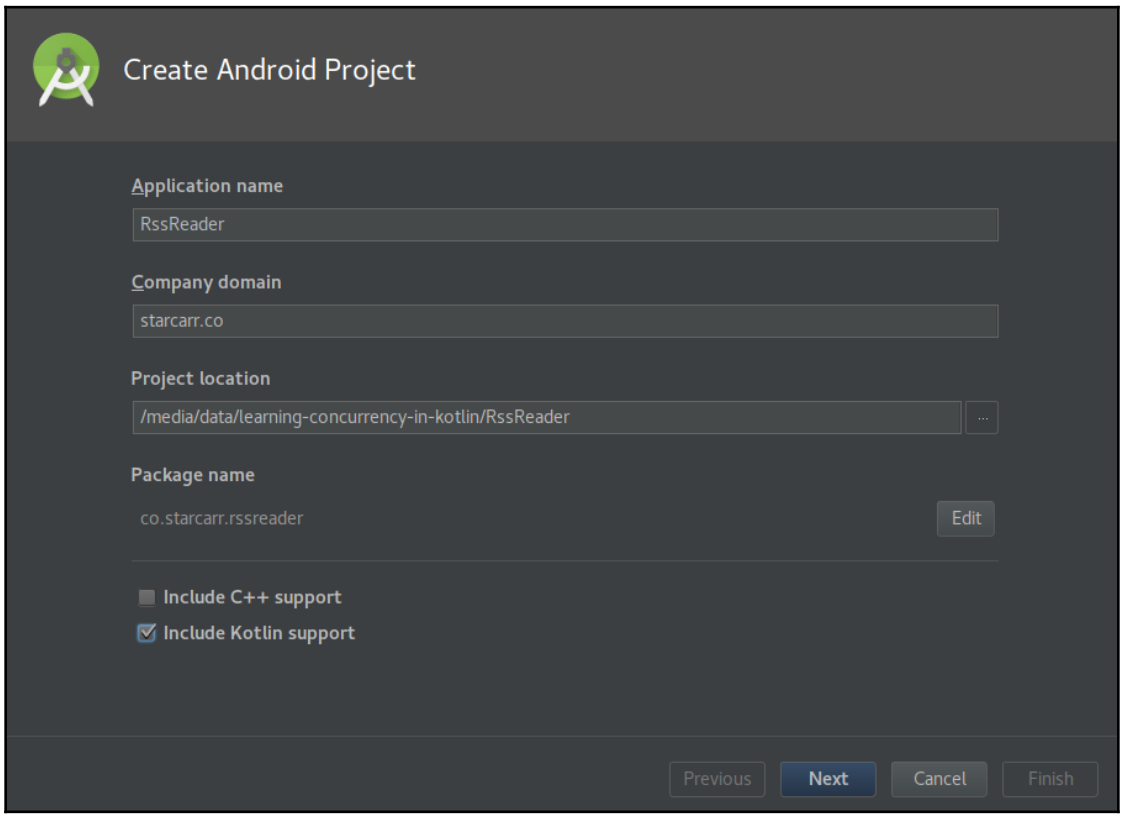
Creating a Kotlin project

When you first open Android Studio, you will be presented with a wizard. To start, select **Start a new Android Studio project**,as shown in the following screenshot:



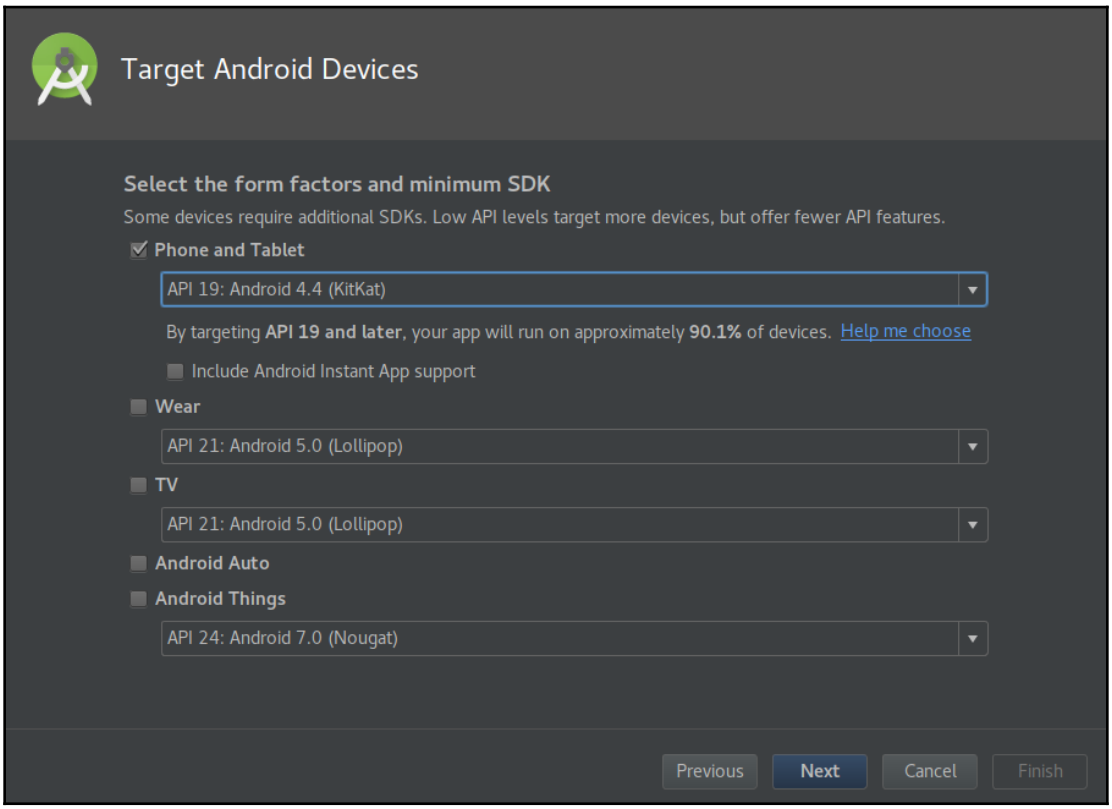
Android Studio wizard

Now you will be asked to define the properties of the project. Please make sure that the **Include Kotlin support** option is enabled:



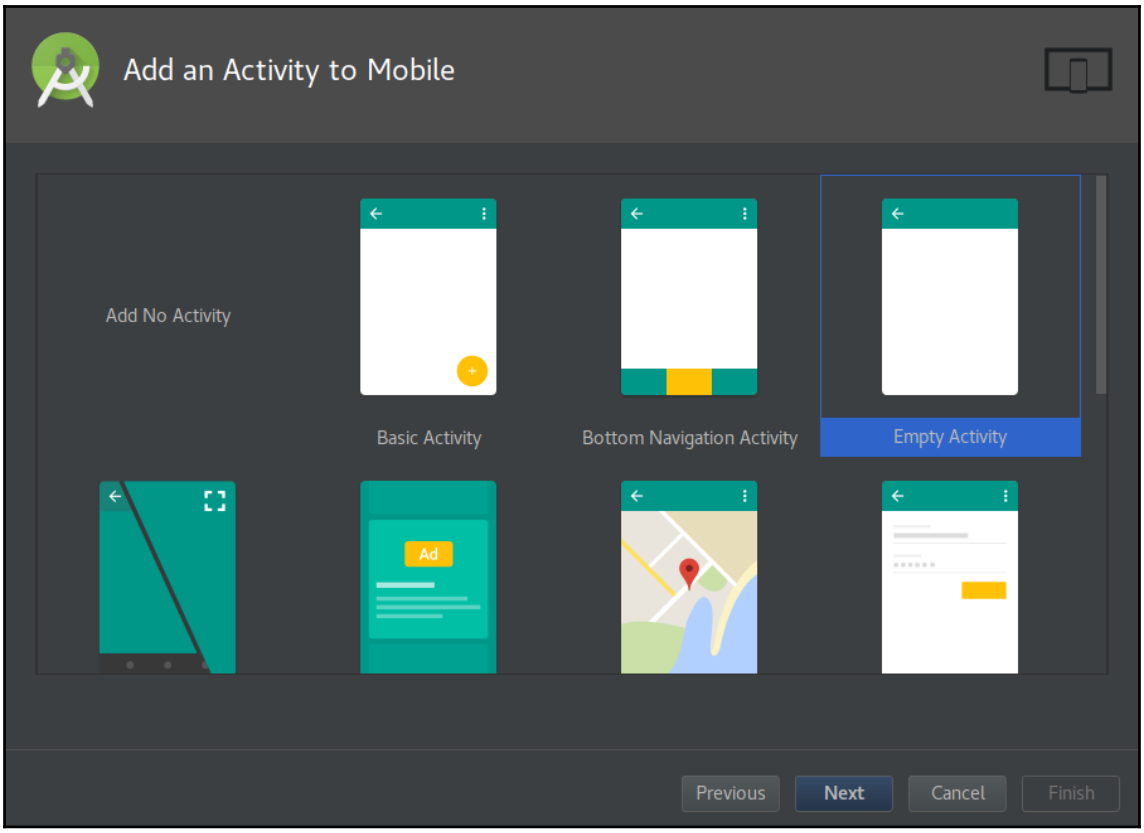
Setting the properties of a new Android project with Kotlin support

The next step is to select the Android version that you are targeting. The default options are good for this step:



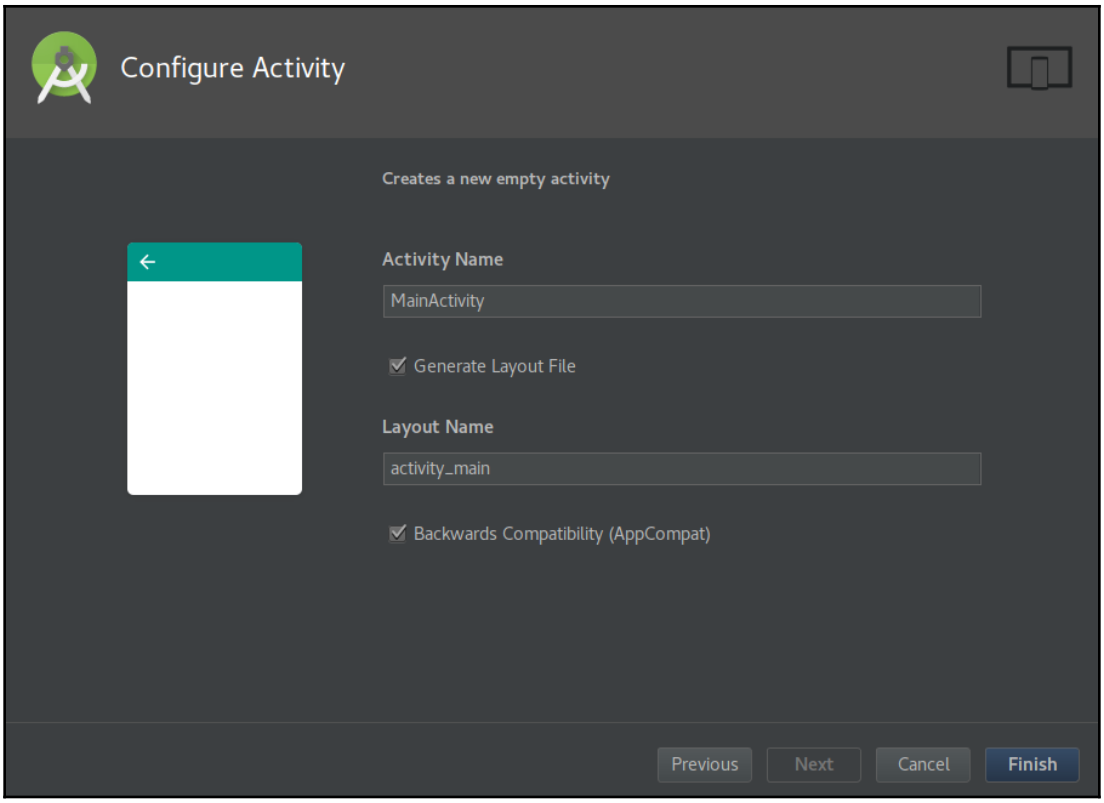
Selecting the target SDKs

Then, select the option to have just an empty activity, and leave its configuration as default:



Selecting an empty activity

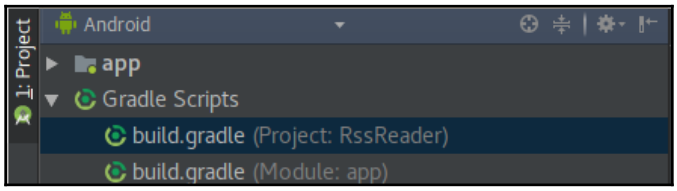
Now you will be prompted to select the name for the class and the layout of the activity. Please write `MainActivity` as the name of the activity, while setting `activity_main` as the name of the layout:



Configuring the default activity

Adding support for coroutines

Now that the project has been created, it's time to add coroutines support. To do this, open the **Project** section and double-click in **build.gradle** for the project. It is located inside the **Gradle Scripts** section:



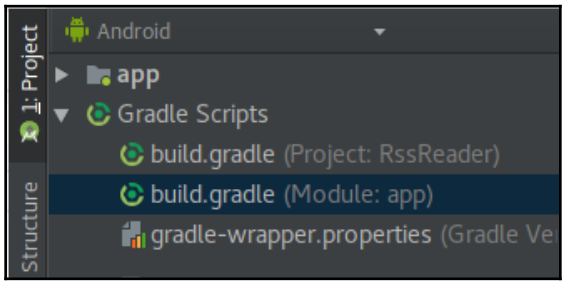
The correct build.gradle file says Project: RssReader inside parenthesis

Inside this document, you will add a new variable to centralize the coroutines version. In a new line below `ext.kotlin.version`, add `ext.coroutines_version`, as shown here, for example:

```
buildscript {
  ext.kotlin_version = '1.2.50'
  ext.coroutines_version = '0.23.3'
  ...
}
```

Every time you modify the **build.gradle** files, Android Studio will ask for the configuration to be synchronized again. Until the synchronization is made, the changes will not be applied.

Now you need to add the dependencies for coroutines. Append the dependency configuration in the **build.gradle** file for the Module:



The build.gradle file for the Module

First, we will add the dependency for coroutines:

```
dependencies {
  implementation fileTree(dir: 'libs', include: ['*.jar'])
  implementation "org.jetbrains.kotlin:kotlin-stdlib-
  jdk7:$kotlin_version"
  implementation "org.jetbrains.kotlinx:kotlinx-coroutines-
  core:$coroutines_version"
  <...>
}
```

The application can be executed either on an emulator or on a real device from within Android Studio or from the command line. Android Studio's User Guide offers excellent information on how to do this, in their documentation at `https://developer.android.com/studio/run/index.html`

Currently, coroutines are considered an experimental feature. In order to remove any warning when compiling or running apps using coroutines, it's necessary to also append this to the `build.gradle` file for the Module:

```
kotlin {
  experimental {
    coroutines "enable"
  }
}
```

Once coroutines are no longer considered experimental, you can safely avoid adding the previous snippet of code.

Android's UI thread

As mentioned in the first chapter, Android applications have a thread that is dedicated to updating the UI and also to listening to user interactions and processing the events generated by the user – like the user clicking on a menu.

Let's review the basics of Android's UI thread to guarantee that we separate the work between the UI and background threads in the best way possible.

CalledFromWrongThreadException

Android will throw a `CalledFromWrongThreadException` whenever a thread that didn't create a view hierarchy tries to update its views. In practical terms, this exception will happen whenever a thread other than the UI thread updates a view. This is because the UI thread is the only one that can create view hierarchies, so it will always be the one that can update them.

It's important to guarantee that code that will update the UI is running in the UI thread. In our example, we will update a label on the UI thread after calling a service on a background thread.

NetworkOnMainThreadException

In Java, network operations are blocking by nature. Calling a web service, for example, will block the current thread until a response is received or a timeout/error happens. Because blocking the UI thread would mean *freezing* all the UI – including animations and any other interactions – Android crashes whenever a network operation is made on the UI thread. The `NetworkOnMainThreadException` exception will be thrown whenever this happens, forcing developers to use a background thread, and thus improving the user experience.

Requesting in the background, updating in the UI thread

Putting these two things together, it's clear that in order to be able to implement our service call we need to have a background thread do the call to the web service, and have the UI thread update the UI once the response is processed.

Creating a thread

Kotlin has simplified thread creation so that it's a straightforward process and can be done easily. For now, creating a single thread will be enough, but in future chapters we will also create thread pools in order to run both CPU-bound and I/O-bound operations efficiently.

CoroutineDispatcher

It's important to understand that in Kotlin, while you can create threads and thread pools easily, you don't access or control them *directly*. What you do is create a `CoroutineDispatcher`, which is basically an orchestrator that distributes coroutines among threads based on availability, load, and configuration.

In our current case, for example, we will create a `CoroutineDispatcher` that has only one thread, so all the coroutines that we attach to it will be running in that specific thread. To do so, we create a `ThreadPoolDispatcher` – which extends `CoroutineDispatcher` – with only one thread.

Let's open the `MainActivity.kt` file that was generated previously – it's in the main package of the app, `co.starcarr.rssreader` – and update it so that it creates such a dispatcher on the class level:

```
class MainActivity : AppCompatActivity() {
  val netDispatcher = newSingleThreadContext(name = "ServiceCall")
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
  }
}
```

In this and many other cases throughout the book, modifiers such as `private`, `protected`, `open`, and others may not be written in the examples as a way to reduce the code – thus improving the formatting for the book. Please take into account that the best practice is to always make each member as inaccessible as possible. Also, in many cases the correct modifier will be found in the code files that come with the book, since there's no formatting constraint there.

Attaching a coroutine to a dispatcher

Now that we have a dispatcher, we can easily start a coroutine using it – which will enforce the coroutine to use the thread we defined. At this point, we will look at two different ways to start a coroutine. The difference between both is crucial for result and error handling.

Starting a coroutine with async

When a coroutine is started with the intention of processing its result, `async()` must be used. It will return a `Deferred<T>`, where `Deferred` is a non-blocking cancellable future – provided by the coroutines' framework – and `T` is the type of the result.

When using `async`, you must not forget to process its result – which can easily happen, so be wary.

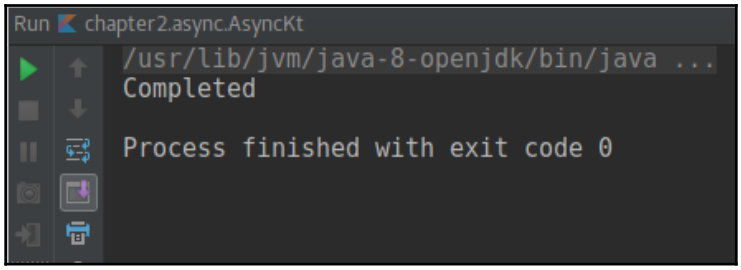
Consider the following code:

```
fun main(args: Array<String>) = runBlocking {
  val task = async {
    doSomething()
  }
  task.join()
  println("Completed")
}
```

Here, `doSomething()` is simply throwing an exception:

```
fun doSomething() {
  throw UnsupportedOperationException("Can't do")
}
```

You may be tempted to assume that this will stop the execution of the application, the stack of the exception will be printed, and the app's exit code will be different from zero – zero means that no error happened, anything different means error. Let's see what happens when this is executed:



As you can see, there is no exception stack trace being printed in logs, the application didn't crash, and the exit code indicates successful execution.

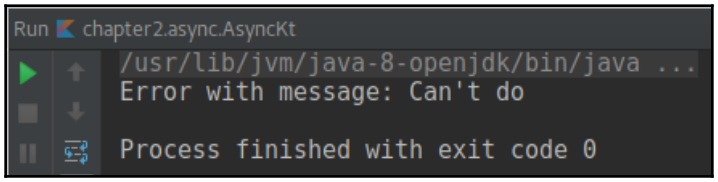
This is because any exception happening inside an `async()` block will be attached to its result, and only by checking there you will find the exception. For this, the `isCancelled` and `getCancellationException()` methods can be used together to safely retrieve the exception. The previous example can be updated so that it validates the exception, like shown here:

```
fun main(args: Array<String>) = runBlocking {
  val task = async {
    doSomething()
  }
  task.join()
  if (task.isCancelled) {
```

```
    val exception = task.getCancellationException()
    println("Error with message: ${exception.message}")
  } else {
    println("Success")
  }
}
```

At the time of writing, there is an effort from the Kotlin team to make the code above work as explained here. If you notice that `isCancelled` is returning `false` in the scenario above, please replace it with `isCompletedExceptionally`. I decided to avoid mentioning `isCompletedExceptionally` because it's being deprecated close to the release of the book. For more information, please read `issue 220` in the coroutines' GitHub repository.

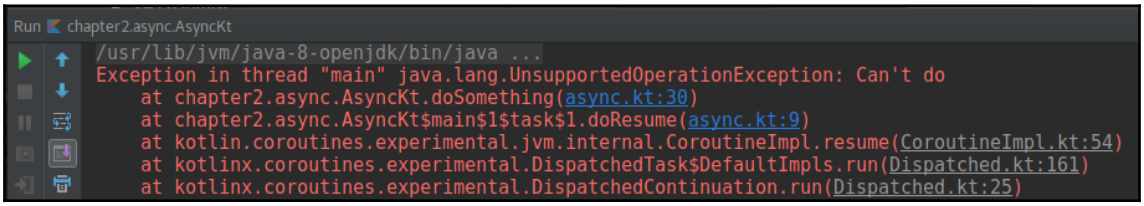
This will then print the following error:



In order to propagate the exception, `await()` can be called on `Deferred`, for example:

```
fun main(args: Array<String>) = runBlocking {
  val task = async {
    doSomething()
  }
  task.await()
  println("Completed")
}
```

This will crash the application:



This crash happens because by calling `await()`, we are unwrapping `Deferred`, which in this case will unwrap to the exception and propagate it.

The main difference between waiting with `join()` and then validating and processing any error, and calling `await()` directly, is that in the first option we are handling the exception without having to propagate it, whereas by just calling `await()` the exception will be propagated.

For this reason, the example using `await()` will return the code *one*, meaning an error during the execution, whereas waiting with `join()` and using `isCancelled` and `getCancellationException()` to handle the error will result in a successful code of *zero*.

In `Chapter 3`, *Life Cycle and Error Handling*, we will talk more about proper exception handling and propagation depending on the expected behavior.

Starting a coroutine with launch

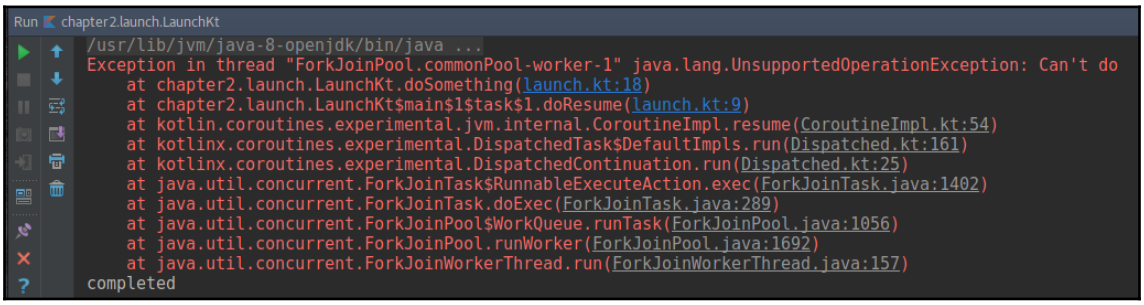
When your objective is to start a coroutine that doesn't return a result, you should use `launch()`. It's designed for *fire-and-forget* scenarios where you only want to be notified if the computation fails, and it still provides you with a function to cancel it if needed. Consider this example:

```
fun main(args: Array<String>) = runBlocking {
  val task = launch {
    doSomething()
  }
  task.join()
  println("completed")
}
```

Here, `doSomething()` throws an exception:

```
fun doSomething() {
  throw UnsupportedOperationException("Can't do")
}
```

The exception will be printed to the stack as expected, but notice that the execution was not interrupted and the application finished the execution of `main()`:



As we will see in `chapter 3`, *Life Cycle and Error Handling*, the default behavior for uncaught exceptions is defined per platform, but can be overwritten.

Using a specific dispatcher when starting the coroutine

So far, we have seen how to create coroutines using `async()` and `launch()`, but in both cases we were using the default dispatcher. Consider the following code:

```
fun main(args: Array<String>) = runBlocking {
  val task = launch {
    printCurrentThread()
  }
  task.join()
}
```

Here, `printCurrentThread()`, as its name suggests, will just print the name of the current thread to the standard output:

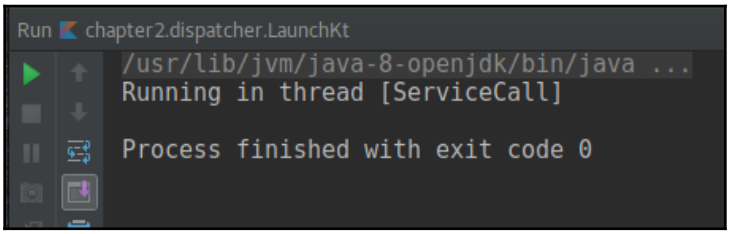
```
fun printCurrentThread() {
  println("Running in thread [${Thread.currentThread().name}]")
}
```

By running this code, we see that by default the coroutine will run in `DefaultDispatcher`, which as the time of writing is the same dispatcher as `CommonPool` – but be aware that this could change in the future.

If we update `main()` to create a `CoroutineDispatcher` the same way we did before, and send it to `launch()`, we will see that the coroutine is being executed in the thread that we indicated, for example:

```
fun main(args: Array<String>) = runBlocking {
  val dispatcher = newSingleThreadContext(name = "ServiceCall")
  val task = launch(dispatcher) {
    printCurrentThread()
  }
  task.join()
}
```

The output looks like the following screenshot. Notice that the name of the thread is the same name that we set for the dispatcher:



Now we will do likewise in `MainActivity`:

```
private val dispatcher = newSingleThreadContext(name = "ServiceCall")
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_main)
  launch(dispatcher) {
    // TODO Call coroutine here
  }
}
```

Adding networking permissions

Android requires applications to explicitly request permissions in order to access many features. This is done in order to present the user with an option to deny specific permissions, and prevent applications from doing something different from what the user expects.

Since we will be doing network requests, we will need to add the *internet* permission to the manifest of the application. Let's locate the `AndroidManifest.xml` file in the `app/src/main` directory, and edit it:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="co.starcarr.rssreader">
  <uses-permission android:name="android.permission.INTERNET" />
  <application
  ...
</manifest>
```

The name *Internet* is not accurate. This permission is needed in order to do any network request regardless of it actually reaching the internet – for example, it could be a LAN request. This permission has to be thought of as *Networking*.

Creating a coroutine to call a service

Now it's a good time to add a call to a service. To start with something simple, we will use Java's `DocumentBuilder` to call an RSS feed for us. First, we will add a variable to hold the `DocumentBuilderFactory` below where we put the dispatcher:

```
private val dispatcher = newSingleThreadContext(name = "ServiceCall")
private val factory = DocumentBuilderFactory.newInstance()
```

The second step is to create a function that will do the actual call:

```
private fun fetchRssHeadlines(): List<String> {
  val builder = factory.newDocumentBuilder()
  val xml = builder.parse("https://www.npr.org/rss/rss.php?id=1001")
  return emptyList()
}
```

Notice how the function is, for now, returning an empty list of strings after calling the feed. The idea is to implement this function so that it returns the headlines of the given feed. But, first let's call this function as part of the coroutine defined previously:

```
launch(dispatcher) {
  fetchRssHeadlines()
}
```

Now, the fetching of the headlines will happen in the thread of `dispatcher`. The next step is to actually read the body of the response and return the headlines. For this example, we are going to parse the XML by hand – as opposed to using some library to do the parsing:

```
private fun fetchRssHeadlines(): List<String> {
  val builder = factory.newDocumentBuilder()
  val xml = builder.parse("https://www.npr.org/rss/rss.php?id=1001")
  val news = xml.getElementsByTagName("channel").item(0)
  return (0 until news.childNodes.length)
      .map { news.childNodes.item(it) }
      .filter { Node.ELEMENT_NODE == it.nodeType }
      .map { it as Element }
      .filter { "item" == it.tagName }
      .map {
          it.getElementsByTagName("title").item(0).textContent
      }
}
```

This code is simply going through all elements in the XML and filtering out everything but the title of each article in the feed.

Both `Element` and `Node` are from the package `org.w3c.dom`.

Now that the function is actually returning the information, we can receive it in preparation to display it on the user interface:

```
launch(dispatcher) {
  val headlines = fetchRssHeadlines()
}
```

Adding UI elements

Now we can start adding some UI elements to do some testing. First, let's update the layout of `MainActivity`, which is located in `res/layout/activity_main.xml`. For now, just adding a `ProgressBar` will do, so let's replace the contents of the `ConstraintLayout` so that a `ProgressBar` is located in the middle of the screen:

```
<android.support.constraint.ConstraintLayout ...>
    <ProgressBar
        android:id="@+id/progressBar"
        style="?android:attr/progressBarStyle"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</android.support.constraint.ConstraintLayout>
```

By running the application, you will now only see the circular progress bar infinitely spinning. Let's use that spinner to see how the wrong handling of threads can affect the UI.

What happens when the UI is blocked

It's a good opportunity to take a practical approach in order to further understand why the UI thread of an Android application should not be blocked. Let's add this simple block of code to `MainActivity`, and run the application again:

```
override fun onResume() {
  super.onResume()
  Thread.sleep(5000)
}
```

This will block the UI thread for five seconds. If you run the application again, you will notice how the screen is completely white during those five seconds. This would be dreadful for the user of the application.

Not only should the UI thread never be blocked, it should also not perform tasks that are CPU-intensive because this would result in a similar experience for the user. In short, you should only use the UI thread to create and update views, anything in between should be done in a background thread.

Displaying the amount of news that were processed

Let's put a `TextView` in our layout and display the amount of news that were processed from the feed. Notice that the `TextView` will be located below the progress bar because of the property `app:layout_constraintBottom_toBottomOf`:

```
<android.support.constraint.ConstraintLayout ...>
    <ProgressBar ...>
    <TextView
        android:id="@+id/newsCount"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="20dp"
        app:layout_constraintTop_toBottomOf="@id/progressBar"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent" />

</android.support.constraint.ConstraintLayout>
```

In order to display the amount of news, we will obtain the `TextView` by its identifier and set the text to be the amount of news that were obtained:

```
launch(dispatcher) {
  val headlines = fetchRssHeadlines()
  val newsCount = findViewById<TextView>(R.id.newsCount)
  newsCount.text = "Found ${headlines.size} News"
}
```

If executed, this code will crash the application with a `CalledFromWrongThreadException`, as explained earlier in the chapter. This makes sense because all the content of our coroutine is being executed in a background thread, and UI updates must happen on the UI thread.

Using a UI dispatcher

In the same way that we used a `CoroutineDispatcher` to run a thread in the background, we can now use one to preform operations in the main thread.

Platform-specific UI libraries

Given that there are many types of GUI applications for the JVM, Kotlin has separated some platform-specific coroutine functionality into libraries.

- `kotlinx-coroutines-android`
- `kotlinx-coroutines-javafx`
- `kotlinx-coroutines-swing`

Notice that all these platforms have the same UI model that we talked about before, in which only the UI thread can create and update the views. So, these tiny libraries are simply a `CoroutineDispatcher` that is implemented to confine coroutines to the UI thread.

The Android library also adds support for some exception pre-handling at thread level that is specific to Android.

Adding the dependency

Now that we know which library we need, we will simply add it to our gradle file for the module, below the dependency for coroutines.

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

Using Android's UI coroutine dispatcher

With this done, we can now use the dispatcher in the same way we use any other, for example:

```
launch(dispatcher) {
  val headlines = fetchRssHeadlines()
  val newsCount = findViewById<TextView>(R.id.newsCount)
  launch(UI) {
    newsCount.text = "Found ${headlines.size} News"
  }
}
```

The `UI` dispatcher comes from the library we just added, *kotlinx-coroutines-android*.

Executing the code at this point will work. The quantity of news will be correctly displayed on the application.

Creating an asynchronous function to hold the request... or not

Currently, a big part of our code to request and display the number of news is inside the `onCreate()` function. This is less than optimal, not only because it's mixed with the creation of the activity, but also because it prevents reusing this code. For example, if there were to be a refresh button, we would need to reuse all the code of the coroutine.

When considering the separation of this coroutine into its own function, there are many possible approaches. Here, we will cover the most common ones.

A synchronous function wrapped in an asynchronous caller

The first approach is quite simple. We can create a `loadNews()` function that directly invokes `fetchRssHeadlines()` and displays its results in the same way we did before:

```
private fun loadNews(){
  val headlines = fetchRssHeadlines()
  val newsCount = findViewById<TextView>(R.id.newsCount)
  launch(UI) {
    newsCount.text = "Found ${headlines.size} News"
  }
}
```

Notice that this function is synchronous, and will call `fetchRssHeadlines()` in the same thread that it's called. Consider this code, for example:

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_main)
  loadNews()
}
```

Doing this will result in a `NetworkOnMainThreadException` being thrown. Since `loadNews()` uses the same thread it's being called on, the request to fetch the feed will happen in the UI thread. To fix this, we can then wrap the call to `loadNews()` in a `launch()` block like we did before, using the dispatcher that was created for the service request. The code would be almost the same:

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_main)
  launch (dispatcher) {
    loadNews()
  }
}
```

This code is quite explicit, indicating that there is some code being executed asynchronously, which is a great practice. It's also quite flexible, because if the caller of `loadNews()` is already in a background thread, it can fetch the news in the same background thread, without having to use a `launch()` or `async()` builder.

The disadvantage of this approach is that if we have many places calling `loadNews()` from the UI thread, we would have many similar blocks of `launch(dispatcher) {...}` spread in the code, which can make the code less readable.

An asynchronous function with a predefined dispatcher

This brings us to the second option. We can write a function, `asyncLoadNews()`, that will contain the `launch()` inside it and return the resulting `Job`. This function can be called without a `launch()` block regardless of the thread; and by returning the `Job`, it's possible for the caller to cancel it:

```
private fun asyncLoadNews() = launch(dispatcher) {
  val headlines = fetchRssHeadlines()
  val newsCount = findViewById<TextView>(R.id.newsCount)
  launch(UI) {
    newsCount.text = "Found ${headlines.size} News"
  }
}
```

By stipulating the signature of a function as `fun asyncDo() = launch {...}`, not only are we making the whole function run inside a coroutine but we are also having it return the `Job` created for the operation. All the coroutine builders can be used in a similar fashion, but they will return according to their own signature. For example, `fun asyncDo() = async {...}` will return a `Deferred<T>`, and `runBlocking {...}` will return `T`.

This function can be called from anywhere in your code without needing to be wrapped, like this:

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_main)
  asyncLoadNews()
}
```

This simplifies the code if the function is called in many places, and it also reduces the flexibility of the function since it will forcefully be executed in a background thread.

It also has the disadvantage that the readability of the code will depend on the correct naming of the function. For example, if you name this implementation `loadNews()`, the caller of the function may not be aware that this will run asynchronously, and they may not wait for it to complete, which could lead to race conditions or other concurrency issues.

This approach relies on the correct naming of the functions. Incorrect naming will easily cause bugs for the people using the function, since they may not realize that they should suspend their code until the call completes.

An asynchronous function with a flexible dispatcher

We can bring back some flexibility to this function by making the dispatcher an optional parameter of the function:

```
private val defDsp = newSingleThreadContext(name = "ServiceCall")
private fun asyncLoadNews(dispatcher: CoroutineDispatcher = defDsp) =
launch(dispatcher) {
    ...
}
```

This approach has more flexibility, since now the caller can use a specific `CoroutineDispatcher` to run this code, but it still has the downside of being explicit only if an appropriate name is given to the function.

How to decide which option is better

Let's do a summary of the options that we have:

- Sync function wrapped with a coroutine: The biggest advantage is how explicit it is, but in doing so it's quite verbose.
- An async function with a specific dispatcher: While being less verbose, it's not as flexible because the caller can't decide which dispatcher should be used – which could be good if you really want to force the coroutine to a specific thread. Being explicit about it the function being asynchronous depends on the developer, which is less than ideal.
- An async function with a flexible dispatcher: Allows the caller to decide where to run the coroutine, but still depends on the developer giving a proper name to the function.

The best decision depends on the specific situation, and there's no silver bullet that will be perfect for all the different scenarios. That being said, here are my takes on the issue.

- Is there a platform limitation? We know that in Android we can't make network requests on the UI thread, so it becomes convenient to use an async function when the code is going to do networking to avoid trying to do the call in the wrong thread.
- Is the function going to be called from many places? Having to wrap a sync function with a `launch()` or `async()` block is fine if you will do it a few times; it makes concurrency explicit while being readable enough. But, if you see yourself spraying the same snippet all over your class, it may be more readable to make it into an async function.
- Do you want the caller to be able to decide what dispatcher should be used? In some cases, you want to enforce that certain code runs in a specific dispatcher – for example, to avoid atomicity violation– regardless of what the caller wants; in this case an async function with a specific dispatcher is the way to go.
- Can it be guaranteed that the naming will be correct? Avoid using async functions if you know that you can't enforce all your team to prefix (or suffix) the async functions to clearly state that they are asynchronous. It's worse to have code that crashes because the naming of a concurrent function is not clear than having to be verbose when calling said function. Fixing the former will usually require more time than doing the latter, too.
- It shouldn't be necessary for you to provide both a synchronous implementation and an asynchronous one for the same function, so avoid it at all costs. I can assure you that the bad side of both approaches will end up happening. The only exception to this could be if you are writing a library; in that case it *may* make sense to provide both options for flexibility and have the users decide what adjusts to their needs.
- Avoid heavy mixing of approaches in the same project. All the approaches are valid, but try your best to stick to one through your code base for consistency. Inconsistency and lack of standards not only affect the readability of code, but are also the root of many bugs.

Summary

This chapter covered many interesting topics of practical concurrency in Kotlin. Let's do a recap of the key topics.

- Android applications will throw `NetworkOnMainThreadException` if a network request is done on the UI thread.
- Android applications can only update the UI on the UI thread, and trying to do it from a different thread will produce a `CalledFromWrongThreadException`.
- Network requests have to be done in a background thread. The information has to be sent to the UI thread for the views to be updated.
- A `CoroutineDispatcher` can be used to enforce a coroutine to run in a specific thread, or group of threads.
- One or many coroutines can be run in a thread by using `launch` or `async`.
- `launch` should be used in fire-and-forget scenarios, meaning cases where we aren't expecting the coroutine to return something.
- `async` should be used when the coroutine will produce a result that will be processed. Using `async` without handling its result can result in exceptions not being propagated.
- Kotlin has specific libraries for Android, Swing, JavaFX, and more. Each of them offers a proper coroutine dispatcher to update UI elements.
- There are many ways to write concurrent code, and it's important to understand how to make the best of Kotlin's flexibility while being explicit, safe, and consistent.

In the next chapter, we will cover what a `Job` is in more detail, how to create a hierarchy of jobs to control chained events, and how to use chaining to stop jobs when they are no longer needed. We will also work on displaying the news obtained from the feed in a recycler view and in adding error handling when a feed can't be reached.

3
Life Cycle and Error Handling

Now that we have an application using coroutines in order to do network requests, it's time to add new features to it while putting into practice new concepts and improving the user experience.

In this chapter, we will start by taking a closer look at two types of asynchronous tasks: *Job* and *Deferred*. We will talk about their similarities and their differences, taking a closer look at their life cycle. We will also cover how to calculate their current state and what to expect in each state. Then, we will improve our RSS reader by having it fetch news concurrently from many news outlets and using the newly introduced topics to handle exceptions when a feed can't be reached.

Some of the topics we will cover during this chapter are as listed:

- Jobs and their use cases
- Life cycle of Job and Deferred
- Use cases for Deferred
- What to expect with each status of a Job
- How to calculate the current status of a Job
- How to handle exceptions

Job and Deferred

We can divide our asynchronous functions into two groups:

- Those without a result. Common scenarios are background tasks that write to a log, send analytics, and similar tasks. This type can include background tasks that may be monitored for completion but that don't have a result.
- Those that return a result. For example, if an asynchronous function is fetching information from a web service, you will most likely want to have that function return that information in order to use it.

In both cases, you will still want to have access to the task and react if there is an exception or cancel them if their work is not required anymore. Let's take a look at how we can create and interact with both types.

Job

A job is a fire-and-forget task. An operation that, once started, you will not be waiting on, unless, maybe, there's an exception. The most common way to create a job is to use the `launch()` coroutine builder, as shown:

```
fun main(args: Array<String>) = runBlocking {
    val job = launch {
        // Do background task here
    }
}
```

However, you can also use the `Job()` factory function, like this:

```
fun main(args: Array<String>) = runBlocking {
    val job = Job()
}
```

`Job` is an interface, and both `launch()` and `Job()` return the `JobSupport` implementation, which is the basis of many implementations of `Job`. `Deferred`—which we will see as we move ahead—is an interface that extends `Job`.

Exception handling

By default, an exception happening inside a job will be propagated to where it was created. This happens even if we aren't waiting for the job to complete. Consider this example:

```
fun main(args: Array<String>) = runBlocking {
    launch {
        TODO("Not Implemented!")
    }

    delay(500)
}
```

Here, we use `delay()` to have the app run for enough time so that the exception is thrown. We aren't using `join()` on purpose, to show that the exception will be propagated regardless of us waiting for the job to complete.

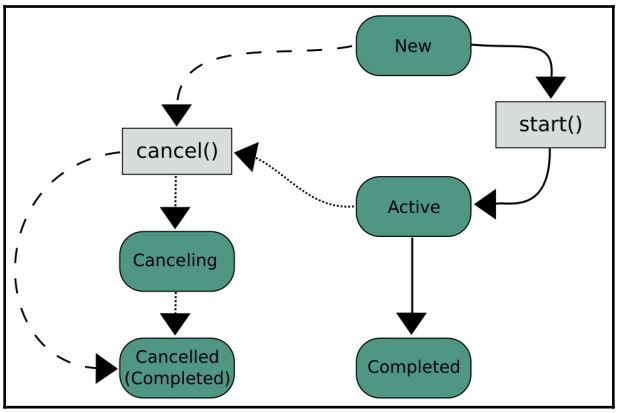
This will propagate the exception to the uncaught exception handler of the current thread. In a JVM application, this means that the exception will be printed on standard error output, as illustrated:

```
/usr/lib/jvm/java-8-openjdk/bin/java ...
Exception in thread "ForkJoinPool.commonPool-worker-1" kotlin.NotImplementedError: An operation is not implemented: Not Implemented!
    at chapter3.job.JobKt$main$1$1.doResume(job.kt:8)
    at kotlin.coroutines.experimental.jvm.internal.CoroutineImpl.resume(CoroutineImpl.kt:54)
    at kotlinx.coroutines.experimental.DispatchedTask$DefaultImpls.run(Dispatched.kt:161)
    at kotlinx.coroutines.experimental.DispatchedContinuation.run(Dispatched.kt:25)
    at java.util.concurrent.ForkJoinTask$RunnableExecuteAction.exec(ForkJoinTask.java:1402)
    at java.util.concurrent.ForkJoinTask.doExec(ForkJoinTask.java:289)
    at java.util.concurrent.ForkJoinPool$WorkQueue.runTask(ForkJoinPool.java:1056)
    at java.util.concurrent.ForkJoinPool.runWorker(ForkJoinPool.java:1692)
    at java.util.concurrent.ForkJoinWorkerThread.run(ForkJoinWorkerThread.java:157)
```

As we move ahead, we will see how to use `CoroutineExceptionHandler` and `invokeOnCompletion()` to handle exceptions.

Life cycle

Here's a diagram showing the life cycle of a job:



By default, a job is **started** as soon as it's created. This happens both when the job is created with `launch()` and when it's done with `Job()`. As we will see in a bit, it's possible to create a job without starting it.

There are five states in the diagram:

- **New**: A job that exists but is not executing yet.
- **Active**: A job that is running. A suspended job is also considered active.
- **Completed**: When the job is not executing any longer.
- **Cancelling**: When **cancel()** is called on a Job that is active, it may require time for the cancellation to complete. This is an intermediate state between **Active** and **Cancelled**.
- **Cancelled**: A job that has completed its execution due to cancellation. Note that a **Cancelled** job can be considered **Completed** too.

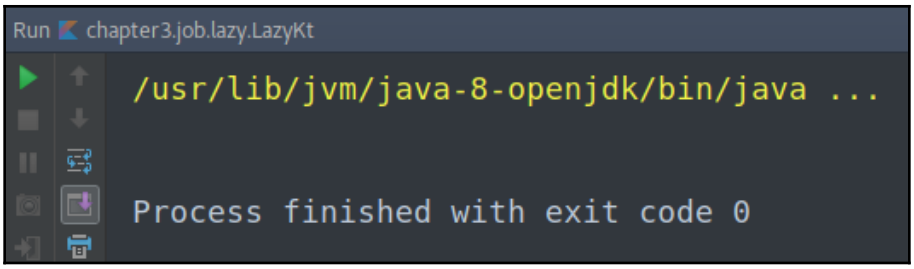
Note that if `cancel()` is called in a job in the **New** state, it will not go through the **Cancelling** state. It will directly go to **Cancelled**.

New

As mentioned earlier, a job is, by default, started automatically both when it's created using `launch()` and when it's created using `Job()`. In order to create a job without starting it, you have to use `CoroutineStart.LAZY` when creating it. Consider the given example:

```
fun main(args: Array<String>) = runBlocking {
    launch(start = CoroutineStart.LAZY) {
        TODO("Not implemented yet!")
    }

    delay(500)
}
```

When running this snippet of code, you will note that it didn't print any errors. Since the job is created but never started, the exception will not be thrown:



Active

A job in the state *new* can be started in many ways, but commonly it will be done by calling `start()` or `join()`, the difference being that the former will start the job without waiting for it to complete, whereas the latter will suspend execution until the job completes. Take this example into consideration:

```
fun main(args: Array<String>) {
    val job = launch(start = CoroutineStart.LAZY) {
        delay(3000)
    }

    job.start()
}
```

The mentioned code will not suspend execution when `job.start()` is invoked, so the application will complete its execution without waiting for `job` to complete.

Since `start()` doesn't suspend execution, it doesn't need to be called from a suspending function or coroutine. It can be called from any part of our application.

If we use `join()`,we will force the application to wait for `job` to complete, as demonstrated:

```
fun main(args: Array<String>) = runBlocking {
    val job = launch(start = CoroutineStart.LAZY) {
        delay(3000)
    }

    job.join()
}
```

As `join()` can suspend execution, it needs to be called from a coroutine or a suspending function. Note that `runBlocking()` is being used for that.

Any job that has been started is therefore active, and it will be active until it completes execution or until cancellation is requested.

Canceling

An active job that is requested to be cancelled may enter a staging state called *canceling*. To request a job to cancel its execution, the `cancel()` function should be called:

```
fun main(args: Array<String>) = runBlocking {
    val job = launch {
        // Do some work here
        delay(5000)
    }

    delay(2000)
    job.cancel()
}
```

Here, the execution of the job will be cancelled after 2 seconds; `cancel()` has an optional `cause` parameter. If an exception is the cause of the cancellation, it's a good practice to send it there; that way, it can be retrieved at a later time:

```
fun main(args: Array<String>) = runBlocking {
    val job = launch {
        // Do some work here
        delay(5000)
    }

    delay(2000)
    // cancel with a cause
    job.cancel(cause = Exception("Timeout!"))
}
```

There is also a `cancelAndJoin()` function. As its name suggests, this will not only cancel the execution but also suspend the current coroutine until the cancellation has been completed.

Currently, there is no implementation of `cancelAndJoin()` that allows a cause to be passed.

Cancelled

A job whose execution has ended due to cancellation or an unhandled exception is considered *cancelled*. When a job has been cancelled, the information about the cancellation can be obtained through the `getCancellationException()` function. This function will return a `CancellationException`, which can be used to retrieve information such as the cause of the cancellation, if it was set. Here's an example:

```
fun main(args: Array<String>) = runBlocking {
    val job = launch {
        delay(5000)
    }

    delay(2000)

    // cancel
    job.cancel(cause = Exception("Tired of waiting"))

    val cancellation = job.getCancellationException()
    cancellation.cause // Exception("Tired of waiting")
}
```

In order to differentiate between a job that was cancelled and one that failed due to exception, you are encouraged to set a `CoroutineExceptionHandler` to it and process the cancellation there, as shown:

```
fun main(args: Array<String>) = runBlocking {
    val exceptionHandler = CoroutineExceptionHandler {
        _: CoroutineContext, throwable: Throwable ->
        println("Job cancelled due to ${throwable.message}")
    }

    launch(exceptionHandler) {
        TODO("Not implemented yet!")
    }

    delay(2000)
}
```

You can also use `invokeOnCompletion()`, as follows:

```
fun main(args: Array<String>) = runBlocking<Unit> {
    launch {
        TODO("Not implemented yet!")
    }.invokeOnCompletion { cause ->
        cause?.let {
            println("Job cancelled due to ${it.message}")
        }
    }

    delay(2000)
}
```

Completed

Any job whose execution has ceased is considered *completed*. This applies regardless of whether the execution ended normally, was cancelled, or ended due to an exception. For this reason, *cancelled* can be considered a substate of *completed*.

Determining the current state of a Job

As we have already seen, a job has many different states, so it's important to know how to determine its current state as its external observers.

A job has three properties that can help with this; they're as listed:

- `isActive`: Whether the job is active. This will return `true` when the job is suspended too.
- `isCompleted`: Whether the job has completed its execution.
- `isCancelled`: Whether the job has been cancelled. This will hold *true* as soon as the cancellation is requested.

These properties can be easily mapped with the list of states given earlier:

State	isActive	isCompleted	isCancelled
Created	false	false	false
Active	true	false	false
Cancelling	false	false	true
Cancelled	false	true	true
Completed	false	true	false

The documentation of Job provides another state—an internal state called **completing**. This state is not covered as an individual state, given that it's internal and its signature is similar to that of *Active*.

Deferred

Deferred extends job in order to make it an asynchronous task that also has a result. Deferred is Kotlin's implementation of what other languages call *Futures* or *Promises*. The idea behind it is that a computation will return you an object that will be *empty* until the asynchronous task completes.

The life cycle of deferred and its states are similar to those of job, so the real differences are in return type—like we saw in the previous chapter—and error handling, as we will cover when we move ahead.

At the time of writing, deferred has a *failed* state, which doesn't exists in job. It is planned to remove it to simplify the life cycle of deferred. Because of this, it's advised that you don't base your usage of deferred around the *failed* state. Refer to `issue 220` in the coroutines' Github repository to learn more about the reasoning behind the change.

To create deferred, you can use `async`, as we did in the previous chapter:

```
fun main(args: Array<String>) = runBlocking {
    val headlinesTask = async {
        getHeadlines()
    }

    headlinesTask.await()
}
```

Alternatively, you can use the constructor of `CompletableDeferred`:

```
val articlesTask = CompletableDeferred<List<Article>>()
```

Deferred is a synonym of Future. The name was selected to avoid confusion with JVM's `java.util.concurrent.Future`.

Exception handling

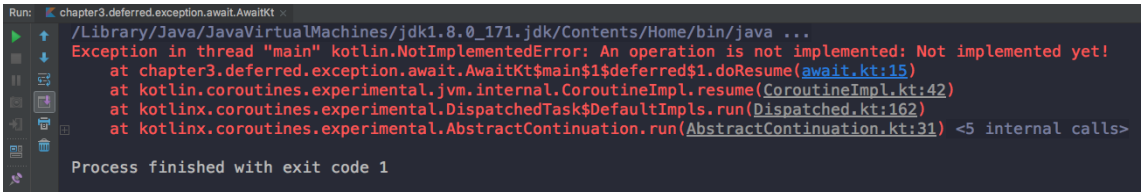
Unlike a pure job, deferred will not propagate unhandled exceptions automatically. It's done this way because it's expected that you will wait for the result of deferred, so it's up to you to validate that the execution was successful. Here's an example where deferred is not awaited:

```
fun main(args: Array<String>) = runBlocking {
    val deferred = async {
        TODO("Not implemented yet!")
    }

    // Wait for it to fail
    delay(2000)
}
```

The preceding example will have deferred fail, but it will not propagate the exception. Here, we are using `delay()` so that we can reproduce an scenario in which we are not monitoring the execution of deferred—which you should never do—because deferred is intended to be monitored. Let's see how we can get the exception to propagate easily:

```
fun main(args: Array<String>) = runBlocking<Unit> {
    val deferred = async {
        TODO("Not implemented yet!")
    }

    // Let it fail
    deferred.await()
}
```

This code, on the other hand, will have the exception propagate and will crash the application:



Deferred is designed this way because the idea is that by calling `await()`, you are indicating that the execution of deferred is an integral part of the code flow. Having it this way makes it easier to write asynchronous code that looks imperative, and it also allows for exception handling using a try-catch block:

```
fun main(args: Array<String>) = runBlocking<Unit> {
    val deferred = async {
        TODO("Not implemented yet!")
    }

    try {
        deferred.await()
    } catch (throwable: Throwable) {
        println("Deferred cancelled due to ${throwable.message}")
    }

}
```

You can also use the `CoroutineExceptionHandler` in the same way as when it was used for the job.

For the rest of this chapter, we will refer to both job and deferred as job since it's the base interface. Unless otherwise specified, everything that is said about job will apply to deferred as well.

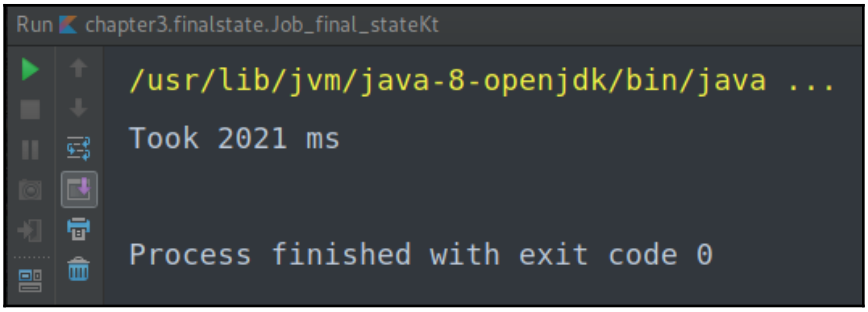
States move in one direction only

Once a state has been reached, a job will not move back to a previous state. Consider the following application:

```
fun main(args: Array<String>) = runBlocking {
    val time = measureTimeMillis {
        val job = launch {
            delay(2000)
        }
        // Wait for it to complete once
        job.join()

        // Restart the Job
        job.start()
        job.join()
    }
    println("Took $time ms")
}
```

This code creates a job that suspends execution for 2 seconds. Once the job completes on the first call to `job.join()`, the `start()` function is called on `job` with the intent of restarting it, followed by a second call to `join()` to wait for the second execution to complete. The complete time of execution was measured and stored in the `time` variable.

Let's see how long the execution of this code takes:



As you can see, the total execution took around 2 seconds, which proves that `job` was executed only once. If calling `start()` on a complete job were to restart it, the total execution would have been around 4 seconds.

This is consistent with our previous statement—once a job has reached a state, it will not go back. In this case, the job reached a *Completed* state, so calling `start()` has no effect on it.

Likewise, calling `join()` in a complete job will not do anything. Since the job has already completed, no suspension will happen.

A note on final states

Some of the states of a job are considered *final states*. Final states are those that a job will never be able to move from. This makes sense when we take into consideration that a job will not go back to a previous state. Those states are *Cancelled* and *Completed.*

RSS – Reading from multiple feeds concurrently

Now that we have a better understanding of the life cycle of a job, it's a good moment to go back to our Android RSS reader and improve it.

Supporting a list of feeds

Let's go back to our Android Studio project and create an immutable list to hold the feeds that we will fetch. For now, three feeds will be enough:

```
class MainActivity : AppCompatActivity() {

    val feeds = listOf(
            "https://www.npr.org/rss/rss.php?id=1001",
            "http://rss.cnn.com/rss/cnn_topstories.rss",
            "http://feeds.foxnews.com/foxnews/politics?format=xml"
    )
...
}
```

Now we have a list of feeds that contain the URLs of NPR, CNN, and Fox News, let's update the `fetchRssHeadlines()` function to have it adjust to our goal:

```
private fun asyncFetchHeadlines(feed: String,
        dispatcher: CoroutineDispatcher) = async(dispatcher) {
    val builder = factory.newDocumentBuilder()
    val xml = builder.parse(feed)
    val news = xml.getElementsByTagName("channel").item(0)

    (0 until news.childNodes.length)
            .map { news.childNodes.item(it) }
            .filter { Node.ELEMENT_NODE == it.nodeType }
            .map { it as Element }
            .filter { "item" == it.tagName }
            .map {
                it.getElementsByTagName("title").item(0).textContent
            }
}
```

Note that instead of having a single feed, now our function takes a `feed` argument that is the URL that it'll use to fetch the news. This will allow us to fetch headlines from more than one feed. Additionally, the signature of the function has changed so that it's now an asynchronous function that takes the dispatcher to be used as an argument.

The name of the function has been updated to include `async`, as good practice suggests. The name has also been shortened for better readability.

Creating a thread pool

The next step is to update the dispatcher. Let's make it a thread pool of size `2` and rename it to *IO*:

```
val dispatcher = newFixedThreadPoolContext(2, "IO")
```

Here, we are increasing the size of the pool because `asyncFetchHeadlines()` is not only fetching the information from the server but also parsing it. This overhead of parsing the XML will affect performance if we use a single thread; sometimes fetching information from one feed can be delayed until the parsing of another is completed.

Fetching the data concurrently

Now we have everything we need in order to request many feeds concurrently. The idea now is to create one deferred for each feed on the list. Let's first update the `asyncLoadNews()` function so that it has a list where we can keep track of every deferred that we will be waiting for:

```
private fun asyncLoadNews() = launch {
    val requests = mutableListOf<Deferred<List<String>>>()
    ...
}
```

Then, let's add an element to the list for each feed on the list:

```
val requests = mutableListOf<Deferred<List<String>>>()

feeds.mapTo(requests) {
    asyncFetchHeadlines(it, dispatcher)
}
```

Following that, let's add code to wait for each of them to complete:

```
requests.forEach {
    it.await()
}
```

Merging the responses

The current implementation of `asyncLoadNews()` will wait for each of the requests to end. But since each of them returns a list of headlines, we want to join all their results into a single list containing all of them. For this, we can flat map the content of each deferred:

```
val headlines = requests.flatMap {
    it.getCompleted()
}
```

So now we have a `headlines` variable that contains all the headlines fetched concurrently from the three feeds. At this point, `asyncLoadNews()` will have two sections. The first one is to fetch and organize the data:

```
private fun asyncLoadNews() = launch {
    val requests = mutableListOf<Deferred<List<String>>>()

    feeds.mapTo(requests) {
        asyncFetchHeadlines(it, dispatcher)
    }
    requests.forEach {
        it.await()
    }

    val headlines = requests.flatMap {
        it.getCompleted()
    }
    ...
}
```

The second half, just below the first one, displays the amount of headlines on the UI:

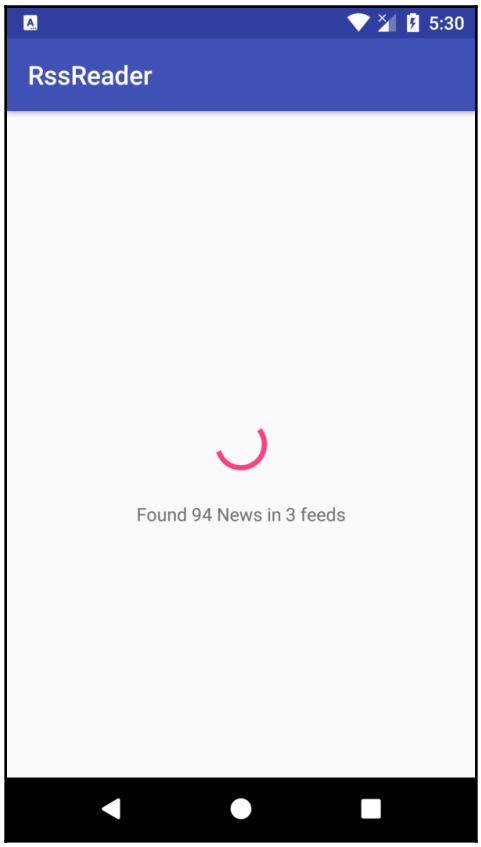
```
val newsCount = findViewById<TextView>(R.id.newsCount)

launch(UI) {
    newsCount.text = "Found ${headlines.size} News"
}
```

Let's update the second section so that the message that we print also displays the amount of feeds that were fetched:

```
val newsCount = findViewById<TextView>(R.id.newsCount)

launch(UI) {
    newsCount.text = "Found ${headlines.size} News " +
            "in ${requests.size} feeds"
}
```

Testing the concurrent requests

When running this new implementation, we will see that much more news is being obtained:



Non-happy path – Unexpected crash

Let's take a look at how we are waiting for each deferred to complete:

```
private fun asyncLoadNews() = launch {
    val requests = mutableListOf<Deferred<List<String>>>()

    feeds.mapTo(requests) {
        asyncFetchHeadlines(it, dispatcher)
    }

    requests.forEach {
        it.await()
    }

    ...
}
```

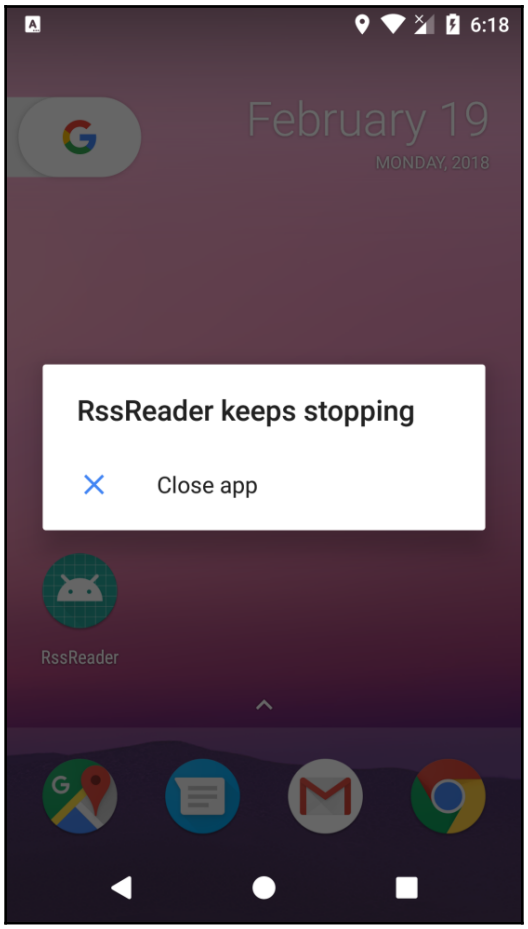
Since we are using `await()` to wait for the completion of our coroutines, any exception happening inside them will be propagated to the current thread. This means that there are two scenarios in which the application will crash easily:

- There's no internet connection
- The URL of one or more feeds is invalid or incorrect

Let's test one of those out. We can add an invalid URL to the list of feeds, which is something like this:

```
private val feeds = listOf(
        "https://www.npr.org/rss/rss.php?id=1001",
        "http://rss.cnn.com/rss/cnn_topstories.rss",
        "http://feeds.foxnews.com/foxnews/politics?format=xml",
        "htt:myNewsFeed"
)
```

If we run the application, it will crash as soon as it tries to fetch this feed:



Also, the log will be explicit about the exception. Since the protocol doesn't exist, the request fails:

```
02-19 18:23:54.356 E/AndroidRuntime: FATAL EXCEPTION: ForkJoinPool.commonPool-worker-1
                                     Process: co.starcarr.rssreader, PID: 15481
                                     java.net.MalformedURLException: unknown protocol: htt
                                         at java.net.URL.<init>(URL.java:596)
                                         at java.net.URL.<init>(URL.java:486)
                                         at java.net.URL.<init>(URL.java:435)
```

Having deferred store the exception

The easy way to handle the exception is by waiting on the deferred using `join()` instead of `await()`. This way, the exception will not be propagated when waiting. The updated code looks like this:

```
private fun asyncLoadNews() = launch {
    val requests = mutableListOf<Deferred<List<String>>>()

    feeds.mapTo(requests) {
        asyncFetchHeadlines(it, dispatcher)
    }

    requests.forEach {
        it.join()
    }
    ...
}
```

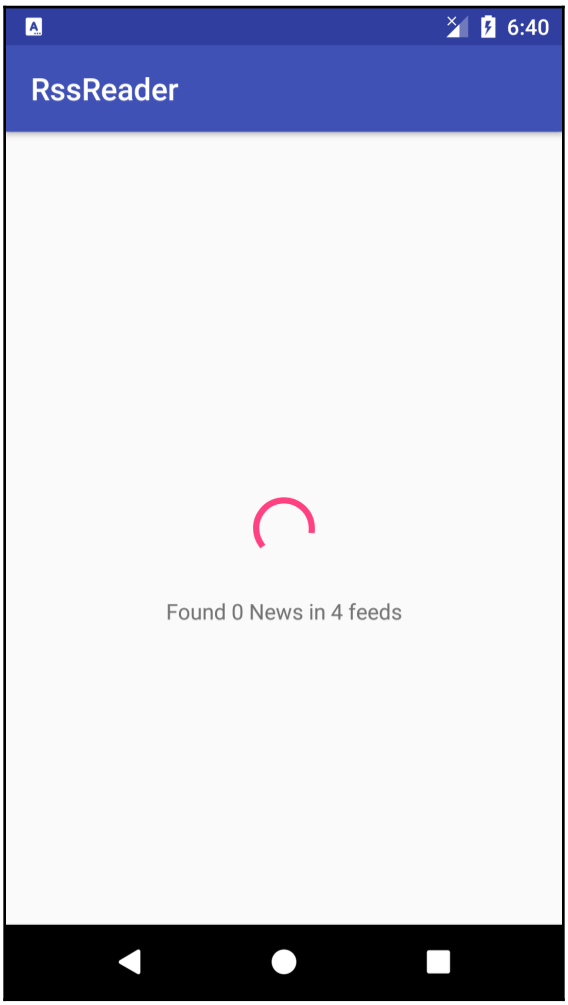
Now if you run the application, you will see that it still crashes. This is happening because even though we are not propagating the exception when waiting for deferred, we are propagating the exception when reading from it. Calling `getCompleted()` in a deferred that was cancelled—in this case because of an exception—will throw the exception.

So we need to also update the code so that `getCompleted()` is called only when deferred didn't fail:

```
val headlines = requests
        .filter { !it.isCancelled }
        .flatMap { it.getCompleted() }
```

At the time of writing, there is an effort from the Kotlin team to make the code above work as expected here. If you notice that the list of headlines is not being loaded correctly, please replace `isCancelled` with `isCompletedExceptionally`. I decided to avoid explaining `isCompletedExceptionally` because it's being deprecated close to the release of the book. For more information, please read `issue 220` in the coroutines' Github repository.

Now, running the application will not crash when an invalid feed is added to the list. It will also handle the device not having connectivity without crashing:



Don't ignore the exception!

We have reached a nice point, at which our application will not crash when obtaining or processing the news from the feed, but there is something wrong here; currently, we are just putting the exceptions aside and not doing anything about them. For example, if you look at the preceding screenshot, for many users it will seem like there no news to fetch, not necessarily that fetching the information was not possible due to the phone being offline. In the case where one of the feeds' URLs was invalid, we made it look like the feed was processed along with the other three, and its news is part of the result.

Ignoring exceptions is bad practice, and we will ensure that we fix that. Let's add a label to our app, in which we will display the amount of feeds that the app failed to fetch. First, let's go to the XML of the activity and add a second label following the current one:

```
<TextView
    android:id="@+id/warnings"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="20dp"
    app:layout_constraintTop_toBottomOf="@id/newsCount"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent" />
```

Now we need to go back to the activity and get a count of the feeds that completed exceptionally. Let's do that just after where we obtain the successful ones:

```
val headlines = requests
        .filter { !it.isCancelled }
        .flatMap { it.getCompleted() }

val failed = requests
        .filter { it.isCancelled }
        .size
```

At the time of writing, there is an effort from the Kotlin team to make the code above work as expected here. If you notice that the list of failed requests is not being loaded correctly, please replace `isCancelled` with `isCompletedExceptionally`. I decided to avoid explaining `isCompletedExceptionally` because it's being deprecated close to the release of the book. For more information, please read `issue 220` in the coroutines' Github repository.

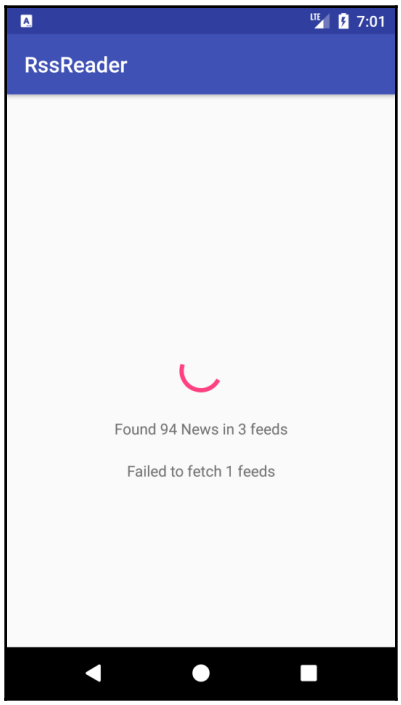
We also need to obtain the *View* in order to be able to set the text. Let's do that below where we are currently obtaining the view with the summary. Also, let's add a convenience variable to hold the number of successful requests:

```
val newsCount = findViewById<TextView>(R.id.newsCount)
val warnings = findViewById<TextView>(R.id.warnings)
val obtained = requests.size - failed
```

Finally, we can now update the UI block to display more accurate information:

```
launch(UI) {
    newsCount.text = "Found ${headlines.size} News " +
            "in $obtained feeds"

    if (failed > 0) {
        warnings.text = "Failed to fetch $failed feeds"
    }
}
```

Now we can display more accurate information to the user:



Summary

This chapter contained information that is vital when working with coroutines. Knowing about the different states of a job and how to calculate the current one is important in order to monitor them. Let's recap the topics that were covered in this chapter:

- `Job` is used for background tasks that don't return anything.
- `Deferred` is used when the background operation returns something that we want to receive.
- A job can have many states: *New*, *Active*, *Canceling*, *Cancelled*, and *Completed.*
- The `isActive`, `isCancelled`, and `isCompleted` properties can be used to determine the current state of a job.
- Deferred extends job to add the possibility to return something.
- The possible states of deferred are the same as that of a job.
- A job's state can only move forward; it can't be returned to a previous state.
- A *final state* is one of the states from which a job can't move.
- The *final states* of job are *Cancelled* and *Completed.*
- If deferred is waited on using `join()`, it's mandatory to validate whether it was cancelled before trying to read its value, to prevent exceptions from being propagated.
- Always log or display the exceptions in your jobs.

In the next chapter, we will update our application's UI to display the news in an scrollable list. From there, we will move on to an important topic—suspending functions. We will study them in detail, along with their use cases, and, of course, we will integrate them into the RSS reader.

4
Suspending Functions and the Coroutine Context

So far, we have limited ourselves to writing suspending code using coroutine builders such as `launch` and `async`, but Kotlin offers more ways to write it. In this chapter, we will start by updating our RSS reader to actually display the articles that it retrieved, then we will learn about suspending functions and compare them with the async functions that we have been using so far. Also, we will cover the coroutine context and its use in detail.

Here is a summary of topics that will be covered in this chapter:

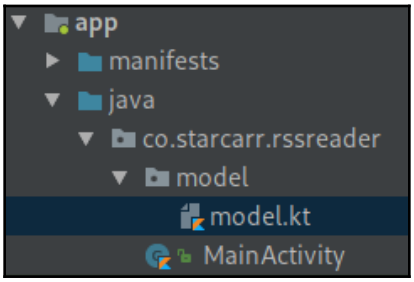
- What a suspending function is
- How to use suspending functions
- When to use async functions instead of suspending functions
- What the coroutine context is
- Different types of contexts such as dispatcher, exception handlers, and non-cancellables
- Combining and separating contexts to define the behavior of coroutines

Improving the UI of the RSS Reader

This is a good moment to make our RSS Reader more user friendly. Let's update the application so that it displays the information of the news articles as a list that the user can scroll, and let's display not only the headline of the article but also a summary of it and the feed that it belongs to.

Giving each feed a name

First, let's start by creating a new package, `model`, so that we can organize our data classes, and let's create a Kotlin `model.kt` inside it, as shown:



We want to be able to identify a feed not only by its URL but also by a name, so let's now create a data class called `Feed` inside our new model file. This class will hold pairs of a `name` and a `url` for each feed:

```
package co.starcarr.rssreader.model

data class Feed(
    val name: String,
    val url: String
)
```

Now we can update `feeds` in `MainActivity` so that its contents are of type `Feed`. Notice that the last element on the list is an invalid feed.

```
private val feeds = listOf(
        Feed("npr", "https://www.npr.org/rss/rss.php?id=1001"),
        Feed("cnn", "http://rss.cnn.com/rss/cnn_topstories.rss"),
        Feed("fox", "http://feeds.foxnews.com/foxnews/latest?format=xml"),
        Feed("inv", "htt:myNewsFeed")
)
```

It will be convenient to update the function `asyncFetchHeadlines()` so that it takes a `Feed` and uses its URL property to fetch the feed:

```
private fun asyncFetchHeadlines(feed: Feed,
        dispatcher: CoroutineDispatcher) = async(dispatcher) {
    val builder = factory.newDocumentBuilder()
    val xml = builder.parse(feed.url)
    ...
}
```

Fetching more information about the articles from the feed

As mentioned previously, we want to display a convenient set of information about each of the articles: feed, title, and summary. So let's create a data class that will hold that information. We can create it right below `Feed`:

```
package co.starcarr.rssreader.model

data class Feed(
    val name: String,
    val url: String
)
data class Article(
    val feed: String,
    val title: String,
    val summary: String
)
```

Instead of having the function `asyncFetchHeadlines()`, which returns only the title, we want the function to now be named `asyncFetchArticles()` and return a `Deferred<List<Article>>` corresponding to the articles of that feed:

```
private fun asyncFetchArticles(feed: Feed,
        dispatcher: CoroutineDispatcher) = async(dispatcher) {
    ...
}
```

In order to return a deferred list of articles, we need to change the mapping so that each item is converted to an `Article`:

```
.map {
    val title = it.getElementsByTagName("title")
            .item(0)
            .textContent
    val summary = it.getElementsByTagName("description")
            .item(0)
            .textContent
    Article(feed.name, title, summary)
}
```

Now our list of requests in `asyncLoadNews()` needs to be updated accordingly:

```
private fun asyncLoadNews() = launch {
    val requests = mutableListOf<Deferred<List<Article>>>()
    ...
}
```

The list `headlines` should now be renamed to `articles` to be consistent with its content:

```
val articles = requests
        .filter { !it.isCancelled }
        .flatMap { it.getCompleted() }
```

At the time of writing, there is an effort from the Kotlin team to make the code above work as expected here. If you notice that the list of articles is not being loaded correctly, please replace `isCancelled` with `isCompletedExceptionally`. I decided to avoid explaining `isCompletedExceptionally` because it's being deprecated close to the release of the book. For more information, please read `issue 220` in the coroutines' Github repository.

Adding a scrollable list for the articles

Currently, the best way to display a list of scrollable items in an Android application is to use a `RecyclerView`, so we will add one to our activity. Let's update the `build.gradle` file of the module, adding the dependency:

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    ...

    // Android support
    implementation "com.android.support:recyclerview-v7:26.1.0"
    ...
}
```

Once the project has synchronized the changes, you will be able to add a `RecyclerView` to the layout. Let's go to `activity_main.xml` and remove the layout of both of the text views. The `ConstraintLayout` inside the XML should now only contain the `ProgressBar`:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout .../>
```

```
        <ProgressBar
            android:id="@+id/progressBar"
            style="?android:attr/progressBarStyle"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            app:layout_constraintLeft_toLeftOf="parent"
            app:layout_constraintRight_toRightOf="parent" />

    </android.support.constraint.ConstraintLayout>
```

Let's add the `RecyclerView` inside the `ConstraintLayout` above the `ProgressBar`. Set its ID to `articles`:

```
    <?xml version="1.0" encoding="utf-8"?>
    <android.support.constraint.ConstraintLayout .../>

        <android.support.v7.widget.RecyclerView
            android:id="@+id/articles"
            android:scrollbars="vertical"
            android:layout_height="wrap_content"
            android:layout_width="match_parent"
            app:layout_constraintTop_toTopOf="parent" />
        <ProgressBar .../>

    </android.support.constraint.ConstraintLayout>
```

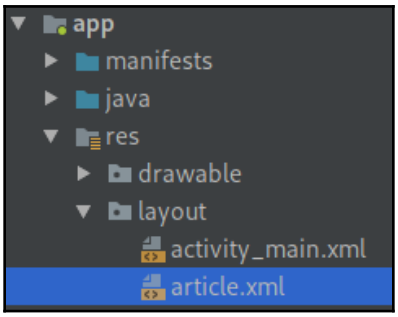
Because the views that we were using to display the amount of news were removed, now the application will not compile. Let's update the second half of `asyncLoadNews()` in `MainActivity` so that we don't use them anymore, and leave a TODO so that we update it later:

```
    private fun asyncLoadNews() = launch {
        ...
        val articles = requests
                .filter { !it.isCompletedExceptionally }
                .flatMap { it.getCompleted() }

        val failedCount = requests
                .filter { it.isCompletedExceptionally }
                .size

        val obtained = requests.size - failedCount

        launch(UI) {
            // TODO: Refresh UI here
        }
    }
```

We are keeping the variables `articles`, `failed`, and `obtained` so that we can use their information later.

Layout for the individual articles

We need a layout to display the information of the articles, so let's create an XML file for that in the `layout` folder:



Let's make it a `LinearLayout` with vertical orientation and some padding at the start and end:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:paddingStart="14dp"
    android:paddingEnd="14dp"
    android:layout_height="wrap_content">

</LinearLayout>
```

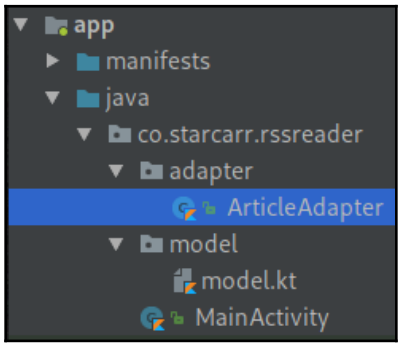
Inside it, we will add three text views; each of them will display some of the article's information:

```
<TextView
    android:id="@+id/title"
    android:textSize="20sp"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />

<TextView
    android:id="@+id/feed"
    android:textSize="12sp"
```

```
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

    <TextView
        android:id="@+id/summary"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
```

Adapter to map the information

In order to map an `Article` into an element of a `RecyclerView`, we will need to create an **Adapter**. Let's create the package `co.starcarr.rssreader.adapter` and add the class `ArticleAdapter` to it:



To start, let's simply add a private list of articles to the adapter. We will add articles to it as we fetch them:

```
package co.starcarr.rssreader.adapter

import co.starcarr.rssreader.model.Article

class ArticleAdapter {
    val articles: MutableList<Article> = mutableListOf()
}
```

Adding a ViewHolder

The idea behind `RecyclerView` is that the expensive process of creating views will be avoided as much as possible. Instead, a small set of views is created and they are reused—no space either side—check throughout hence the name `RecyclerView`—to display information as the user scrolls.

For this to work, we are expected to have a `ViewHolder`, which is an object that holds the layout elements of the view so that they can be reused later – recycling them. Let's create a `ViewHolder` inside the adapter, based on the layout that we defined for each item:

```
class ArticleAdapter {

    private val articles: MutableList<Article> = mutableListOf()

    class ViewHolder(
            val layout: LinearLayout,
            val feed: TextView,
            val title: TextView,
            val summary: TextView
    ) : RecyclerView.ViewHolder(layout)
}
```

Notice that our `ViewHolder` class extends `RecyclerView.ViewHolder` and is passing the layout itself to the super constructor.

Mapping the data

For our adapter to work, it needs to know how to create a `ViewHolder` and how to replace its contents, and it needs to be able to return the amount of elements that it currently has. The first step is to extend `RecyclerView.Adapter` using our own `ViewHolder` as the type:

```
class ArticleAdapter : RecyclerView.Adapter<ArticleAdapter.ViewHolder>() {
    ...
}
```

Once this is done, we will be requested to implement three functions:

- `onCreateViewHolder()`: This is called each time a new `ViewHolder` needs to be created; it will inflate all the views as needed and return a `ViewHolder` ready to be used.
- `onBindViewHolder()`: This is called to load/replace the content of a `ViewHolder` with the content of the element at a given position. It needs to update the content of the views accordingly.
- `getItemCount()`: This has to return the amount of elements that the adapter has.

onCreateViewHolder

This implementation is quite straightforward. We will inflate the XML layout that we defined for the individual articles, and find the views that will be used to display the information:

```
override fun onCreateViewHolder(parent: ViewGroup,
                                viewType: Int): ViewHolder {
    val layout = LayoutInflater.from(parent.context)
            .inflate(R.layout.article, parent, false) as LinearLayout

    val feed = layout.findViewById<TextView>(R.id.feed)
    val title = layout.findViewById<TextView>(R.id.title)
    val summary = layout.findViewById<TextView>(R.id.summary)

    return ViewHolder(layout, feed, title, summary)
}
```

Now each time a new `ViewHolder` is needed, this function will be called.

onBindViewHolder

Here we need to retrieve the article according to the position that we received, and load the text of the views accordingly:

```
override fun onBindViewHolder(holder: ViewHolder, position: Int) {
    val article = articles[position]

    holder.feed.text = article.feed
    holder.title.text = article.title
    holder.summary.text = article.summary
}
```

This function will be called initially to display the first group of articles, and later will be called when the user scrolls to replace the content of the articles.

getItemCount

Here we need to return the size of `articles`:

```
override fun getItemCount() = articles.size
```

Allowing the incremental addition of articles to the adapter

Currently, our adapter doesn't expose any function that will allow external clients to add articles to it. Let's add a simple function to allow groups of articles to be added:

```
fun add(articles: List<Article>) {
    this.articles.addAll(articles)
    notifyDataSetChanged()
}
```

`notifyDataSetChanged()` will let the adapter know that the contents have changed. This may cause items to be redrawn.

Connecting the adapter to the activity

Now we have an adapter that can map a `List<Article>` into views. We just need to use this adapter in conjunction with the `RecyclerView` that we added to the main activity.

For this, we will need some variables in our `MainActivity` class:

```
class MainActivity : AppCompatActivity() {

    ...
    private lateinit var articles: RecyclerView
    private lateinit var viewAdapter: ArticleAdapter
    private lateinit var viewManager: RecyclerView.LayoutManager
    ...
}
```

We will instantiate all of them inside the `onCreate()` function:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    viewManager = LinearLayoutManager(this)
    viewAdapter = ArticleAdapter()
    articles = findViewById<RecyclerView>(R.id.articles).apply {
        layoutManager = viewManager
        adapter = viewAdapter
    }

    asyncLoadNews()
}
```

To be able to actually display the articles in the UI, we now need to update `asyncLoadNews()` so that it adds the elements retrieved to the adapter. We will replace the TODO that we left there previously:

```
launch(UI) {
    // TODO: Refresh UI here
}
```

We now want to add code to hide the `ProgressBar`, and then add the new articles to `viewAdapter`:

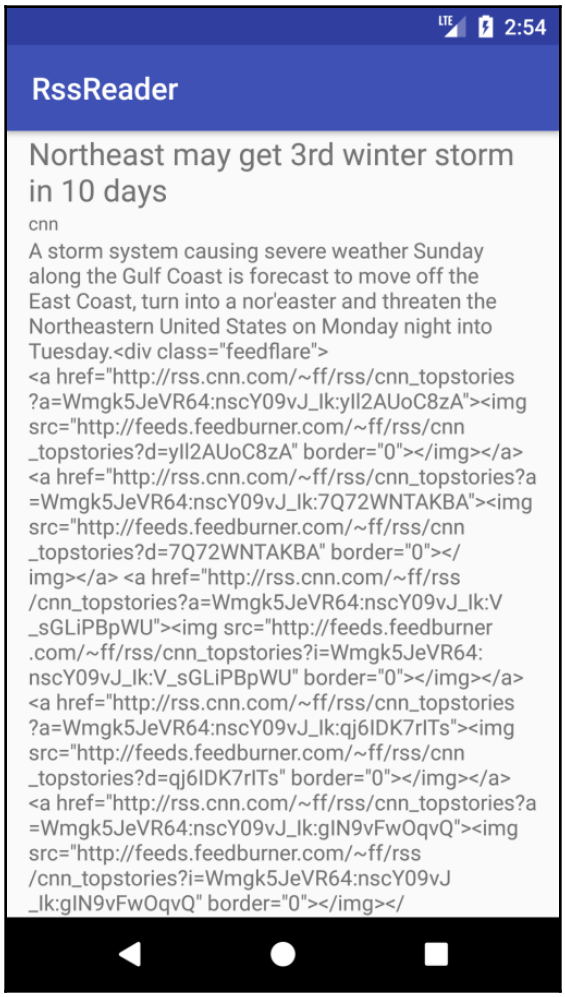
```
launch(UI) {
    findViewById<ProgressBar>(R.id.progressBar).visibility = View.GONE
    viewAdapter.add(articles)
}
```

Testing the new UI

Before we proceed to testing the UI, let's add some delay to `asyncFetchArticles()` so that we can actually see the `ProgressBar` for some time before we display the data:

```
private fun asyncFetchArticles(feed: Feed,
        dispatcher: CoroutineDispatcher) = async(dispatcher) {
    delay(1000)
    val builder = factory.newDocumentBuilder()
    ...
}
```

Now, when running the application, you will find that the news is actually being displayed along with the scrollbars on the side, indicating the current scrolling position. But if you scroll enough you will see some issues with some content:



At the time of writing, *Instant Run* doesn't work exceptionally well with applications that use coroutines. Please consider disabling it if you run into issues or exceptions.

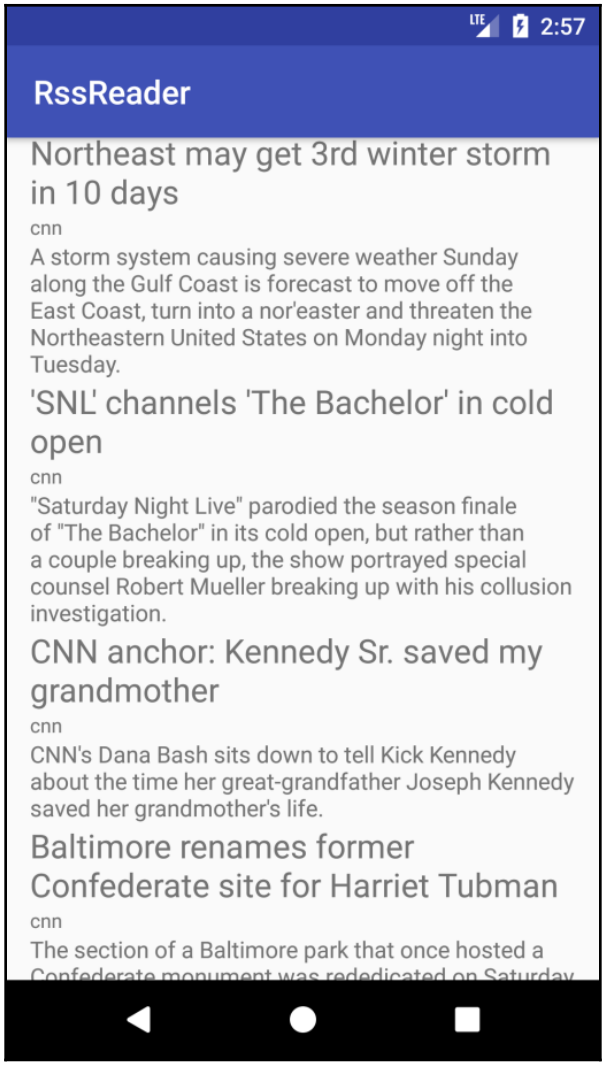
Sanitizing the data

Some of the articles display HTML tags in the summary. Usually these tags are fine because the summary is displayed in a viewer that supports HTML, but in our case we don't, so it's better to get rid of them.

To do this, we just need to tweak `asyncFetchArticles()` to cut the description if a `div` element is found:

```
.map {
    val title = it.getElementsByTagName("title")
            .item(0)
            .textContent
    var summary = it.getElementsByTagName("description")
            .item(0)
            .textContent

    if (!summary.startsWith("<div")
            && summary.contains("<div")) {
        summary = summary.substring(0, summary.indexOf("<div"))
    }


    Article(feed.name, title, summary)
}
```

Now all the articles look okay – notice that we cut the `div` only if the summary is not starting with one, to avoid having empty summaries. See updated results below:



Suspending functions

Throughout the book, we have written many suspending algorithms, but most of the time it has been done using coroutine builders such as `launch()`, `async()`, and `runBlocking()`. When calling a coroutine builder, the code that we pass to it is a **suspending lambda**. Now it's time to take a detailed look at how to write suspending code that exists as a function instead:

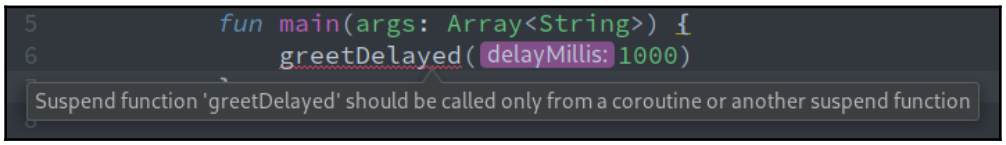
```
suspend fun greetDelayed(delayMillis: Int) {
    delay(delayMillis)
    println("Hello, World!")
}
```

To create a suspending function, you only need to add the modifier `suspend` to its signature. Notice that suspending functions can call other suspending functions such as `delay()` directly – no need to wrap inside a coroutine builder– making our code clean and easy to read.

Nevertheless, if we try calling this function outside of a coroutine it will not work. Consider this example:

```
fun main(args: Array<String>) {
    greetDelayed(1000)
}
```

This code will not compile because, as we already know, suspending computations can be called only from other suspending computations:



So in order to call this function from non-suspending code, we need to wrap it using a coroutine builder. For example:

```
fun main(args: Array<String>) {
    runBlocking {
        greetDelayed(1000)
    }
}
```

Now we have a function that suspends as part of its execution, and that function can be invoked using whichever dispatcher is more convenient at the time – for example, we could call it using a single-thread context.

Suspending functions in action

In `Chapter 2`, *Coroutines in Action*, we talked about whether it's more convenient to use async functions instead of using coroutine builders to implement concurrent code. Now it's time to expand this topic by adding suspending functions to the toolset. First, let's compare a simple implementation of a repository using an async function with the equivalent implementation using suspending functions.

As a reminder, we refer to async functions as functions that return an implementation of `Job`, including `Deferred`. Notice that those functions are commonly a function wrapped into a `launch()` or `async()` builder, but as long as an implementation of `Job` is returned, we will refer to them as async functions.

Writing a repository with async functions

Having functions that return an implementation of `Job` can be convenient in some scenarios, but it has the disadvantage of requiring the code to use `join()` or `await()` in order to suspend while a coroutine is being executed. What if we want to suspend as the default behavior? Let's design a repository using async functions and see what an implementation would look like. Let's start with the following data class:

```
data class Profile(
    val id: Long,
    val name: String,
    val age: Int
)
```

Now let's design an interface of a client that retrieves profiles based on `name` or `id`. The initial design would look like this:

```
interface ProfileServiceRepository {
    fun fetchByName(name: String) : Profile
    fun fetchById(id: Long) : Profile
}
```

But we want the implementation to have async functions, so we switch to returning a `Deferred` of `Profile` and rename the functions accordingly. Something like the following would work:

```
interface ProfileServiceRepository {
    fun asyncFetchByName(name: String) : Deferred<Profile>
    fun asyncFetchById(id: Long) : Deferred<Profile>
}
```

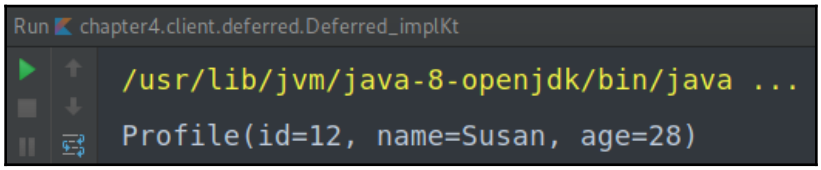
A mock implementation would be straightforward:

```
class ProfileServiceClient : ProfileServiceRepository {
    override fun asyncFetchByName(name: String) = async {
        Profile(1, name, 28)
    }

    override fun asyncFetchById(id: Long) = async {
        Profile(id, "Susan", 28)
    }
}
```

This implementation can then be invoked from any other suspending computation. See the following example:

```
fun main(args: Array<String>) = runBlocking {
    val client : ProfileServiceRepository = ProfileServiceClient()

    val profile = client.asyncFetchById(12).await()
    println(profile)
}
```

The output is what we expect:



There are some things that we can observe from this implementation:

- The name of the functions is conveniently verbose: it's important that we make it explicit that the function is async to make the client aware that they should wait until completion to continue if they need.

- Because of the nature of this client, it's probable that the caller will always have to suspend until the request is completed, so the call to `await()` will commonly be there right after the call to the function.
- The implementation will be tied to `Deferred`. There is no clean way to implement the interface `ProfileServiceRepository` with a different type of future. It would be messy to try to write an implementation that uses concurrency primitives that aren't from Kotlin. For example, imagine yourself writing an implementation with RxJava or Java's `Future`.

Upgrading to suspending functions

Let's now refactor the code to use suspending functions. We start with the same data class:

```
data class Profile(
    val id: Long,
    val name: String,
    val age: Int
)
```

But instead of the verbose names, we can have cleaner ones. And even more important, instead of the interface forcing the implementation to be an async function, we only care about it suspending and returning a `Profile` – we can now remove `Deferred` from the equation:

```
interface ProfileServiceRepository {
    suspend fun fetchByName(name: String) : Profile
    suspend fun fetchById(id: Long) : Profile
}
```

The implementation can be easily converted, too. Right now it doesn't need to actually suspend; it's a mock after all:

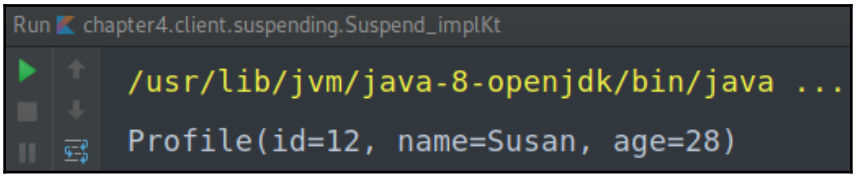
```
class ProfileServiceClient : ProfileServiceRepository {
    override suspend fun fetchByName(name: String) : Profile {
        return Profile(1, name, 28)
    }

    override suspend fun fetchById(id: Long) : Profile {
        return Profile(id, "Susan", 28)
    }

}
```

A real implementation could use *rxJava*, *retrofit*, or any other library to do the actual request, without having to tie the interface `ProfileServiceRepository` to their implementation of futures.

Now the caller's code is a little bit cleaner:

```
fun main(args: Array<String>) = runBlocking {
    val repository: ProfileServiceRepository = ProfileServiceClient()
    val profile = repository.fetchById(12)
    println(profile)
}
```

The output is the same as before:



This approach has some clear advantages over the async implementation:

- **Flexible**: The details of the implementation are not leaked, so the implementation can be made using any library that supports futures. As long as the implementation doesn't block the current thread, and returns the expected `Profile`, any type of future will work.
- **Simpler**: Using an async function for a task that we want to be sequential adds the verbosity of having to call `await()` all the time, and forces us to name the functions with explicitly async names. Using a suspending function removes both the need to change the name and the need to call `await()` each time the repository is used.

Suspending functions versus async functions

Some guidelines for the usage of suspending functions instead of async functions are as follows:

The list below uses `Job` to refer to implementations of it, including `Deferred<T>`.

- Generally, it is advised to use suspending functions over async functions to avoid tying the implementation to a `Job`.
- Always use suspending functions when defining an interface. Using async functions will force the implementation to return a `Job`.
- Likewise, always use a suspending function when defining an abstract function.
- The more visible a function is, the more important it is to use suspending functions: as already explained, you don't want to be forced to deprecate your public API because the implementation will be changed when you can't—or don't want to— return a `Job` anymore.
- The async functions should be limited to **private** and *maybe* **internal** functions. In some scenarios, like the one studied in `Chapter 2`, *Coroutines in Action*, having an async function can simplify the code. And having all the code using an async function in a small scope reduces the impact of a refactoring if returning a `Job` is no longer an option.

The coroutine context

The execution of a coroutine always happens inside a context. This context is a group of elements that will allow us to define how the coroutine will be executed and how it should behave. Let's start by talking about some of the contexts that we have already seen.

Each of the items in the context can be considered a context with a single element, that is, a context with a single behavior defined. As explained, a context can actually contain more than one element, so in the next section we will learn how to add and remove elements from a context to create combined behaviors. But for now, we want to talk about them as individual contexts to better explain how they work by themselves.

Dispatcher

Dispatchers determine the thread in which a coroutine will be executed, this includes both where it will be started and where it will be resumed after suspension.

We have talked about dispatchers since the very first chapter. So let's look at a small recap with examples of them.

CommonPool

`CommonPool` is a thread pool that is automatically created by the framework for CPU-bound operations. Its maximum size is the amount of cores of the machine minus one. Currently, it's used as the default dispatcher, but if you want to make its usage explicit, you can use it like any other dispatcher:

```
launch(CommonPool) {
    // TODO: Implement CPU-bound algorithm here
}
```

It's unclear whether `CommonPool` will stop being the default dispatcher. Consider using it explicitly for your CPU-bound operations to prevent unexpected changes.

Default dispatcher

Currently, it's the same as `CommonPool`. To use the default dispatcher you can use a builder without passing a dispatcher:

```
launch {
    // TODO: Implement suspending lambda here
}
```

Or you can make it explicit/verbose:

```
launch(DefaultDispatcher) {
    // TODO: Implement suspending lambda here
}
```

Unconfined

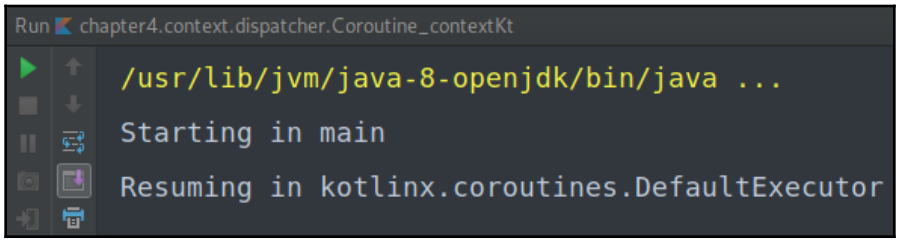
This dispatcher will execute the coroutine in the current thread until the first suspension point is reached. After suspension, the coroutine will be resumed in whichever thread is used by the suspending computation invoked by it when suspending.

For example, if your unconfined function `suspend a()` calls `suspend b()`, which is executed with a dispatcher in a specific thread, `a()` will be resumed in the same thread that `b()` was executed. This happens because of the way in which suspending computations are compiled, and we will explain that in `chapter 9`, *The Internals of Concurrency in Kotlin*.

Here is an example of its usage:

```
fun main(args: Array<String>) = runBlocking {
    launch(Unconfined) {
        println("Starting in ${Thread.currentThread().name}")
        delay(500)
        println("Resuming in ${Thread.currentThread().name}")
    }.join()
}
```

This will print an output similar to this. Notice that initially it was running in `main()`, but then it moved to `DefaultExecutor`:



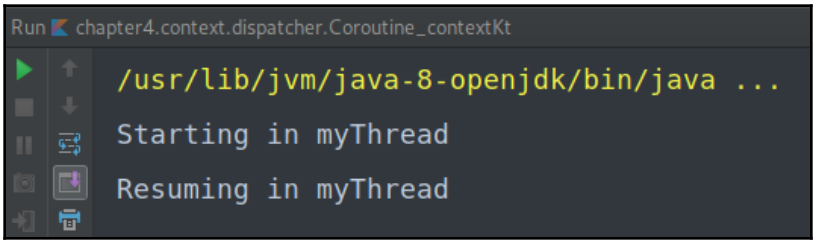
Single thread context

This dispatcher guarantees that, at all times, the coroutine is executed in a specific thread. To create a dispatcher of this type you have to use `newSingleThreadContext()`:

```
fun main(args: Array<String>) = runBlocking {
    val dispatcher = newSingleThreadContext("myThread")

    launch(dispatcher) {
        println("Starting in ${Thread.currentThread().name}")
```

```
            delay(500)
            println("Resuming in ${Thread.currentThread().name}")
        }.join()
    }
```

This will always execute in that same thread, even after any suspensions:

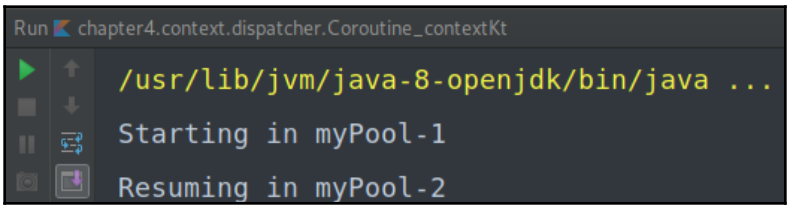


Thread pool

This dispatcher has a pool of threads and will start and resume coroutines in any of the threads that are available from that pool. The runtime takes care of determining which thread is available and it also decides how to distribute the load, so there's no work from our side other:

```
fun main(args: Array<String>) = runBlocking {
    val dispatcher = newFixedThreadPoolContext(4, "myPool")

    launch(dispatcher) {
        println("Starting in ${Thread.currentThread().name}")
        delay(500)
        println("Resuming in ${Thread.currentThread().name}")
    }.join()
}
```

As seen in the preceding code, a thread pool can be created by using `newFixedThreadPoolContext()`. The output of the previous code will commonly look as follows:



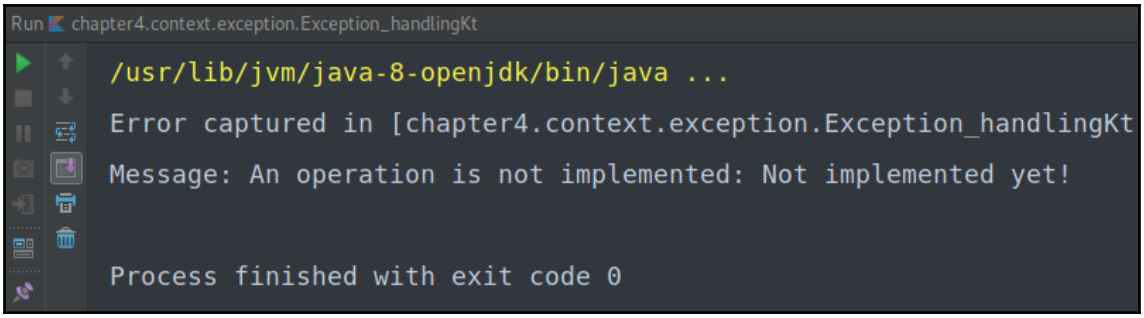
Exception handling

Another important use of the coroutine context is to define behavior for uncaught exceptions. This type of context can be created by implementing `CoroutineExceptionHandler`, as follows:

```
fun main(args: Array<String>) = runBlocking {
    val handler = CoroutineExceptionHandler({ context, throwable ->
        println("Error captured in $context")
        println("Message: ${throwable.message}")
    })

    launch(handler) {
        TODO("Not implemented yet!")
    }

    // wait for the error to happen
    delay(500)
}
```

In this example, we are creating a `CoroutineExceptionHandler` that prints the information of an uncaught exception. Then we start a coroutine that will throw an exception and give the application some time to print the message.

The application will then *handle* the exception gracefully:

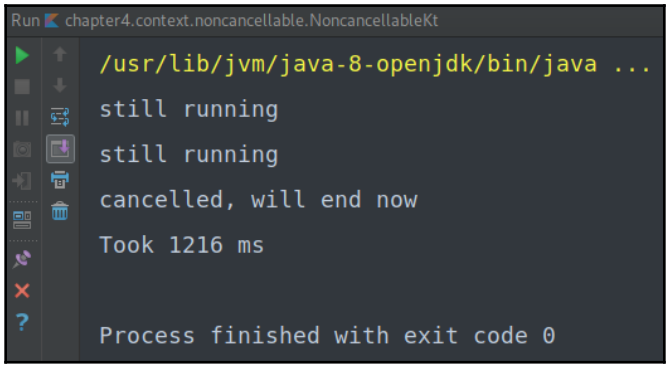


Non-cancellable

As covered in the previous chapter, when the execution of a coroutine is cancelled, an exception of the type `CancellationException` will be thrown inside the coroutine, causing the coroutine to end. Since an exception is thrown inside the coroutine, we can use a **try-finally** block to do some cleaning operations like closing resources – or for logging as well. This will look something like the following:

```
fun main(args: Array<String>) = runBlocking {
    val duration = measureTimeMillis {
        val job = launch {
            try {
                while (isActive) {
                    delay(500)
                    println("still running")
                }
            } finally {
                println("cancelled, will end now")
            }
        }

        delay(1200)
        job.cancelAndJoin()
    }

    println("Took $duration ms")
}
```

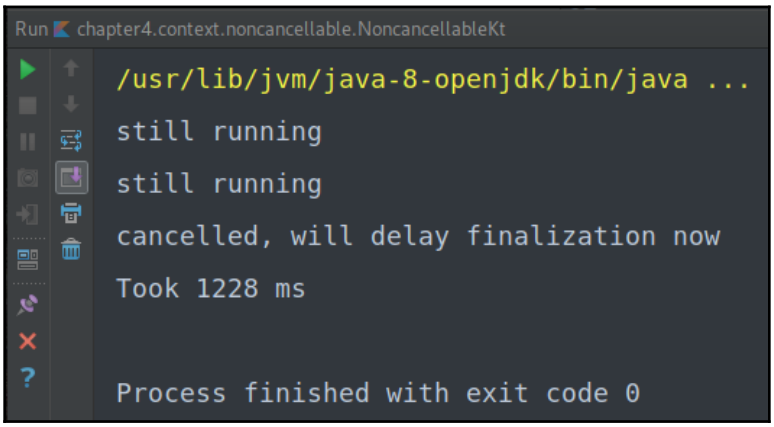
This code will print `"still running"` every 500 milliseconds until the coroutine is cancelled; on cancellation the finally block will be executed. The main thread will be suspended for 1.2 seconds, then it will cancel the job and print the total execution time before ending the application:



As expected, the job was cancelled soon after the 1.2 seconds delay ended, and the `finally` printed the message. Now let's modify the `finally` so that it stalls for five seconds before actually letting the coroutine end:

```
} finally {
    println("cancelled, will delay finalization now")
    delay(5000)
    println("delay completed, bye bye")
}
```

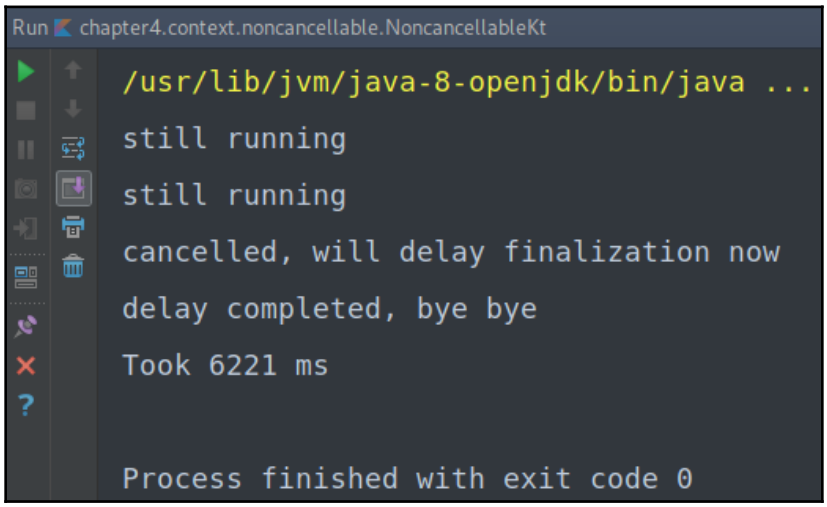
Now, upon execution, you'll notice that it doesn't work that way:



The delay in the `finally` actually didn't happen. It seems like the coroutine was ended as soon as it tried to suspend. This is because, **by design, a cancelling coroutine is not able to suspend**. If it's required for a coroutine to suspend during cancellation, it's necessary to use the `NonCancellable` context. Let's update the `finally` block once more:

```
} finally {
    withContext(NonCancellable) {
        println("cancelled, will delay finalization now")
        delay(5000)
        println("delay completed, bye bye")
    }
}
```

This implementation will guarantee that regardless of whether the coroutine is cancelling or not, the suspending lambda passed to `withContext()` will be able to suspend:



We will talk about `withContext()` in the next section of this chapter.

More about contexts

As we have seen, the context can define many different details of how a coroutine will behave. Contexts can also be operated on to define mixed behaviors. Let's talk about how contexts can be used in more creative ways.

Mixing contexts

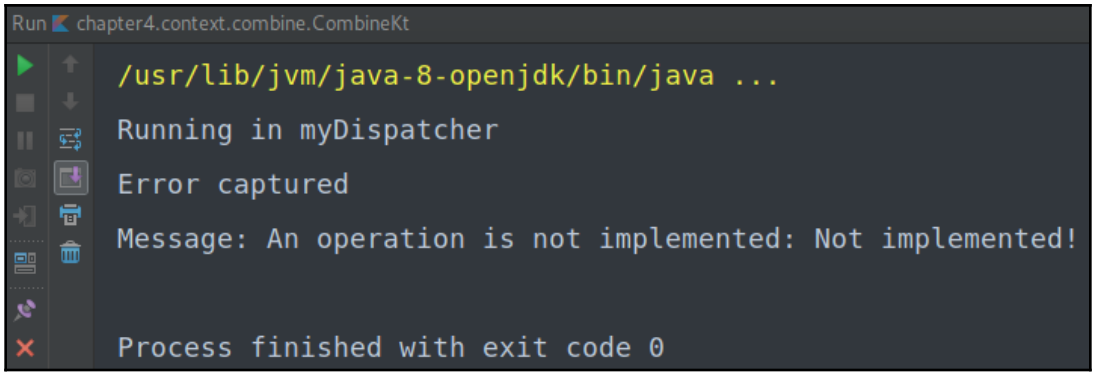
As we have seen in this chapter, there are different types of elements that can be part of a context. We can mix these elements to create a context that meets our needs.

Combining contexts

For example, let's say that you want to have a coroutine running in a specific thread, and at the same time you want to set an exception handler for it. To do this, you can combine both by using the plus operator:

```
main(args: Array<String>) = runBlocking {
    val dispatcher = newSingleThreadContext("myDispatcher")
    val handler = CoroutineExceptionHandler({ _, throwable ->
        println("Error captured")
        println("Message: ${throwable.message}")
    })

    launch(dispatcher + handler) {
        println("Running in ${Thread.currentThread().name}")
        TODO("Not implemented!")
    }.join()
}
```

In this case, we are combining a single thread dispatcher with an exception handler, and the coroutine will behave accordingly:



You can, of course, create a variable to hold the combined contexts, to avoid having to use the plus more than once:

```
val context = dispatcher + handler

launch(context) { ... }
```

Separating contexts

You can also remove context elements from a combined context if needed. To do this, you need to have a reference to the *key* of the element that you want to remove. For example, let's modify the previous example so that we separate the combined context:

```
fun main(args: Array<String>) = runBlocking {
    val dispatcher = newSingleThreadContext("myDispatcher")
    val handler = CoroutineExceptionHandler({ _, throwable ->
        println("Error captured")
        println("Message: ${throwable.message}")
    })

    // Combine two contexts together
    val context = dispatcher + handler

    // Remove one element from the context
    val tmpCtx = context.minusKey(dispatcher.key)

    launch(tmpCtx) {
        println("Running in ${Thread.currentThread().name}")
        TODO("Not implemented!")
    }.join()
}
```

This will be the same as if we use `launch(handler) {...}`. Notice here that the thread corresponds to the default dispatcher instead of it being the thread of `dispatcher`:

Temporary context switch using withContext

When we are already in a suspending function we can change the context for a block of code using `withContext()`. This is a suspending function that will use the given context to execute the block of code. For example, if we need to execute an operation in a different thread but we will always wait for it to finish before continuing. Let's start with this example:

```
fun main(args: Array<String>)= runBlocking {
    val dispatcher = newSingleThreadContext("myThread")
    val name = async(dispatcher) {
        // Do important operation here
        "Susan Calvin"
    }.await()

    println("User: $name")
}
```

Here we are using `async()` to do an operation using the context `dispatcher`, but since `async()` returns a `Deferred<String>`, we are forced to call `await()` right away so that we suspend until the name is ready.

Instead of that, we can use `withContext()`. This function will not return a `Job` or a `Deferred`; it will return the value corresponding to the last statement of the lambda that we pass to it:

```
fun main(args: Array<String>) = runBlocking {
    val dispatcher = newSingleThreadContext("myThread")
    val name = withContext(dispatcher) {
        // Do important operation here and then return the name
        "Susan Calvin"
    }

    println("User: $name")
}
```

This way, the code behaves sequentially: `main` will suspend until the name is retrieved, without the need to call `join()` or `await()` to do so.

Summary

This chapter has introduced many new topics that are going to be needed for the upcoming, more advanced topics. Let's review the chapter once more to refresh our memory:

- We started by doing some Android programming, creating a `RecyclerView` and displaying the news from the RSS feeds.
- We talked about how an `Adapter` maps a set of data into views, and how a `ViewHolder` is used as part of that.
- We learned that Android's `RecyclerView` will avoid creating views as much as possible; instead, it will recycle them as the user scrolls.
- We talked about suspending functions, and learned that they offer a flexible way to define suspending code.
- We mentioned that async functions – a function returning an implementation of `Job` – should never be part of a public API, to avoid the risk of forcing a certain implementation.
- We covered the interesting topic of the coroutine context and how it works.
- We listed many different types of coroutine contexts, starting with dispatchers and moving on to exception handlers and the unique non-cancellable context.
- We learned how to mix many contexts into one to get the expected behavior in our coroutines.
- We discussed the details of separating a combined context by removing the key of one of its elements.
- We covered the interesting `withContext()`, a suspend function that allows us to switch to a different context, without having to involve `Job` in the process.

In the next chapter, we will dive into more advanced topics. We will talk about generators, covering both iterators and sequences, and we will use them to create a potentially infinite data source that is loaded on demand.

5
Iterators, Sequences, and Producers

When dealing with data sources, it's common to retrieve and display information on demand. In a news reader, for example, you may want to fetch only some articles when the user first opens the app, and fetch more as the user scrolls down. And depending on the source of the information, this data can potentially be infinite.

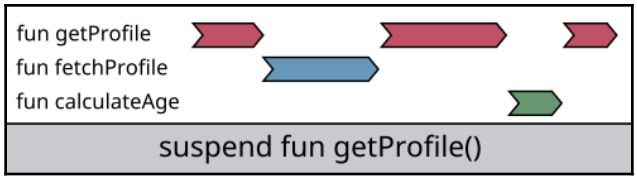
In this chapter, we will talk about different ways to implement data sources as suspending functions and as async functions by leveraging Kotlin's concurrency primitives. We will start by doing sample implementations using the Fibonacci sequence both as an iterator and as a sequence, and then we will add suspending data retrieval to our RSS reader.

Some of the topics covered in this chapter are:

- Suspendable sequences
- Suspendable iterators
- Yielding data in suspendable data sources
- Differences between sequences and iterators
- Async data retrieval using producers
- Real-life examples of a producer

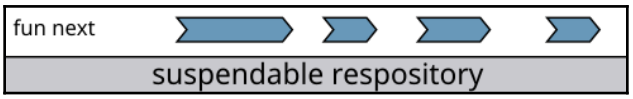
Suspendable sequences and iterators

So far, we have discussed and implemented suspending functions in one way: as functions that suspend while waiting for the execution of one or more computations to happen. A simple representation of this is as follows:



In this example, **getProfile** starts its execution and, soon after, it suspends to wait for **fetchProfile** to execute. Once the response has been processed, **getProfile** suspends once more, this time to call **calculateAge**. When **calculateAge** finishes, the execution of **getProfile** continues until completion.

In this chapter, we will cover a different scenario: writing functions that are suspended between executions. For example, we can have a repository that is suspended until the next page is required:



In the previous example, the `next()` function will yield the first element from the data source, and will suspend immediately after returning it. Whenever `next()` is called again, the execution will be resumed until a value is yielded, and then the function will suspend once more.

We will start by taking a look at suspendable sequences and suspendable iterators. Both of them share a few important characteristics:

- While they suspend between invocations, *they can't be suspended during execution*. Because of this, it's possible to iterate on them without having to be in a suspending computation.
- Their builders don't receive a `CoroutineContext`. This means that by default their code will be executed on the same context that they are invoked in.
- They can only suspend after yielding information. For this to happen, the `yield()` or `yieldAll()` function must be invoked.

Yielding values

As mentioned previously, yielding a value will result in the suspension of the sequence or iterator until a value is requested from it again. A simple example of this can be seen here:

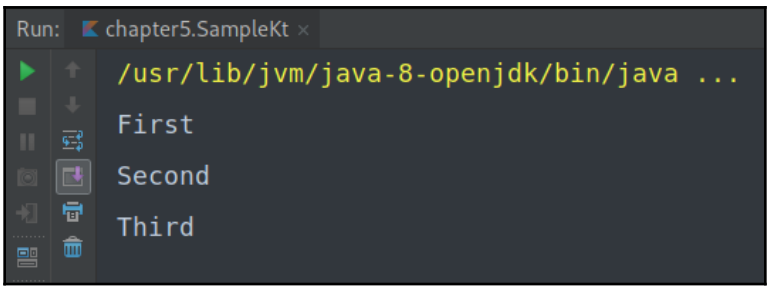
```
val iterator = buildIterator {
    yield("First")
    yield("Second")
    yield("Third")
}
```

This code will build an iterator that contains three elements. The first time an element is requested, the first line will be executed, yielding the value `"First"` and suspending execution afterwards.

When the next element is requested, the second line will be executed, yielding `"Second"` and suspending again. Therefore, to be able to obtain the three elements that this iterator contains, it's simply necessary to call the `next()` functions three times:

```
fun main(args: Array<String>) {
    val iterator = buildIterator {
        yield("First")
        yield("Second")
        yield("Third")
    }

    println(iterator.next())
    println(iterator.next())
    println(iterator.next())
}
```

In this example, the iterator was suspended three times, each time after a value was yielded:



As pointed out previously, because they can't suspend during their execution, suspending sequences and iterators can be called from non-suspendable code. Notice how in this case `next()` is being called from `main` directly, with no coroutine builder being involved.

Iterators

Iterators are particularly useful to go through a collection of elements in order. Some characteristics of Kotlin's iterators are:

- They can't retrieve an element by index, so elements can only be accessed in order
- They have the function `hasNext()`, which indicates whether there are more elements
- Elements can be retrieved in a single direction only; there's no way to retrieve previous elements
- They can't be reset, so they can only be iterated once

To create a suspending iterator, we use the builder `buildIterator()`, passing it a lambda with the body of the iterator. This will return an `Iterator<T>` where `T` will be determined by the elements that the iterator yields, unless otherwise specified:

```
val iterator = buildIterator {
    yield(1)
}
```

In this case, for example, `iterator` will be of type `Iterator<Int>`. If for some reason we want to override this definition, we can define a type, and it will work as long as all the values that are yielded are compliant with the type. For example, the following will work:

```
val iterator : Iterator<Any> = buildIterator {
    yield(1)
    yield(10L)
    yield("Hello")
}
```

And the following will result in a compile error:

```
val iterator : Iterator<String> = buildIterator {
    yield("Hello")
    yield(1)
}
```

Interacting with an iterator

Now that we have an idea of what an iterator is, it's a good time to talk about how we can use one. The current section covers common ways to use iterators, along with some explanations on how to avoid exceptions and some details on when the values are computed.

Going through all the elements

In some cases, you want to retrieve all the elements of an iterator at once, instead of doing it one by one. Notice that because we are talking about this topic from the point of view of a data source, our use case is more oriented towards getting one element or a group of them at a time.

You can use the functions `forEach()` and `forEachRemaining()` to go through the whole iterator. Because an iterator can only go forward, both functions will work the same:

```
iterator.forEach {
    println(it)
}
```

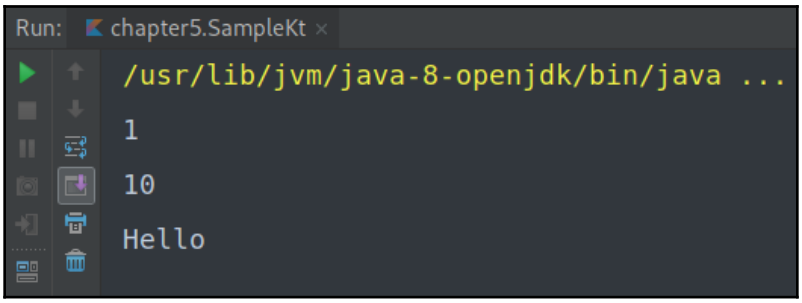
While both functions will behave the same way, using one or the other will improve the readability of the code. For example, if you know that some of the elements may have been read already, use `forEachRemaining()` to make it clear to anyone reading the code that some of the elements may not be in the iterator by that point.

Getting the next value

As mentioned previously, to read elements from an iterator you can use `next()`. For example, this code will simply print each of the elements:

```
fun main(args: Array<String>) {
    val iterator : Iterator<Any> = buildIterator {
        yield(1)
        yield(10L)
        yield("Hello")
    }

    println(iterator.next())
    println(iterator.next())
    println(iterator.next())
}
```

Shown as follows:

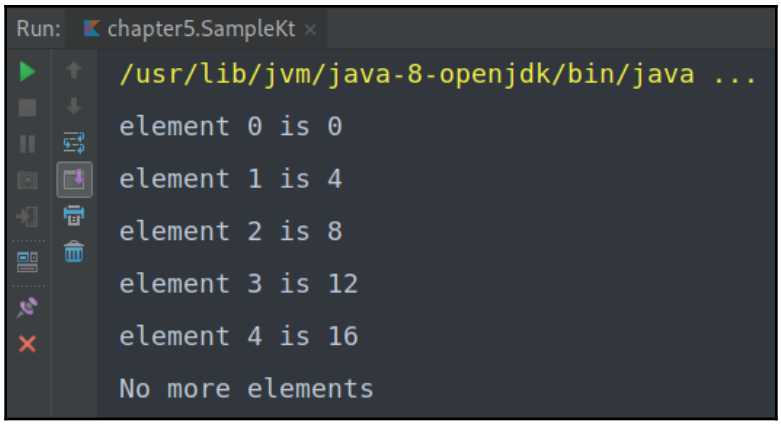


Validating whether there are more elements

Another useful function is `hasNext()`, which returns `true` if the iterator can emit one more element, and `false` otherwise. An example of this could be the following:

```
fun main(args: Array<String>) {
    val iterator = buildIterator {
        for (i in 0..4) {
            yield(i * 4)
        }
    }

    for (i in 0..5) {
        if (iterator.hasNext()) {
            println("element $i is ${iterator.next()}")
        } else {
            println("No more elements")
        }
    }
}
```

In this example, we create an iterator that yields a number the first five times it's invoked. Then we call it six times. The result is as expected:

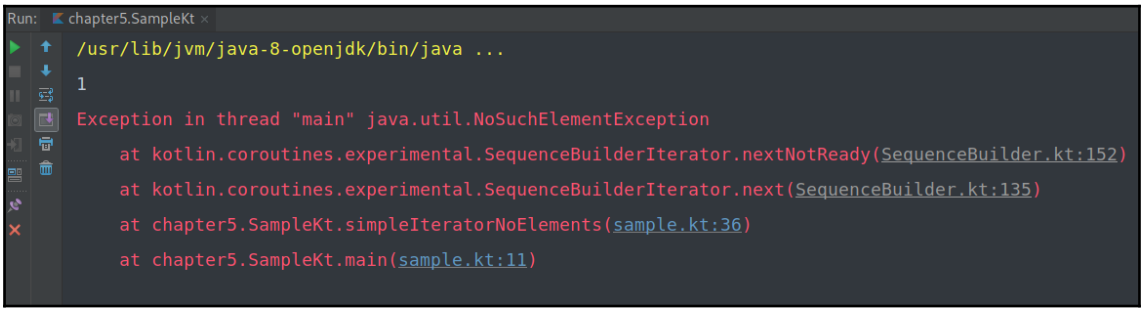


Calling next() without validating for elements

When obtaining elements from an iterator using `next()`, it's recommended to always call `hasNext()` first. Failing to validate that there's an element to retrieve will result in `NoSuchElementExceptions` during execution, for example:

```
val iterator = buildIterator {
    yield(1)
}

println(iterator.next())
println(iterator.next())
```

This iterator can only yield one value. Because `next()` is being called without validating the element, an exception will be thrown:



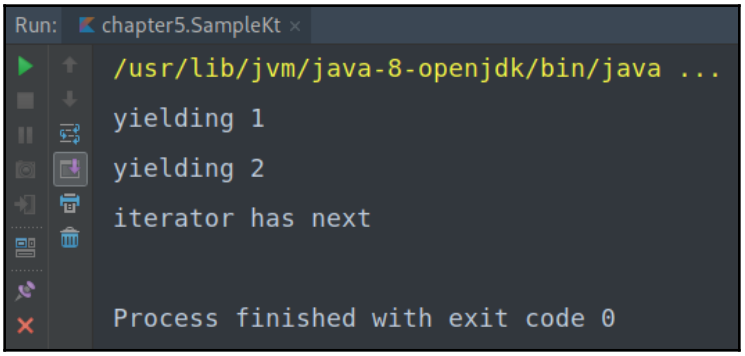
A note on the inner working of hasNext()

In order for `hasNext()` to work, the runtime will resume the execution of the coroutine. If a new value is yielded, the function will return `true`. On the contrary, if the execution of the iterator is completed without yielding more values, the function will return `false`.

If a value was yielded because of a call to `hasNext()`, the value will be kept and returned the next time that `next()` is called. Let's look at an example of this:

```
fun main(args: Array<String>) {
    val iterator = buildIterator {
        println("yielding 1")
        yield(1)
        println("yielding 2")
        yield(2)
    }

    iterator.next()

    if (iterator.hasNext()) {
        println("iterator has next")
        iterator.next()
    }
}
```

This code creates a simple iterator that yields two values, printing a message right before the call to `yield()`. The code obtains the first element, then validates whether there's a second element and prints it if there is one. Because the call to `hasNext()` will make the iterator yield the second value, the second call to `next()` will only retrieve it:



As you can see in the previous screenshot, `"yielding 2"` will be printed before `"iterator has next"`, because it was yielded during `hasNext()`.

Sequences

Suspending sequences are quite different from suspending iterators, so let's take a look at some of the characteristics of suspending sequences:

- They can retrieve a value by index
- They are stateless, and they reset automatically after being interacted with
- You can take a group of values with a single call

In order to create a suspending sequence, we will use the builder `buildSequence()`. This builder takes a suspending lambda and returns a `Sequence<T>`, where `T` can be inferred by the elements that are yielded, as for example the following:

```
val sequence = buildSequence {
    yield(1)
}
```

This will make the sequence a `Sequence<Int>`. But, similar to iterators, you can always specify a `T` as long as the values that are yielded are compliant:

```
val sequence: Sequence<Any> = buildSequence {
    yield("A")
    yield(1)
    yield(32L)
}
```

Interacting with a sequence

Consider the following sequence:

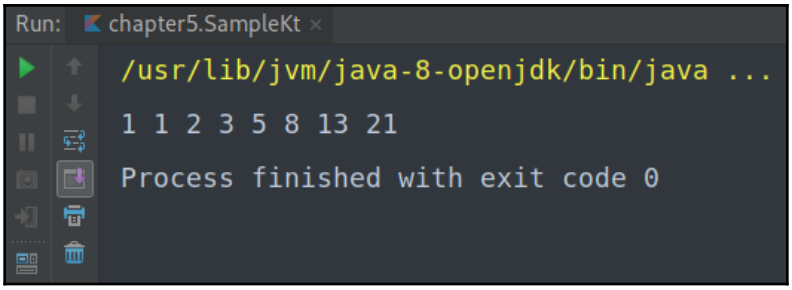
```
val sequence = buildSequence {
    yield(1)
    yield(1)
    yield(2)
    yield(3)
    yield(5)
    yield(8)
    yield(13)
    yield(21)
}
```

Reading all the elements in the sequence

To go through all the elements in the sequence, you can use `forEach()` and `forEachIndexed()`. Both of them will behave similarly, but `forEachIndexed()` is an extension function that provides the index of the value along with the value itself:

```
sequence.forEach {
    print("$it ")
}

sequence.forEachIndexed { index, value ->
 println("element at $index is $value")
}
```

The first of the previous snippets will print the eight numbers in the sequence:



Obtaining a specific element

The previous sequence can yield up to eight values. If you want to retrieve values by index, you can use one of the following functions.

elementAt

This function takes an index and returns the element in that position, as follows for example:

```
sequence.elementAt(4)
```

This will return `5`, matching the fifth value yielded in the sequence.

elementAtOrElse

This takes an index and a lambda to be executed if there's no element at the given index. The lambda will receive the index that was passed. For example, the following code will return `20`, which is the index `10` that we passed times `2` – because the sequence has only eight elements:

```
sequence.elementAtOrElse(10, { it * 2 })
```

elementAtOrNull

This function takes an index and returns `T?`. It will return `null` if there's no element at the given index, as follows for example:

```
sequence.elementAtOrNull(10)
```

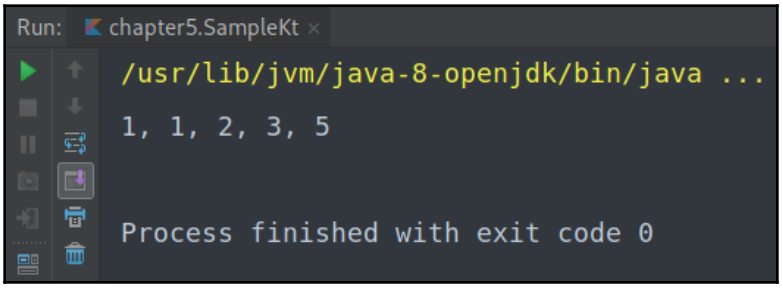
This will simply return `null`, given that the sequence only has eight elements.

Obtaining a group of elements

The values can also be retrieved in groups. For example, you can take a group of values at once:

```
val firstFive = sequence.take(5)
println(firstFive.joinToString())
```

This will print the first five values separated by comma:



Notice that `take()` is an intermediate operation, so it returns a `Sequence<T>` that will be actually calculated at a later time—when a terminal operation in invoked. So, in our example, calling `take(5)` will not actually have the sequence yield the values, but calling `joinToString()` will.

Sequences are stateless

At the beginning of this section, we mentioned that suspending sequences are stateless and that they reset after being used. Consider the following sequence:

```
val sequence = buildSequence {
    for (i in 0..9) {
        println("Yielding $i")
        yield(i)
    }
}
```

It's a simple sequence that can yield up to 10 values. Now, we might read the values like this:

```
fun main(args: Array<String>) {
    println("Requesting index 1")
    sequence.elementAt(1)

    println("Requesting index 2")
    sequence.elementAt(2)

    println("Taking 3")
    sequence.take(3).joinToString()
}
```

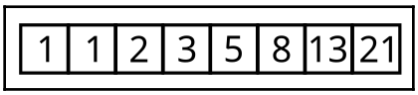
You may notice that a sequence will simply execute from the beginning for each call, which is different from using an iterator:

Suspending Fibonacci

The main topic of this chapter is data sources, so we will use a well-known sequence to get an implementation of suspending sequences and iterators: the famous Fibonacci sequence. I am sure that you have already written algorithms to calculate the Fibonacci sequence before, but this may be the first time that you will do so using suspending functions.

If you aren't familiar with it, the Fibonacci sequence is a sequence of numbers in which each number is the result of adding the previous two.

You can see the first eight numbers of the sequence in the following image:



The objective that we have, then, is to write a function that returns numbers from that sequence on demand, and stays suspended between calls. We will first do an implementation with a sequence and then do a similar implementation with an iterator.

Writing a Fibonacci sequence

The following is an initial implementation for a sequence that can yield the numbers on demand:

```
val fibonacci = buildSequence {
    yield(1)
    var current = 1
    var next = 1
    while (true) {
        yield(next)
        val tmpNext = current + next
        current = next
        next = tmpNext
    }
}
```

The following is an explanation of how this code works:

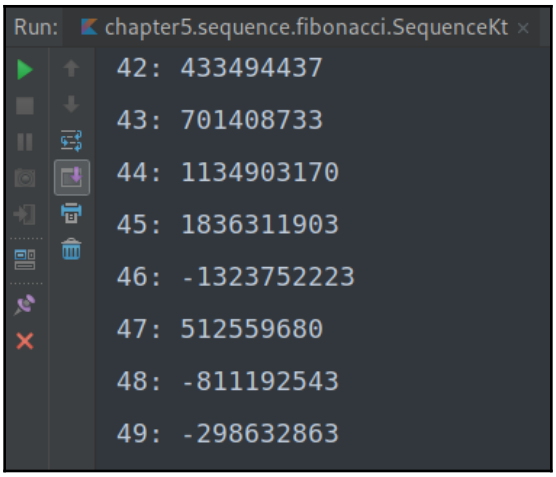
- The first thing that the sequence does is to yield the number one. This is useful because we don't want to actually have to calculate the first numbers of the sequence.
- Once the first number is yielded, the sequence suspends.
- Once the second number is requested, the sequence will create two variables, `current` and `next`. Each of them is initialized to the number one, given that `1` is the first and second number in the sequence.
- Then we enter an infinite loop. This part is what allows the data source to emit as many numbers from the sequence as requested.
- The next time that a number from the sequence is requested, the first thing that happens is that the value of `next` is yielded – so the sequence is suspended.
- And from that point on, whenever a value is requested, both `current` and `next` are recalculated to contain the new values and `next` is yielded.

So let's test this code. Let's include a simple main function that prints the first 50 numbers in the sequence, separated by commas:

```
fun main(args: Array<String>) {
    val fibonacci = buildSequence {
        ...
    }

    val indexed = fibonacci.take(50).withIndex()

    for ((index, value) in indexed) {
        println("$index: $value")
    }
}
```

Similar to `take()`, `withIndex()` is an intermediate operation.

If you look at the last elements printed, you will notice that some of them are actually negative numbers:



This is because the sum of the elements with index `44` and `45` exceed the maximum value that an `Int` can hold. So it leads to an overflow. To add support for more numbers, we are simply going to change the implementation to use `Long`:

```
val fibonacci = buildSequence {
    yield(1L)
    var current = 1L
    var next = 1L
    while (true) {
        yield(next)
        val tmpNext = current + next
        current = next
        next = tmpNext
    }
}
```

Notice that this implementation will have similar issues around element `92`. But for now, it is good enough for us.

Writing a Fibonnaci iterator

The body of the iterator can be exactly the same as the one in the sequence:

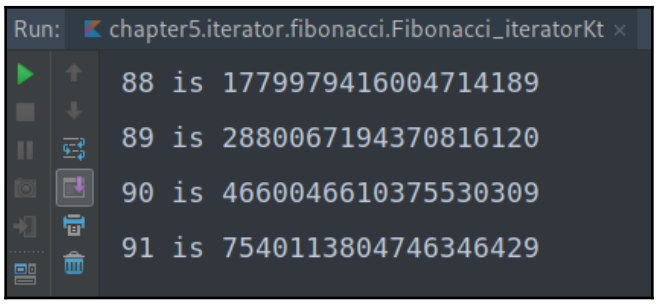
```
val fibonacci = buildIterator {
    yield(1L)
    var current = 1L
    var next = 1L
    while (true) {
```

```
            yield(next)
            val tmpNext = current + next
            current = next
            next = tmpNext
        }
    }
```

So now we can easily, and on demand, get up to `92` numbers in the sequence from this iterator:

```
for (i in 0..91) {
    println("$i is ${fibonacci.next()}")
}
```

If you are curious, these are the last few elements that can be calculated with this implementation:



Producers

As we discussed previously, there is a limitation in both sequences and iterators: they can't suspend during execution. This is a big limitation for many use cases, because often what would be ideal is to be able to suspend while waiting for other operations to complete.

To overcome this limitation, we have to use producers. Their philosophy and usage are close to the ones of suspending sequences and iterators, but with a few small differences. The following are some important details on producers:

- A producer suspends after a value is produced, and resumes when a new value is requested – similar to suspending sequences and iterators
- A producer can be built with a specific `CoroutineContext`

- The body of the suspending lambda that is passed can suspend at any point
- Because it can suspend at any point, a value from a producer can only be received in a suspending computation
- Because it works using a channel, you can think of it like a stream of data, in which receiving an element also removes the element from the stream

Creating a producer

To create a producer, the coroutine builder `produce()` must be called. It returns a `ReceiveChannel<E>`. Because producers are built on top of channels, to yield elements in a producer, you use the function `send(E)`:

```
val producer = produce {
    send(1)
}
```

It's possible to specify a `CoroutineContext` in the same way that you would with `launch()` or `async()`:

```
val context = newSingleThreadContext("myThread")

val producer = produce(context) {
    send(1)
}
```

Similar to iterators and sequences, you can specify a type, and it will work as long as the elements being emitted are compliant with it:

```
val producer : ReceiveChannel<Any> = produce(context) {
    send(5)
    send("a")
}
```

In `chapter 6`, *Channels – Share Memory by Communicating*, we will talk in more detail about channels, how they work, and how to use them.

Interacting with a producer

Interacting with a producer is a mix between how it's done with a sequence and with an iterator. Because in the next chapter we will cover channels in complete detail, for now we will talk only about some of the functions of `ReceiveChannel` that are most relevant.

Reading all the elements in the producer

To go through all the elements in the producer, you can use the `consumeEach()` function, as follows for example:

```
val context = newSingleThreadContext("myThread")

val producer = produce(context) {
    for (i in 0..9) {
        send(i)
    }
}
```

This producer will produce up to 10 numbers; to retrieve all of them we can simply call `consumeEach()` on the producer:

```
fun main(args: Array<String>) {
    producer.consumeEach {
        println(it)
    }
}
```

Receiving a single element

In order to read a single element from a producer, the function `receive()` can be used, as follows for example:

```
val producer = produce {
    send(5)
    send("a")
}

fun main(args: Array<String>) {
    println(producer.receive())
    println(producer.receive())
}
```

his code will first print the number five, and then it will print the letter `a`.

Taking a group of elements

You can also read values using `take()`, sending the amount of elements to take as a parameter. For example, you can consume the first three elements from a producer like this:

```
producer.take(3).consumeEach {
    println(it)
}
```

Notice that `take()` on a `ReceiveChannel<E>` will return a `ReceiveChannel<E>` too, and because `take()` is an intermediate operation, the actual value of those three elements will be calculated when a terminal operation happens – in this case the terminal operation will be `consumeEach()`.

Taking more elements than those available

With iterators and sequences, trying to retrieve more elements than those possible to yield would result in an exception of the type `NoSuchElementException`.

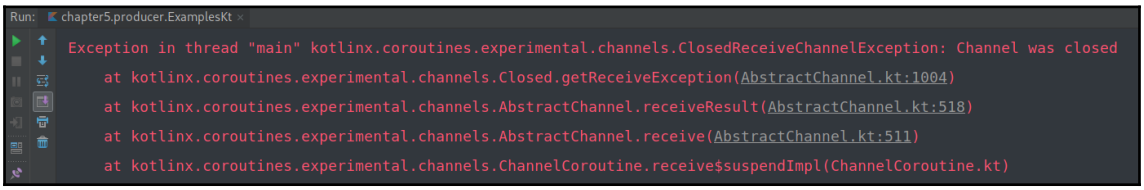
With producers, on the other hand, it depends on how you are trying to obtain that element. For example, given a channel that can emit up to 10 elements, the following code would not fail:

```
producer.take(12).consumeEach {
    println(it)
}
```

This is because `consumeEach` will stop once there are no more elements, regardless of how many we intended to take. If we modify the code to add another `receive()` for another element, the application will crash:

```
producer.take(12).consumeEach {
    println(it)
}

val element = producer.receive()
```

It will crash because once the producer has completed its execution, the channel is closed. So, the exception being thrown is `ClosedReceiveChannelException`:

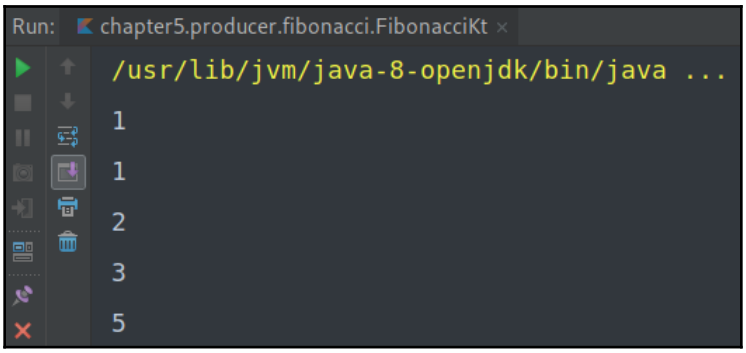


Suspending a Fibonacci sequence using a producer

The implementation of a Fibonacci sequence using a producer is actually quite similar to the ones for iterators and sequences. You only need to replace `yield()` with `send()`:

```
val context = newSingleThreadContext("myThread")

val fibonacci = produce(context) {
    send(1L)
    var current = 1L
    var next = 1L
    while (true) {
        send(next)
        val tmpNext = current + next
        current = next
        next = tmpNext
    }
}

fun main(args: Array<String>) = runBlocking {
    fibonacci.take(10).consumeEach {
        println(it)
    }
}
```

The previous example will print the first ten numbers produced:



Producers in action

To finish this chapter, we are going to update our RSS Reader so that it fetches information on demand. The idea is that we will fetch one feed at the beginning, and from there on we fetch one more feed as the user scrolls down close to the end of the list. That way, we reduce the amount of data consumed.

Having the adapter request more articles

We want to be able to request articles as the user scrolls, so the first step is to connect the adapter with the articles to a listener that can fetch more articles when needed. To do this, we are first going to add an interface to the same file where the class `ArticleAdapter` resides:

```
interface ArticleLoader {
    suspend fun loadMore()
}

class ArticleAdapter: RecyclerView.Adapter<ArticleAdapter.ViewHolder>(){
    ...
}
```

This interface defines a suspending function that will load more articles if possible. Now we need to have the adapter receive an instance of that interface in the constructor, so that anyone using the adapter sends the proper listener. We will change the signature of the class to this:

```
class ArticleAdapter(
        private val loader: ArticleLoader
): RecyclerView.Adapter<ArticleAdapter.ViewHolder>() {
    ...
}
```

Now we need to add a flag so that the listener is not called more than it is necessary. Let's add a variable `loading` to the adapter. We can put it below the definition of the list of articles, like this:

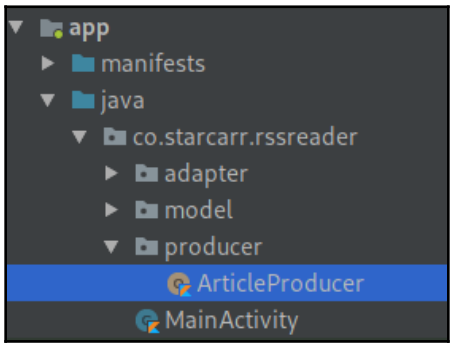
```
private val articles: MutableList<Article> = mutableListOf()
private var loading = false
```

Finally, we have to update the function `onBindViewHolder()` so that it calls the listener when the binding of one of the elements close to the bottom happens:

```
override fun onBindViewHolder(holder: ViewHolder, position: Int) {
    val article = articles[position]

    // request more articles when needed
    if (!loading && position >= articles.size - 2) {
        loading = true

        launch {
            loader.loadMore()
            loading = false
        }
    }

    ...
}
```

Creating a producer that fetches feeds on demand

Now we will create the producer. To do so, let's first create a package `co.starcarr.rssreader.producer` and put the file `ArticleProducer.kt` inside it:



In that file, we will create an object `ArticleProducer` and we will move some of the code from the activity there, starting with the list of feeds:

```
object ArticleProducer {
    private val feeds = listOf(
            Feed("npr", "https://www.npr.org/rss/rss.php?id=1001"),
            Feed("cnn", "http://rss.cnn.com/rss/cnn_topstories.rss"),
            Feed("fox",
"http://feeds.foxnews.com/foxnews/latest?format=xml")
    )
}
```

Notice that we are removing the invalid feed for now.

We will also move the dispatcher and the factory here:

```
object ArticleProducer {

    ...
    private val dispatcher = newFixedThreadPoolContext(2, "IO")
    private val factory = DocumentBuilderFactory.newInstance()
}
```

Finally, we will move the function `asyncFetchArticles()`. Now, it will be called `fetchArticles()` and will return a `List<Article>` directly:

```
object ArticleProducer {

    ...

    private fun fetchArticles(feed: Feed) : List<Article> {
        ...
    }
}
```

In order for this function to compile, we will have to add a return statement at the end of it. Something simple like the following will work:

```
private fun fetchArticles(feed: Feed) : List<Article> {
    val builder = factory.newDocumentBuilder()
    val xml = builder.parse(feed.url)
    val news = xml.getElementsByTagName("channel").item(0)

    return (0 until news.childNodes.length)
            .map { news.childNodes.item(it) }
            .filter { Node.ELEMENT_NODE == it.nodeType }
            .map { it as Element }
            ...
}
```

If you still had the call `delay()` that we added in a previous chapter inside this function, please remove it as well – this function is no longer suspending, so it can't call other suspending functions unless we use a coroutine builder.

So we are only missing the actual producer. Since we want to fetch each feed on demand, we only need to iterate through the list of feeds, sending the articles of each feed. A simple implementation would be the following:

```
val producer = produce(dispatcher) {
    feeds.forEach {
        send(fetchArticles(it))
    }
}
```

Notice that now we are sending all the articles of each feed through the producer. So this is a `Producer<List<Article>>`.

Adding the articles to the list on the UI

Since we moved much of the code in `MainActivity` to `ArticleProducer`, we now only have `onCreate()` and `asyncLoadNews()` in it. We will delete `asyncLoadNews()` first – along with the call to it that was being made in `onCreate()` – and then we will update the signature of the activity so that it implements `ArticleLoader`:

```
class MainActivity : AppCompatActivity(), ArticleLoader {
    ...
}
```

Now we will need to implement the suspending function `loadMore()`. Let's begin by having it retrieve the producer from `ArticleProducer` and keeping a reference to it in a variable:

```
override suspend fun loadMore() {
    val producer = ArticleProducer.producer
}
```

Now that we have access to the producer, we need to validate that the producer is not closed, and request more articles from it if it's open. Those articles will be added to the adapter so that they are displayed:

```
override suspend fun loadMore() {
    val producer = ArticleProducer.producer

    if (!producer.isClosedForReceive) {
        val articles = producer.receive()

        launch(UI) {
            findViewById<ProgressBar>(R.id.progressBar).visibility =
View.GONE
            viewAdapter.add(articles)
        }
    }
}
```

We also need to pass `this` as the loader to the adapter:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    viewManager = LinearLayoutManager(this)
    viewAdapter = ArticleAdapter(this)
    ...
}
```

And finally, add the call to `loadMore()` inside `onCreate()` using `launch()`, shown as follows:

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

    launch {
        loadMore()
    }
}
```

If you test the application now, you will notice that when opening the app, the scroll bar indicates that there's not much content. But once you start scrolling and get close to the bottom, the scroll bar will become smaller, indicating that there are now more articles to read.

With just a few changes to the code, we have been able to implement a suspending data source. The code is not complex, yet the functionality of the app has improved.

Summary

This chapter covered a lot of topics; they were an interesting display of how suspending computations can be used in a more creative way. Furthermore, we were able to see how this can impact application development by allowing us to write simpler code that is still concurrent. Let's summarize this chapter's lessons:

- We talked about a different type of suspending function that are suspended when they are not needed.
- Some characteristics of a sequence are: it's stateless, so it resets itself after each invocation; it allows us to retrieve information by index; and it allows us to take a group of values at once.
- Some characteristics of an iterator are: it has a state; it can only be read in one direction, so previous elements cannot be retrieved; and it doesn't allow us to retrieve elements by index.
- Both sequences and iterators can suspend after yielding one or more values, but they cannot suspend as part of their execution, so they are good for data sources that don't need async operations, such as sequences of numbers.

- Because sequences and iterators can't suspend during execution, they can be called from non-suspending computations.
- Producers can suspend at any point, including during their execution.
- Because producers can suspend during execution, they can only be invoked from suspending computations and coroutines.
- Producers use channels to output data.

In the next chapter, we will talk about channels and about the safety of concurrency based on suspension and communication. The next chapter will cover some of the most relevant topics for concurrency in Kotlin, and we will once more update our RSS Reader to put this new knowledge into practice in real-world situations.

6
Channels - Share Memory by Communicating

Many errors related to concurrency happen when memory is shared between different threads, for example, having an object that will be modified by different threads. Sharing memory this way is dangerous because unless there's bulletproof synchronization, there will be scenarios in which the shared object will enter a state that it should never enter—and writing bulletproof synchronization is more difficult than it may seem.

Deadlocks, race conditions, and atomicity violations are related to shared states. Sometimes they happen because a shared state is invalid, and other times they will cause the state to become inconsistent.

In order to overcome these issues, modern programming languages like Kotlin, Go, and Dart provide channels. Channels are tools that will help you write concurrent code, which, instead of sharing a state, allows threads to communicate by sending and receiving messages.

In this chapter, we will talk about channels, and we will cover the following topics:

- Understanding channels with real-life examples
- Types of channels
- Interacting with channels
- Real implementation of channels for the RSS Reader

Understanding channels

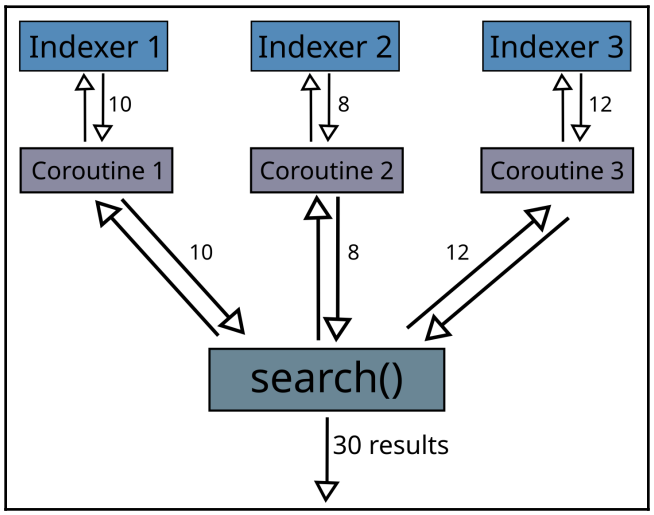
Channels are tools to allow safe communication between concurrent code. They allow concurrent code to communicate by sending messages. Channels can be thought of as pipelines for safely sending and receiving messages between different coroutines – no matter the thread in which they are running.

Let's look at some examples of how channels can be used in real-life, non-trivial scenarios. This will help you to understand how useful channels can be when implementing tasks that require many coroutines to do collaborative work. As an exercise, try to think of a way to solve the problems listed here without channels.

Use case – streaming data

Recently, I was faced with a relatively simple programming task: querying 10 content indexers for a specific keyword, and then displaying the results of the search to the user. A valid first approach would have been to launch a coroutine for each content indexer, returning `Deferred<List<ResultDto>>`, and upon completion of all of the coroutines, merging the results and sending them to the UI.

A representation of this approach could look like this:



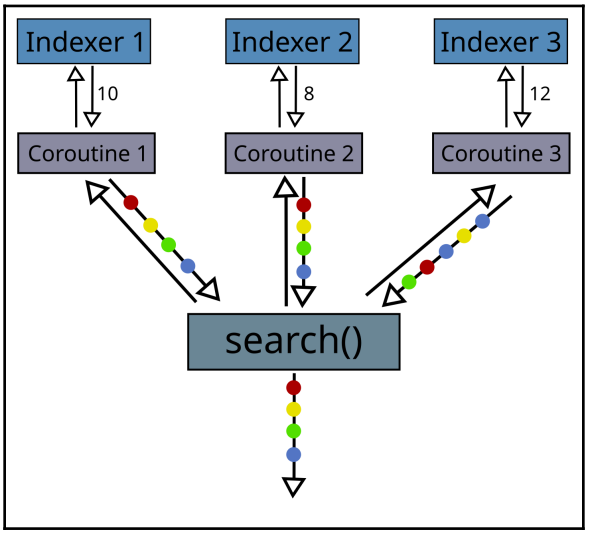
In this example, `search()` starts three coroutines, giving them the keywords to search. Each coroutine will fetch the results from the indexer. Upon completion of the three coroutines, `search()` will compile and return the results.

The problem with this approach is that each indexer will take a different amount of time to return a response, and some of them may take considerably longer. So, having to wait for all of the coroutines to complete will delay the UI from displaying results, thus preventing the user from interacting with results as soon as they become available.

A better solution requires a couple of changes to be made. First, we want `search()` to return a `Channel<ResultDto>`—as we will see later in the chapter, `ReceiveChannel<ResultDto>` is the best option— this way the results can be sent to the UI as soon as they arrive. Second, we want `search()` to be able to seamlessly receive the results from each coroutine as they arrive as well.

To accomplish this, each coroutine will send results through a single channel as they retrieve and process the response. Meanwhile, `search()` will simply return the channel so that the caller can process the results at their convenience.

It would look something like this:



This way, the caller will receive the results as they are obtained from the content indexer, and the UI will be able to display the results incrementally, allowing the user to interact with them as soon as they are available, for a fluid and fast experience.

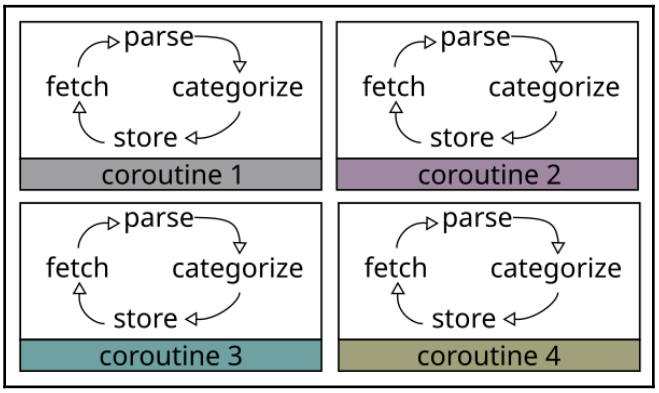
Use case – distributing work

Some time ago I wanted to write an application to index content from Usenet groups. Usenet groups were the precursor to today's forums. Each group encompasses a topic, people can create new threads (called articles), and anyone can reply to those articles.

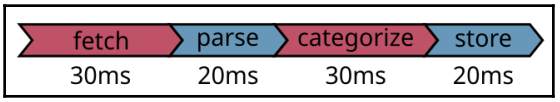
As opposed to forums, though, you don't access Usenet using an internet browser. Instead, you have to connect through a client that supports the NNTP protocol. Also, when accessing a Usenet server, you have a limited amount of concurrent connections to the server, let's say, 40 connections.

In summary, the purpose of the application is to use all the possible connections to retrieve and parse as many articles as possible, categorize them, and store the processed information in a database. A naïve approach to this challenge would be to have one coroutine for each possible connection—40 in the preceding example—having each of them fetch the article, parse it, catalog its contents, and then put it in the database. Each coroutine can move forward to the next article as they finish one.

A representation of this approach could be something like the following:



The problem with this approach is that we aren't maximizing the usage of the available connections, which means that we are indexing at a lower speed than is possible. Suppose that, on average, fetching an article takes 30 milliseconds, parsing its content takes 20, cataloging it takes another 30, and storing it in the database another 20. Here is a simple breakdown for reference:

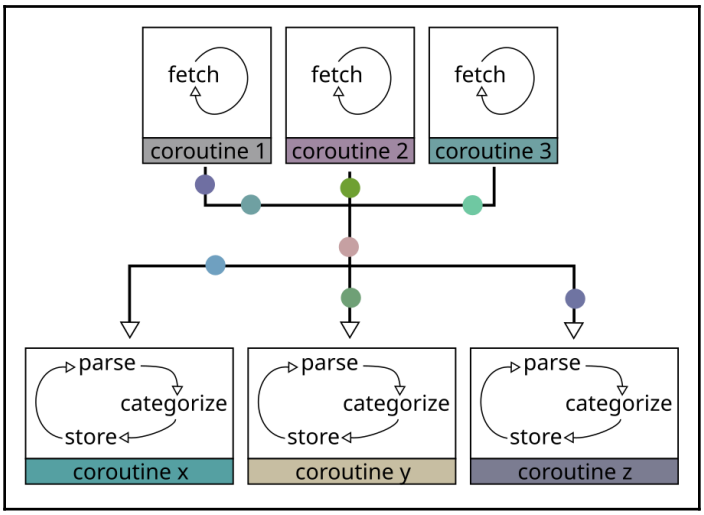


Only 30 percent of the time it takes to index an article is spent on retrieving it, and the other 70 percent is used for processing. Nevertheless, since we are using one coroutine to do all the steps of the indexation, we aren't using the connection for 70 milliseconds per each article that is processed.

A better approach is to have 40 coroutines dedicated only to retrieving articles, and then having a different group of coroutines do the processing. Say, for example, 80 coroutines. By setting the correct configuration between both groups, it would be possible to index content more than three times faster—assuming that we have enough hardware to handle the load—because we could maximize the connection usage so that all the connections are always being used during the whole execution of the indexation. We could retrieve data non-stop, and then maximize the use of hardware to process as much as possible concurrently.

To perform such an implementation, we only need to put a channel between the group of 40 article retrievers and the group of 80 article processors. The idea is that the channel will work as a pipeline in which all the retrievers will put the raw articles as they fetch them, and the processors will take and process them whenever they are available.

The following is a representation of this design:



In this example, the bridge between the coroutines that fetch and the ones that process is the channel. The coroutines fetching data only need to care about fetching and putting the raw response in the channel—so they will maximize the usage of each connection—whereas the processing coroutines will process data as they receive it through the channel.

Because of how channels work, we don't need to worry about distributing the load—as coroutines become available to process, they will receive data from the channel.

Now, after this change, we will be retrieving around three articles from the indexer every 90 milliseconds—that's more than three times faster. The processing may not be this fast; it would depend on whether there's enough hardware to have other coroutines processing data as soon as it gets retrieved.

Types of channels and backpressure

The `send()` function in `Channel` is suspending. The reasoning behind this is that you may want to pause the code that is sending elements until there is someone actually listening for the data. This concept is often referred to as backpressure, and helps to prevent your channels from being flooded with more elements than the receivers can actually process.

In order to configure this backpressure, you can define a buffer for your channel. The coroutine sending data through the channel will be suspended when the elements in the channel have reached the size of the buffer. Once elements are removed from the channel, the sender will be resumed.

Unbuffered channels

Channels without a buffer are called unbuffered channels.

RendezvousChannel

Currently the only implementation of unbuffered channels is `RendezvousChannel`. This implementation of a channel has no buffer at all, so calling `send()` on it will suspend it until a receiver calls `receive()` on the channel. You can instantiate this type of channel in a few different ways. You can do it by calling its constructor:

```
val channel = RendezvousChannel<Int>()
```

You can also use the `Channel()` function. This function has two implementations, one that takes the buffer capacity as parameter, and another that doesn't take any parameters. To get a `RendezvousChannel` you can call the function without a parameter:

```
val rendezvousChannel = Channel<Int>()
```

You will have the same result if you send a capacity of zero:

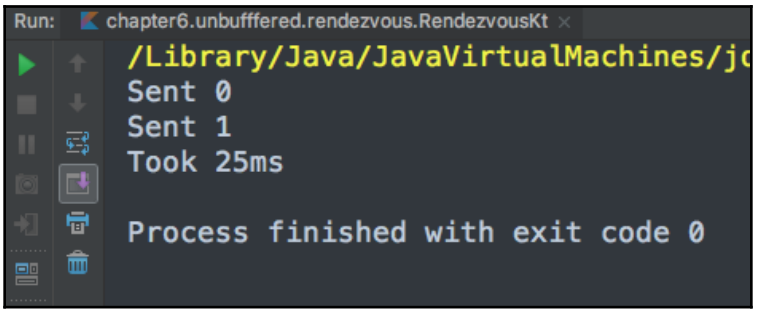
```
val rendezvousChannel = Channel<Int>(0)
```

As mentioned, this channel will suspend the sender's execution until the element on it is retrieved, as the following for example:

```
fun main(args: Array<String>) = runBlocking {
    val time = measureTimeMillis {
        val channel = Channel<Int>()
        val sender = launch {
            repeat(10) {
                channel.send(it)
                println("Sent $it")
            }
        }

        channel.receive()
        channel.receive()
    }

    println("Took ${time}ms")
}
```

In this example, the `sender` coroutine can send up to 10 numbers through the channel. But because only two of the elements are received from the channel before the execution ends, only two elements are sent.

The output of this coroutine looks like the following:



Buffered channels

The second type of channels are those with a buffer. As mentioned before, this type of channel will suspend the execution of any sender whenever the amount of elements in the channel is the same as the buffer size. There are a few types of buffered channels, which differ according to the size of the buffer.

LinkedListChannel

If you want a channel in which you are able to send an unlimited amount of elements without it suspending, then you need a `LinkedListChannel`. This type of channel will not suspend any senders. In order to instantiate this type of channel, you can use the constructor:

```
val channel = LinkedListChannel<Int>()
```

You can also use the `Channel()` function with the parameter `Channel.UNLIMITED`:

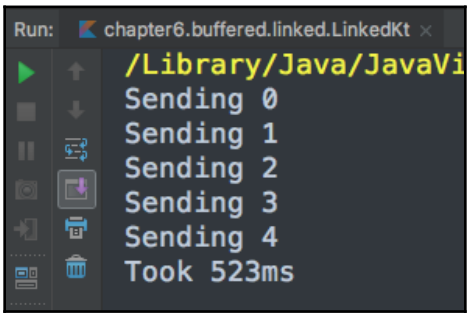
```
val channel = Channel<Int>(Channel.UNLIMITED)
```

`Channel.UNLIMITED` is equal to `Int.MAX_VALUE`, so sending either of them to `Channel()` will retrieve an instance of `LinkedListChannel`.

This channel will never suspend a sender. So for example consider the following code:

```
fun main(args: Array<String>) = runBlocking {
    val time = measureTimeMillis {
        val channel = Channel<Int>(Channel.UNLIMITED)
        val sender = launch {
            repeat(5) {
                println("Sending $it")
                channel.send(it)
            }
        }

        delay(500)
    }

    println("Took ${time}ms")
}
```

The previous implementation will allow `sender` to emit the five elements regardless of the channel not having any receiver to process them:



In reality, there is a limit for how many elements you can emit using `LinkedListChannel`s, the memory. You have to be careful when using this type of channel because it may consume too much memory. It's recommended to use a buffered channel with a defined buffer size based on the requirements and the target devices instead of this.

ArrayChannel

This type of channel has a buffer size bigger than zero and up to `Int.MAX_VALUE - 1`, and will suspend the senders when the amount of elements it contains reaches the size of the buffer. It can be created by sending any positive value lower than `Int.MAX_VALUE` to `Channel()`:

```
val channel = Channel<Int>(50)
```

It can also be created by calling its constructor directly:

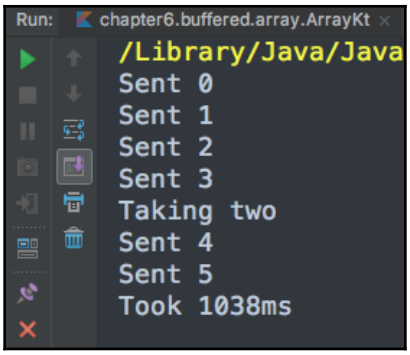
```
val arrayChannel = ArrayChannel<Int>(50)
```

This type of channel will suspend the sender when the buffer is full, and resume it when one or more of the items is retrieved, as the following for example:

```
fun main(args: Array<String>) = runBlocking {
    val time = measureTimeMillis {
        val channel = Channel<Int>(4)
        val sender = launch {
            repeat(10) {
                channel.send(it)
                println("Sent $it")
            }
```

```
            }

            delay(500)
            println("Taking two")
            channel.take(2).receive()
            delay(500)
        }

        println("Took ${time}ms")
    }
```

In this example, `sender` is able to emit up to 10 elements, but because the capacity of the channel is four, it will be suspended before sending the fifth element. Once two elements are received, `sender` is resumed until the buffer is full again:



ConflatedChannel

There is a third type of buffered channel, based on the idea that it's fine if elements that were emitted are lost. This means that this type of channel has a buffer of only one element, and whenever a new element is sent, the previous one will be lost. This also means that the sender will never be suspended.

You can instantiate it by calling its constructor:

```
val channel = ConflatedChannel<Int>()
```

You can also do so by calling the `Channel()` function with the parameter `Channel.CONFLATED`:

```
val channel = Channel<Int>(Channel.CONFLATED)
```

This channel will never suspend a sender. Instead, it will override the elements that were not retrieved. Consider this example:

```
fun main(args: Array<String>) = runBlocking {
    val time = measureTimeMillis {
        val channel = Channel<Int>(Channel.CONFLATED)
        launch {
            repeat(5) {
                channel.send(it)
                println("Sent $it")
            }
        }
        delay(500)
        val element = channel.receive()
        println("Received $element")
    }

    println("Took ${time}ms")
}
```

This implementation will have `element` contain the last value that was sent through the channel. In this case, it will be the number four:

Interacting with channels

The behavior of `Channel<T>` is composed by two interfaces, `SendChannel<T>` and `ReceiveChannel<T>`. In this section, we will take a look at the functions defined by each and how they are used to interact with a channel.

SendChannel

This interface defines a couple of functions to send elements through a channel and other functions in order to validate that it's possible to send something.

Validating before sending

There are a few validations that you can perform before trying to send elements through the channel. The most common one is to validate that the channel has not been closed for sending. To do this, you can use `isClosedForSend`:

```
val channel = Channel<Int>()
channel.isClosedForSend // false
channel.close()
channel.isClosedForSend // true
```

You can also check whether the channel is out of capacity. When a channel is full, it will suspend the next time you call `send`, so this property is useful if you would prefer not to suspend your coroutine at the moment:

```
val channel = Channel<Int>(1)
channel.isFull // false
channel.send(1)
channel.isFull // true
```

Sending elements

To send an element through the channel, you must use the function `send()`, as we saw previously. This is a suspending function that will suspend the sender if the buffer is full in the case of a buffered channel, and will suspend until `receive()` is called if it's a `RendezvousChannel`:

```
val channel = Channel<Int>(1)
channel.send(1)
```

The `send()` function will throw a `ClosedChannelException` if the channel is closed:

```
val channel = Channel<Int>(1)
channel.close()
channel.send(1) // ClosedChannelException is thrown
```

Offering elements

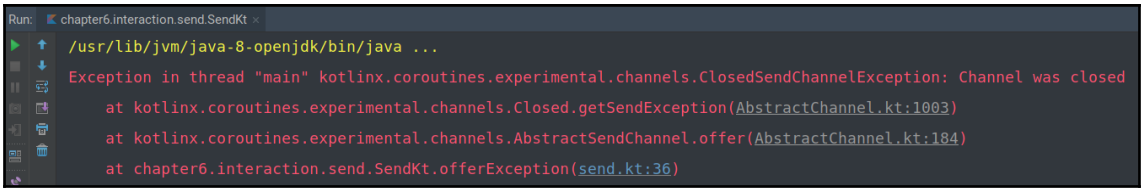
There is a non-suspending function that will allow you to send elements through a channel under certain circumstances. This `offer()` function takes an element to try to add it to the queue, and returns a Boolean or throws an exception depending on the state of the channel.

On channel closed

If `channel.isClosedForSend` is `true`, `offer()` will throw an exception of the type `ClosedSendChannelException`, as the following for example:

```
val channel = Channel<Int>(1)
channel.close()
channel.offer(10)
```

This will crash, indicating that the channel was closed:



On channel full

If `isFull` is `true`, `offer()` will simply return `false`:

```
val channel = Channel<Int>(1)
channel.send(1)
channel.offer(2) // false
```

On channel open and not full

If the channel is open and it's not full, `offer()` will add the element to the queue. This is the only way to add elements to a channel without it happening from a suspending computation:

```
val channel = Channel<Int>(1)
channel.offer(2) // true
channel.receive() // 2
```

The `send()` implementation in `AbstractSendChannel` tries to add the element to the queue using `offer()` first; that way, the sender will not suspend if it's not needed.

ReceiveChannel

We have already covered some of the basics of `ReceiveChannel` in the previous chapter. Let's complete our analysis by taking a look at a couple of functions not mentioned before.

Validating before reading

There are a couple of checks that can be made before trying to read from a `ReceiveChannel` to avoid exceptions or in general to improve the flow of our code.

isClosedForReceive

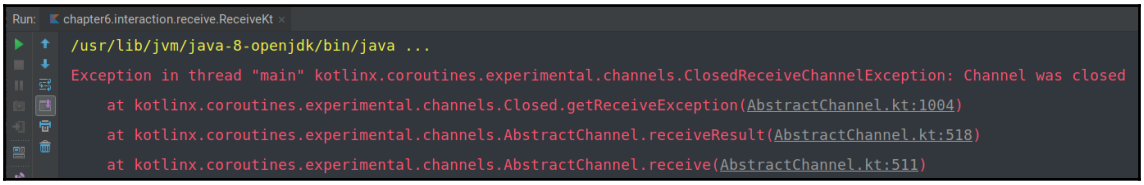
The first one would be to check the property `isClosedForReceive`, which indicates whether or not the channel has been closed for receiving, as the following for example:

```
val channel = Channel<Int>()
channel.isClosedForReceive // false
channel.close()
channel.isClosedForReceive // true
```

If receive is called in a channel that is closed, a `ClosedReceiveChannelException` will be thrown:

```
val channel = Channel<Int>()
channel.close()
channel.receive()
```

This code will simply crash:



isEmpty

The second validation would be to check whether there is something to receive. This can be done using the `isEmpty` property:

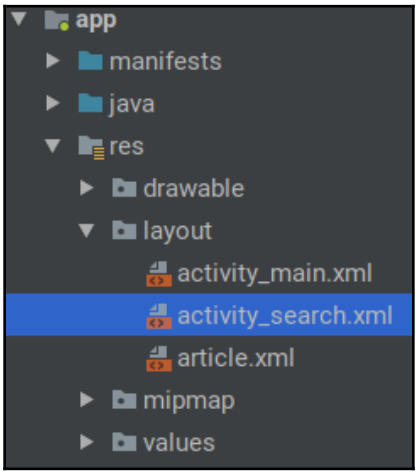
```
val channel = Channel<Int>(1)
channel.isEmpty // true
channel.send(10)
channel.isEmpty // false
```

Channels in action

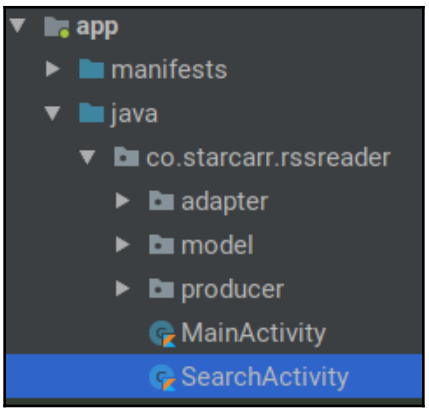
It's about time for us to actually use channels on our RSS Reader. In this chapter, we will add a new search feature to it, and will use channels to search all the outlets that we have configured at once.

Adding a search activity

First, let's copy the layout file `activity_main.xml` and name that copy `activity_search.xml`:



Once that is done, let's create a new Kotlin file, and name it `SearchActivity`:



Now we will create an activity inside that file. Initially, we want it to simply inflate the layout that we just created:

```
class SearchActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_search)
    }
}
```

Now we need to modify the layout so that we include a search field and a button to click to perform the search. To do so, let's modify `activity_search.xml` to add the two elements at the start of the constraint layout, as follows:

```
<android.support.constraint.ConstraintLayout ...>

    <EditText
        android:id="@+id/searchText"
        android:layout_width="250dp"
        android:layout_height="wrap_content"
        app:layout_constraintTop_toTopOf="parent" />
    <Button
        android:id="@+id/searchButton"
        android:text="Search"
        app:layout_constraintLeft_toRightOf="@id/searchText"
        android:layout_width="80dp"
        android:layout_height="50dp" />
```

Then we have to update the recycler view and the progress bar so that they appear below `searchText`. For simplicity, we also want to remove the progress bar, so let's update the layout of the `RecyclerView` and remove the XML corresponding to the progress bar:

```
<android.support.v7.widget.RecyclerView
    android:scrollbars="vertical"
    android:id="@+id/articles"
    android:layout_height="wrap_content"
    android:layout_width="match_parent"
    app:layout_constraintTop_toBottomOf="@id/searchText"
    />
```

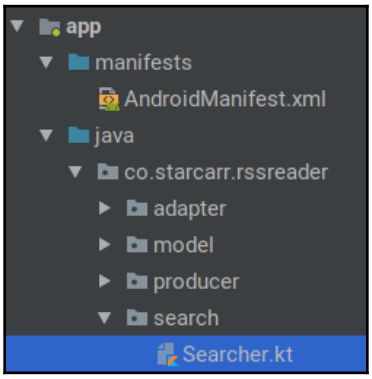
For now, we want the application to open directly into the search activity. To do so, let's update the `AndroidManifest.xml`, changing the default activity to `.SearchActivity`:

```
<application ...>
    <activity android:name=".SearchActivity">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />

            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
</application>
```

Make sure that only `SearchActivity` has `android.intent.action.MAIN`.

Adding the search function

The second step is to add the logic to do the searches properly. Let's create the package `co.starcarr.rssreader.search` and place the file `Searcher.kt` inside it:



Now we will create the class `Searcher` inside it, and include a function that will return a `ReceiveChannel<Article>`, given a `query`. Notice that we want to return a `ReceiveChannel` so that the caller doesn't use this channel to send information:

```
class Searcher() {
    fun search(query: String) : ReceiveChannel<Article> {
        val channel = Channel<Article>(150)
        return channel
    }
}
```

Implementing the collaborative search

Now we can do the actual implementation for the search. First we want to add a list of feeds to the `Searcher` class, and also a dispatcher and a document factory to retrieve the articles:

```
class Searcher() {

    val dispatcher = newFixedThreadPoolContext(3, "IO-Search")
    val factory = DocumentBuilderFactory.newInstance()

    val feeds = listOf(
            Feed("npr", "https://www.npr.org/rss/rss.php?id=1001"),
            Feed("cnn", "http://rss.cnn.com/rss/cnn_topstories.rss"),
            Feed("fox",
```

```
"http://feeds.foxnews.com/foxnews/latest?format=xml")
    )
    ...
}
```

Now we can add a `search()` function that will retrieve a feed, filter the articles containing the given query on the title or the description, and send them through a `SendChannel` received as a parameter. The initial implementation will be quite similar to the one we have used in the producer before:

```
private suspend fun search(
        feed: Feed,
        channel: SendChannel<Article>,
        query: String) {
    val builder = factory.newDocumentBuilder()
    val xml = builder.parse(feed.url)
    val news = xml.getElementsByTagName("channel").item(0)

    (0 until news.childNodes.length)
            .map { news.childNodes.item(it) }
            .filter { Node.ELEMENT_NODE == it.nodeType }
            .map { it as Element }
            .filter { "item" == it.tagName }
            .forEach {
                // TODO: Parse and filter
            }

}
```

But instead of mapping the content, we want to send it through a channel after filtering:

```
val title = it.getElementsByTagName("title")
        .item(0)
        .textContent
var summary = it.getElementsByTagName("description")
        .item(0)
        .textContent

if (title.contains(query) || summary.contains(query)) {
    if (summary.contains("<div")) {
        summary = summary.substring(0, summary.indexOf("<div"))
    }

    val article = Article(feed.name, title, summary)
    channel.send(article)
}
```

Connecting the search functions

Now we have a couple of `search()` functions: a public one that receives a query and returns a `ReceiveChannel<Article>`, and a private one that takes a feed, query, and a channel, and does the actual search. Our objective now is to connect the two of them. Let's go to the public `search()` function and update it so that it performs the search on each feed:

```
fun search(query: String) : ReceiveChannel<Article> {
    val channel = Channel<Article>(150)

    feeds.forEach { feed ->
        launch(dispatcher) {
            search(feed, channel, query)
        }
    }

    return channel
}
```

Now we have a public `search()` function that takes a query, and returns a channel that will be used to send results as they are found in the feeds – so the caller of `search()` can receive articles as they become available.

Updating ArticleAdapter

Before we can continue, we will need to update the logic that we implemented in `ArticleAdapter`. What we want to do is remove the code related to the `ArticleLoader`, since this implementation will not load on demand. First, let's update the constructor so that it doesn't take an `ArticleAdapter`:

```
class ArticleAdapter
    : RecyclerView.Adapter<ArticleAdapter.ViewHolder>() {

    ...
}
```

Now we can update the function `onBindViewHolder()` so that it doesn't try to load more articles when needed. The updated implementation looks as follows:

```
override fun onBindViewHolder(holder: ViewHolder, position: Int) {
    val article = articles[position]

    holder.feed.text = article.feed
```

```
        holder.title.text = article.title
        holder.summary.text = article.summary
    }
```

We will add a function to add articles incrementally to the adapter. Let's add this function at the end of the class:

```
    fun add(article: Article) {
        this.articles.add(article)
        notifyDataSetChanged()
    }
```

Below it, let's add a function to clear the adapter. This will come in handy because we need to clear the list between searches:

```
    fun clear() {
        this.articles.clear()
        notifyDataSetChanged()
    }
```

You will need to update `MainActivity` so that it doesn't pass in `this` when instantiating the adapter. After that, you will be able to use this adapter for the search feature.

Displaying the results

Now we need to have the UI perform the search and display the results. First, we need to add a `search()` function to be called when the button is clicked, and we also want to add the variables for the recycler view, the adapter, and the layout manager:

```
    class SearchActivity : AppCompatActivity()
    {
        private lateinit var articles: RecyclerView
        private lateinit var viewAdapter: ArticleAdapter
        private lateinit var viewManager: RecyclerView.LayoutManager

        ...

        private suspend fun search() {
            // TODO: Implement search function
        }
    }
```

To add the click listener, we just need to update the `onCreate()` function. Let's take this opportunity to also instantiate some of the added variables:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_search)

    viewManager = LinearLayoutManager(this)
    viewAdapter = ArticleAdapter()
    articles = findViewById<RecyclerView>(R.id.articles).apply {
        layoutManager = viewManager
        adapter = viewAdapter
    }

    findViewById<Button>(R.id.searchButton).setOnClickListener {
        viewAdapter.clear()
        launch {
            search()
        }
    }
}
```

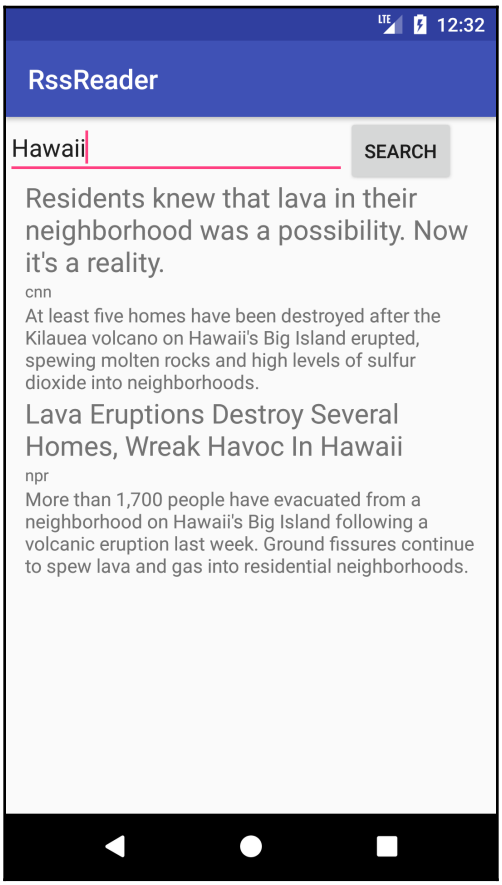
Now, going back to the `search()` function of the activity, we need to retrieve the current text from the `EditText`, and then call `Searcher.search()`, passing that text as the query. To do that, let's add the searcher to the instance of our activity:

```
class SearchActivity : AppCompatActivity()
{
    private val searcher = Searcher()
    ...
}
```

Now we can perform the actual search and add the items to the adapter as they arrive:

```
private suspend fun search() {
    val query = findViewById<EditText>(R.id.searchText)
            .text.toString()

    val channel = searcher.search(query)

    while (!channel.isClosedForReceive) {
        val article = channel.receive()

        launch(UI) {
            viewAdapter.add(article)
        }
    }
}
```

Now, whenever an article is found with the given query, the activity will receive it and add it to the adapter. Now you will be able to perform searches on all the feeds.



As an exercise, you can manually force delays in some of the feeds, and you will notice that as long as one of the feeds is fast, the experience of using the app will be acceptable.

Summary

This chapter was all about collaborative concurrency. We were able to cover many different topics, from practical and technical points of view. Let's recap the chapter to keep the knowledge fresh:

- We discussed some real-life examples of using channels to solve challenges related to collaborative concurrency.
- We learned that channels are a communication tool, allowing us to safely send messages between coroutines regardless of their thread.
- We talked about unbuffered channels. These are channels that will suspend `send()` until `receive()` is called for each element.
- We also covered three different types of buffered channels: `ConflatedChannel`, which keeps only the last element that was sent; `LinkedListChannel`, which will never suspend when `send()` is called because it can hold unlimited elements – or at least as many as possible in the available memory; and `ArrayChannel`, which will suspend on `send()` when the amount of elements within it reaches the size of its buffer.
- We covered many important properties and functions of `SendChannel`, such as `isClosedForSend` as `isFull`. We also talked about the suspending function `send()` and the non-suspending `offer()`. We talked about possible exceptions when trying to send or offer in a closed send channel.
- We also talked about `ReceiveChannel`, and we discussed the usage of the properties `isClosedForReceive` and `isEmpty`. On top of that, we talked about the possible exceptions when trying to receive from a closed receive channel.
- We put all of this into practice by adding collaborative concurrency to our RSS Reader, using channels and streams to add a search feature.

In the next chapter, we will talk about other advanced concepts such as thread confinement, actors, and mutual exclusions. All of them are really important approaches to concurrency.

With the knowledge of the next chapter, you will have a complete set of tools to pick and choose from whenever you are confronted with a concurrency challenge in Kotlin.

7
Thread Confinement, Actors, and Mutexes

Kotlin offers plenty of tools that can be used to write concurrent code safely. In the previous chapter, we talked about channels in the context of collaborative concurrency; in this chapter, we will dive into more tools that can be used to avoid sharing states between threads. We will start by talking again about atomicity violation, and then we will talk about ways of avoiding it. This chapter presents many topics in the context of atomicity violation, but it's important to understand that they can be used in other scenarios – for example, you can prevent race conditions using the methods covered in this chapter as well.

Let's take a look at the topics that we will cover in this chapter:

- Atomicity violation recap
- Thread confinement
- When and how to use thread confinement for safe concurrency
- Actors
- How actors work and when to use them
- Mutual exclusion
- How to use mutual exclusion
- Volatile variables
- When to use volatile variables
- Atomic variables

Atomicity violation

As mentioned in the first chapter, there is a type of concurrency error called atomicity violation. This type of error happens when the state of an object is modified concurrently without correct synchronization. The example that we saw in the first chapter was pretty simple: many coroutines modified the state of an object from different threads, and this resulted in some of those modifications being lost.

This type of error can happen with Kotlin too, but the language offers primitives that will help you to use the right design so that you avoid atomicity violation.

First, we need to truly understand what atomicity means and how violations can happen. Then we will be able to understand how to write our code to make it run atomically.

What atomicity means

In the context of software execution, an operation is atomic when it is single and indivisible. When talking about shared state, we are often talking about reading or writing to a single variable from many threads.

A challenge is raised because modifying the state of a variable is usually not atomic, since it usually consists of multiple steps like reading, modifying, and storing the updated value. During the execution of concurrent applications it's possible that a block of code that modifies a shared state does it by overlapping changes from other threads—for example, one thread can read the current value while another is modifying it, but before it being written. This situation will mean that one or more modifications to that shared state will be lost as it is overwritten.

Let's consider this simple function as an example:

```
private var counter = 0

fun increment() {
    counter ++
}
```

When executed sequentially, we are able to call `increment()` as many times as we want without having to worry about the value of `counter`. The value of counter will always correspond to the amount of times that `increment()` was called.

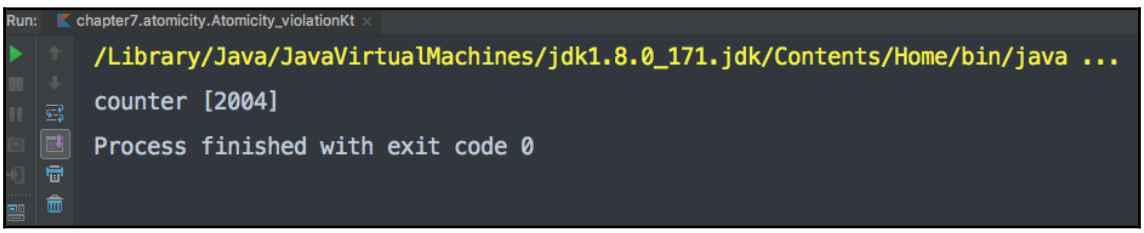
But when we add concurrency to the equation, a whole lot of things change under the hood. Let's start with this basic asynchronous function from `Chapter 1`, *Hello, Concurrent World!*:

```
var counter = 0

fun asyncIncrement(by: Int) = async(CommonPool) {
    for (i in 0 until by) {
        counter++
    }
}
```

It's increasing `counter` as many times as requested, using `CommonPool` as its `CoroutineContext`. We can call it from a main function like this – assuming that it runs in a device with more than one processing unit:

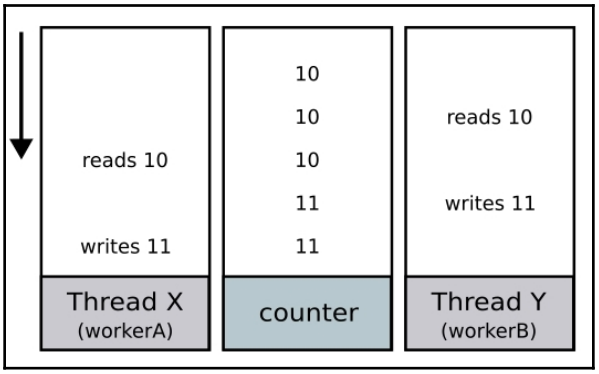
```
fun main(args: Array<String>) = runBlocking {
    val workerA = asyncIncrement(2000)
    val workerB = asyncIncrement(100)

    workerA.await()
    workerB.await()

    print("counter [$counter]")
}
```

You will notice that after execution, the value of `counter` will often be lower than `2100`:



This happens because the code `counter++` is not atomic. This single line of code can be broken into three operations: read, modify, and write, and because of how threads work, the write changes in one thread may not be visible for other threads as they read or modify the value, meaning that many threads can potentially increase `counter` to the same value.

What this means in practice is that many cycles of the `for` inside `asyncIncrement` could affect the value of the counter only once. Let's review the diagram explanation from `Chapter 1`, *Hello, Concurrent World!*:



In this diagram, we have a representation of a scenario in which two threads read the same value in a way in which the read and modify overlap, so one of the increments is lost.

To make any block of code atomic, we need to guarantee that all the memory accesses happening within the block cannot be executed concurrently. This can be accomplished in many different ways, and the best way to do it largely depends on the specifics of the situation.

Notice that in the previous example, to effectively reproduce the error you may have to run it multiple times. That's part of the problem with issues related to concurrency: they occur only under certain circumstances that may be difficult to identify or reproduce.

Thread confinement

The challenges of atomicity violation can be solved in many ways. The first one tackles the problem in a simple fashion: since we know that the problem can only happen when a state is shared between different threads, we make sure that this never happens.

What is thread confinement?

Thread confinement, as its name suggests, means confining all the coroutines accessing a shared state so that they execute on a single thread. This means that the state is no longer shared between threads: only one thread will modify the state.

This solution is great if you know that having all the coroutines modifying the state in the same thread will not negatively affect the performance of your application.

Confining coroutines to a single thread

Let's modify the example at the beginning of this chapter so that `asyncIncrement()` runs confined to a single thread:

```
var counter = 0
val context = newSingleThreadContext("counter")

fun asyncIncrement(by: Int) = async(context) {
    for (i in 0 until by) {
        counter++
    }
}
```

Now, regardless of how many times `asyncIncrement()` is called, it will run on a single thread, meaning that all the changes made to counter will be sequential.

Actors

Thread confinement works fine for the scenario mentioned previously, but say that we need a way to scale it for scenarios in which we need to modify the shared state from many different parts of the app, or if we want more flexibility on our atomic block. For more complicated scenarios, we can build upon the idea of thread confinement and improve our solution by using a concurrency primitive that we saw before: channels. By mixing both of them we can create an actor.

What is an actor?

Actors are a combination of two powerful tools: we can confine the accesses of a state to a single thread and allow other threads to request modifications to the state using channels. This way, we have not only a safe way to update the value but also a powerful communication mechanism to do it.

Creating an actor

Say that we need to modify a counter safely from many different threads. First, let's create a new class. In that class, we can have a private counter and a private single-thread dispatcher, with a function to retrieve the current value of the counter:

```
private var counter = 0
private val context = newSingleThreadContext("counterActor")

fun getCounter() = counter
```

Now that we have encapsulated the value of the counter, we simply need to add an actor that will increase the value upon each received message:

```
val actorCounter = actor<Void?>(context) {
    for (msg in channel) {
        counter++
    }
}
```

Because we don't actually use the message being sent, we can simply set the type of the actor to `Void?`, so that the caller can send `null`. Now we can update our main function to use this actor:

```
fun main(args: Array<String>) = runBlocking {
    val workerA = asyncIncrement(2000)
    val workerB = asyncIncrement(100)

    workerA.await()
    workerB.await()

    print("counter [${getCounter()}]")
}
fun asyncIncrement(by: Int) = async(CommonPool) {
    for (i in 0 until by) {
        actorCounter.send(null)
    }
}
```

Notice that `asyncIncement()` is using `CommonPool` as its context.

Using actors to extend the functionality

The biggest advantage of using actors is that channels allow for more flexibility while maintaining the whole coroutine atomic. We can use this messages to expand the functionality of the actor. For example, let's make it so that we can increase or decrease the counter using the actor. First, we can add an `enum` with both options:

```
enum class Action {
    INCREASE,
    DECREASE
}
```

And then we can update the actor and its coroutine to map these actions to operations:

```
var actorCounter = actor<Action>(context) {
    for (msg in channel) {
        when(msg) {
            Action.INCREASE -> counter++
            Action.DECREASE -> counter--
        }
    }
}
```

And from then on, we can simply call send with the action that we want. For the sake of completion, let's add an `asyncDecrement()` function:

```
fun asyncDecrement(by: Int) = async {
    for (i in 0 until by) {
        actorCounter.send(Action.DECREASE)
    }
}
```

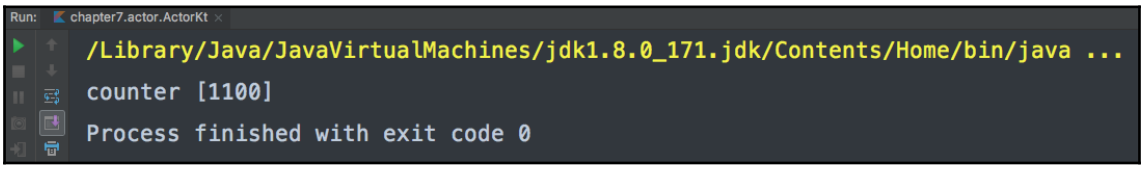
And we also need to update `asyncIncrement()` to send the action:

```
fun asyncIncrement(by: Int) = async {
    for (i in 0 until by) {
        actorCounter.send(Action.INCREASE)
    }
}
```

Now we can update our main function to use it:

```
fun main(args: Array<String>) = runBlocking {
    val workerA = asyncIncrement(2000)
    val workerB = asyncIncrement(100)
    val workerC = asyncDecrement(1000)

    workerA.await()
    workerB.await()
    workerC.await()

    print("counter [${getCounter()}]")
}
```

This will work just fine:



Notice that this implementation has an interesting flow. The main thread will use `CommonPool` to create coroutines that increase and decrease the counter concurrently, and those coroutines will do so by sending our actor messages with the action. Because the actor is a coroutine confined to a specific thread, all the modifications will be atomic, so we know that the value will always be correct.

While in this example we still called the actor from a class in the same package, a big advantage using an actor is that you can call it from many places as long as you have an instance of the actor you want to communicate with.

More on actor interaction

From a client perspective, an actor will simply be a send channel. But from an implementation standpoint, we are allowed to tinker with how our actor behaves.

Buffered actors

An actor can be buffered, the same as any other send channel. By default, all actors are unbuffered: they suspend the caller on `send()` until the message has been received. To create a buffered actor you need to pass the `capacity` parameter to the builder:

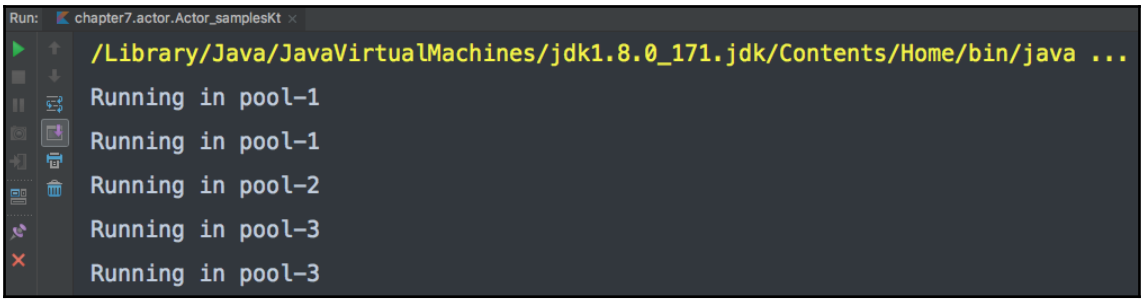
```
fun main(args: Array<String>) {
    val bufferedPrinter = actor<String>(capacity = 10) {
       for (msg in channel) {
           println(msg)
       }
    }

    bufferedPrinter.send("hello")
    bufferedPrinter.send("world")

    bufferedPrinter.close()
}
```

Actor with CoroutineContext

Like we did to solve the `counter` example, you can pass a `CoroutineContext` when building an actor. The suspending lambda of the actor will be executed in the given context. For example, you can create an actor that processes its messages in a pool of threads:

```
val dispatcher = newFixedThreadPoolContext(3, "pool")
val actor = actor<String>(dispatcher) {
    for (msg in channel) {
        println("Running in ${Thread.currentThread().name}")
    }
}

for (i in 1..10) {
    actor.send("a")
}
```

The suspending lambda will be executed accordingly:



CoroutineStart

By default, actors will be started as soon as they are created. You can pass a `CoroutineStart` to change this behavior according to what you need. For example:

```
val actor = actor<String>(start = CoroutineStart.LAZY) {
    for (msg in channel) {
        println(msg)
    }
}
```

In this case, the actor will be started when it first receives a message.

You can use all the other values of `CoroutineStart` as well, like `DEFAULT`, `ATOMIC`, and `UNDISPATCHED`.

Mutual exclusions

So far, we have avoided atomicity violation by guaranteeing that all memory access in a block of code happens in a single thread. But there's yet another way for us to avoid two blocks of code from being executed concurrently: mutual exclusions.

Understanding mutual exclusions

We are looking for ways to synchronize a block of code so that it's not executed concurrently, thus eliminating the risk of atomicity violation. Mutual exclusion refers to a synchronization mechanism that guarantees that only one coroutine can execute a block of code at a time.

The most important characteristic of Kotlin's mutexes is that they are not blocking: the coroutines waiting to be executed will be suspended until they can acquire the lock and execute the block of code. Nevertheless, it's possible to lock a mutex without being in a suspending function.

If you are familiar with Java, you can think of a mutex as a non-blocking `synchronized`.

Creating mutexes

To create a mutex, you only need to create an instance of the `Mutex` class:

```
var mutex = Mutex()
```

Once you have a mutex, you can use the suspending extension function `withLock()` that executes a lambda using a lock, shown as follows:

```
fun asyncIncrement(by: Int) = async {
    for (i in 0 until by) {
        mutex.withLock {
            counter++
        }
    }
}
```

Here we don't set any context when calling `async()`, because we don't really care about the thread that is calling the function – it will work regardless.

This will guarantee that all the increments to counter are synchronized by allowing only one coroutine to hold the lock at a time, and suspending any other coroutine trying to get it. So, none of the increments in `counter` will be lost regardless of how many times it's called, shown as follows:

Interacting with mutual exclusions

More often than not, using `withLock()` will be enough for your needs. But in case you need more control of the locking and unlocking, you can use the suspending function `lock()` and the regular—as in non-suspending—function `unlock()`:

```
val mutex = Mutex()

mutex.lock() // this will suspend if the lock has been taken already
print("I am now an atomic block")
mutex.unlock() // this is not suspending
```

As a matter of fact, the implementation of `withLock()` is quite close to that. This is the current implementation of it:

```
lock(owner)
try {
    return action()
} finally {
    unlock(owner)
}
```

You can use the property `isLocked` to validate whether the mutex is currently locked:

```
val mutex = Mutex()

mutex.lock()
mutex.isLocked // true
mutex.unlock()
```

You can also use `tryLock()`, which returns a `Boolean` indicating whether it was possible to lock the mutex:

```
val mutex = Mutex()

mutex.lock()
val lockedByMe = mutex.tryLock() // false
mutex.unlock()
```

It will return `true` if `tryLock()` was able to lock the mutex, and `false` otherwise:

```
val mutex = Mutex()

val lockedByMe = mutex.tryLock() // true
mutex.unlock()
```

Notice that this last example doesn't need to be executed in a suspending function, given that `tryLock()` and `unlock()` are not suspending.

Volatile variables

Allow me to start by clarifying that volatile variables will not solve problems like the thread-safe counter that we are trying to implement. Nevertheless, volatile variables can be used in some scenarios as a simple solution when we need information to be shared among threads.

Thread cache

On the JVM, each thread may contain a cached copy of any non-volatile variable. This cache is not expected to be in sync with the actual value of the variable at all times. So changing a shared state in a thread will not be visible in other threads until the cache is updated.

@Volatile

In order to force changes in a variable to be immediately visible to other threads, we can use the annotation `@Volatile`, a in the following example:

```
@Volatile
var shutdownRequested = false
```

This will guarantee visibility of changes for other threads as soon as the value is changed. This means that if thread *X* modifies the value of `shutdownRequested`, thread *Y* will be able to see the change immediately.

Notice that `@Volatile` is only available in Kotlin/JVM. For other platforms, it's not available since it relies on the JVM to guarantee volatility.

Why @Volatile doesn't solve thread-safe counters

There is a common misconception about what volatile variables guarantee. So to understand when it's useful, we need to go back to considering the thread-safe counter from previous examples. During previous explanations, I indicated that the atomicity violations in the example happen because two threads read the value of a variable really close in time, which in turn causes some of the changes to be lost. But there's a little more to it; there can be two reasons why both threads read the same value:

- When the reads of a thread happen during the reading or modifying of another thread: both threads start with the same data and make the same increment. Both change the counter from *X* to *Y*, so one increment is lost.
- When the read of one thread happens after the modifying of another thread, but the cache local to the thread has not been updated: a thread can read the value of the counter as *X* even after another thread set it to *Y* because the local cache was not updated in time. The end result is similar: the second thread will increase the value of the counter, but since it started with an obsolete value, it will only override the change already made.

Notice that even though the cause is a little bit different, both of these situations have the same result: data changes being lost due to lack of synchronization.

As you can probably guess, volatile variables will offer protection in the second case, by making sure that any read of a state will always hold the most recent value. But they will not guarantee protection against the first case, because two threads can still read the current value close enough for them to try to perform the same increment, resulting in data loss.

When to use @Volatile

There are still some scenarios in which using volatile variables can help us write better code. These scenarios are based on two premises, both of which need to be true – if either of them is false then a different solution needs to be implemented:

- A change in the value of the variable doesn't depend on its current state
- The volatile variable doesn't depend on any other variables and other variables don't depend on it

The first premise helps to rule out scenarios like the thread-safe counter: because the change in the state is not atomic, future values can't safely depend on the current one.

The second premise, on the other hand, helps us to avoid the creation of inconsistent states because of dependencies from or to volatile variables. This example will not be safe because the second premise is not being followed:

```
class Something {

    @Volatile
    private var type = 0
    private var title = ""
    fun setTitle(newTitle: String) {
        when(type) {
            0 -> title = newTitle
            else -> throw Exception("Invalid State")
        }
    }
}
```

In this case, trying to set `title` should throw an exception if `type`—which is volatile—is different from zero. The problem here is that `type` can be zero when a thread enters the `when` clause, but a different thread can set it to something different by the time `title` is modified. If that happens, then we will have an inconsistent state in which `type` is not zero, yet a `title` was set.

On the other hand, if both premises mentioned before are followed, we can proceed with using volatile variables. A common example of using them is to contain flags. Consider the following class:

```
class DataProcessor {

    @Volatile
    private var shutdownRequested = false
```

```
        fun shutdown() {
            shutdownRequested = true
        }

        fun process() {
            while (!shutdownRequested) {
                // process away
            }
        }
    }
```

In this example both premises are valid:

- The modification of `shutdownRequested` made in `shutdown()` doesn't depend on the current state of the variable itself – it's always set to `true`
- No other variable depends on `shutdownRequested`, and it doesn't depend on any other variable either

The advantage of this approach is that it allows any thread to request shutdown and it will be visible for all the threads immediately.

Atomic data structures

During this chapter, we have covered how to write our own atomic blocks of code, but there's yet another topic that we need to mention regarding atomicity: atomic data structures. These are data structures that offer atomic operations out of the box.

Currently these atomic data structures are provided by the JVM, not Kotlin's standard library. So they may not be available if your code is for JS, Kotlin/Native, or multiplatform.

For example, using an atomic integer looks like the following:

```
val counter = AtomicInteger()
counter.incrementAndGet()
```

Because the implementation of `incrementAndGet()` is atomic, we can use it easily to implement a thread-safe counter:

```
var counter = AtomicInteger()

fun asyncIncrement(by: Int) = async {
    for (i in 0 until by) {
        counter.incrementAndGet()
    }
}
```

We can call it from main like in our original implementation:

```
fun main(args: Array<String>) = runBlocking {
    val workerA = asyncIncrement(2000)
    val workerB = asyncIncrement(100)

    workerA.await()
    workerB.await()

    print("counter [$counter]")
}
```

It will always do all the increments to `counter` correctly.

Notice that there are many different atomic classes that you can use for simple scenarios, such as `AtomicBoolean`, `AtomicLong`, or `AtomicIntegerArray`. While all of them can be used for problems like the one in the example of the counter, they tend not to scale well when the shared state is more complex.

Actors in action

Now that we understand the synchronization problem completely, we will do a real-life implementation. The objective is to add a label to our RSS reader so that it shows the amount of news that were found with the given search parameters.

Because we have been modifying our app to be reactive and to display results as they come from different sources – which are obtained and parsed from different coroutines – we need to guarantee that the counter of the amount of news is accurate at all times.

Adding the label to the UI

The first step is to update the UI so that we have a TextView to display the amount of results. Let's go to `activity_search.xml` and add the following element `searchButton`:

```
<Button android:id="@+id/searchButton" ... />

<TextView android:id="@+id/results"
    app:layout_constraintTop_toBottomOf="@id/searchText"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

<android.support.v7.widget.RecyclerView ... />
```

Then, we need to update the `RecyclerView` so that it appears below the newly created `results`:

```
<android.support.v7.widget.RecyclerView
    android:id="@+id/articles"
    android:scrollbars="vertical"
    android:layout_height="wrap_content"
    android:layout_width="match_parent"
    app:layout_constraintTop_toBottomOf="@id/results" />
```

Creating an actor to use as a counter

The next step will be to create an actor that will encapsulate the counter with the amount of news that has been found for the given search. Because this actor can be accessed from many areas, we will write it to be a singleton.

Let's create the Kotlin file `ResultsCounter.kt` in the package `co.starcarr.rssreader.search` and then create the singleton `ResultsCounter` inside it with a couple of properties:

```
object ResultsCounter {
    private val context = newSingleThreadContext("counter")
    private var counter = 0
}
```

First we have `context`, which is a `CoroutineContext` that consists of a single thread. The coroutine of the actor will ultimately run with that context, so we will avoid atomicity violations by having all the increments run confined there. Let's now add the actor as a private property:

```
private val actor = actor<Void?>(context) {
    for (msg in channel) {
        counter++
    }
}
```

Similar to our examples before, this implementation will initially be a coroutine increasing the counter. But we have made it private so that we can offer a method in `ResultsCounter` for the clients of this API to increase it easily. Let's add this function now:

```
suspend fun increment() = actor.send(null)
```

This function can be called from any coroutine, and will keep the API clean and the implementation encapsulated. If tomorrow we decide to change to a mutex, we could still implement it without having to break the API.

Increasing the counter as results are loaded

Now we simply need to call the method to increase the function when appropriate. Let's go to the `Searcher` class of the package `co.starcarr.rssreader.search` and add the call just after we send the processed article:

```
...
val article = Article(feed.name, title, summary)
channel.send(article)

// Increment the singleton with the counter
ResultsCounter.increment()
...
```

Adding a channel so that the UI reacts to updates

To make the implementation more interesting, let's make it so that the UI can listen to increments in the counter – and update the label – by subscribing to a channel. First, let's add a channel property to `ResultsCounter`:

```
object ResultsCounter {
    private val context = newSingleThreadContext("counter")
    private var counter = 0
    private val notifications = Channel<Int>(Channel.CONFLATED)
    ...
}
```

We are making this channel private so that we can retrieve it as a `ReceiveChannel`, preventing the clients of the API from trying to send messages using it. We are making the channel conflated because we don't care if some notifications are lost: the UI only cares about the most recent value for the counter.

Notice that the channel is of type `Int`; this will allow us to simply send the new value on each change. Now let's add a function to expose the channel to the UI:

```
fun getNotificationChannel() : ReceiveChannel<Int> = notifications
```

Sending the updated value through the channel

The missing part in `ResultsCounter` is to actually send the updated value using the channel. Let's update the actor to do so:

```
private val actor = actor<Void?>(context) {
    for (msg in channel) {
        counter++
        notifications.send(counter)
    }
}
```

Updating the UI on changes

Now we only need to connect the UI to this channel. Let's go to the activity `SearchActivity` and add a function to monitor the channel and update the UI on changes:

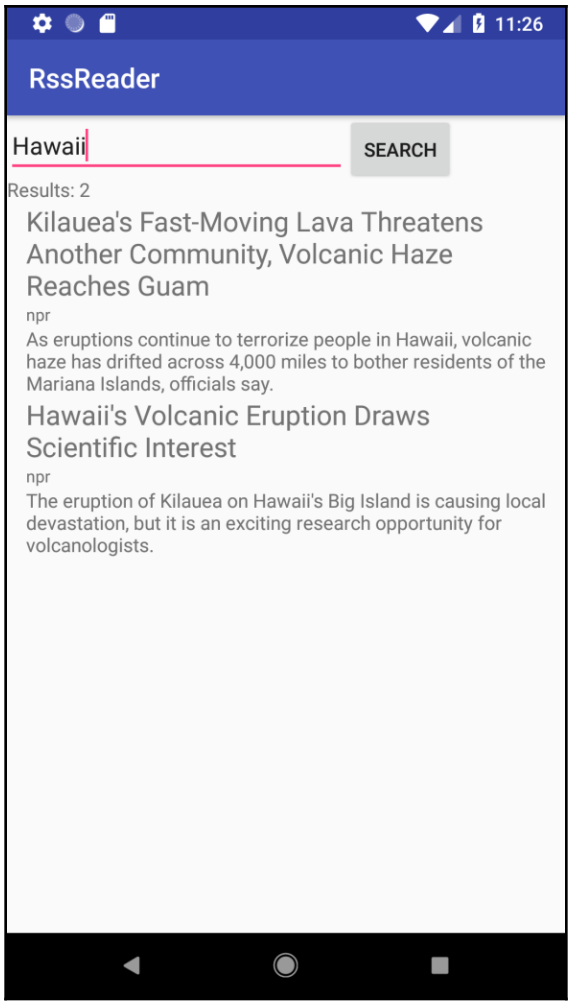
```
private suspend fun updateCounter() {
    val notifications = ResultsCounter.getNotificationChannel()
    val results = findViewById<TextView>(R.id.results)

    while (!notifications.isClosedForReceive) {
        val newAmount = notifications.receive()

        withContext(UI) {
            results.text = "Results: $newAmount"
        }
    }
}
```

This code is simply getting the channel from the singleton and entering a loop for as long as the channel is open. Upon receipt of a new amount, it will simply set the new value to the `TextView` on the UI thread. Finally, we need to update `onCreate()` to make sure that it calls `updateCounter()`, as follows:

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

    launch {
        updateCounter()
    }
}
```

Testing the implementation

If we execute the app now, we will see the counter increase at the same rate as the results appear:



But if you do multiple searches without closing the app, you will notice an error in our implementation: the counter can only increase, and will never be reset to zero.

Extending the actor to allow for resetting the counter

Because actors are so flexible, we will be able to add the reset functionality easily. Let's go back to `ResultsCounter` and create an `enum` for the action to be performed:

```
enum class Action {
    INCREASE,
    RESET
}
```

Then we can update the definition of the actor and the implementation of the coroutine to use an `Action` channel:

```
private val actor = actor<Action>(context) {
    for (msg in channel) {
        when (msg) {
            Action.INCREASE -> counter++
            Action.RESET -> counter = 0
        }
        notifications.send(counter)
    }
}
```

Now we are receiving an `Action`, and either increasing or resetting the counter accordingly. The next step is to update our public API so that `increase()` uses the channel correctly:

```
suspend fun increment() = actor.send(Action.INCREASE)
```

And we will now also need our API to allow the counter to be reset. Let's create a function for this:

```
suspend fun reset() = actor.send(Action.RESET)
```

Resetting the counter upon new searches

Now we just need to reset the counter whenever a new search is started. This could be done in the `Searcher` class before starting a search, or when the button on the UI is clicked. Because our counter is thread-safe, either of those options should work fine.

Let's reset the counter when the search button is clicked in `SearchActivity`, as part of the click listener already being set during `onCreate()`:

```
findViewById<Button>(R.id.searchButton).setOnClickListener {
    viewAdapter.clear()
    launch {
        ResultsCounter.reset()
        search()
    }
}
```

Now, upon further testing, you will find that the counter is reset before each search.

Summary

This chapter has covered some really important topics on how to use Kotlin to prevent common pitfalls of concurrent programming. All the different tools covered in this chapter will come in handy in different circumstances when you are writing concurrent applications. In the same way an actor is a mix of a coroutine with a channel, you can mix many of the solutions covered here to create an implementation that meets your requirements.

As mentioned at the beginning of this chapter, you shouldn't limit these tools to atomicity violation. They will help you to tackle other concurrency challenges as well.

Let's recap this chapter and the more important topics:

- Having a shared state can be a problem in concurrent code. A thread's cache and the atomicity of memory access can cause modifications coming from different threads to be lost. It can also cause the state to become inconsistent.
- There are two main ways to avoid such problems: guaranteeing that only one thread interacts with the state—hence making it shared only for reading but not for writing – and using locks to make a block of code atomic, forcing the synchronization of all the threads trying to run that block of code.
- We can use a `CoroutineContext` with a dispatcher of only one thread to force the execution of a coroutine to happen in a single thread. This is called thread confinement.

- An actor is the pairing of a coroutine with a send channel. You can confine the actor to a single thread to build a more robust synchronization mechanism that is based on messages. You can request a change by sending a message from any thread you want, but the change will be executed on a specific thread.
- While actors are particularly good when paired with thread confinement of the coroutine, you can specify any `CoroutineContext` to be used by the actor, so you can have the actor execute in a pool of threads, for example.
- Since actors are coroutines, they can be started in different ways. For example, you can have actors that are started lazily.
- To synchronize coroutines using a lock, you can use a mutex. Doing so, you will be able to suspend coroutines while they wait to be able to perform a synchronized operation.
- JVM offers volatile variables, which are variables that will not be put in the cache of the thread. This can help us solve basic concurrency challenges if the variable being shared among threads has two characteristics: when modified, the new value doesn't depend on the previous one; and the volatile variable's state will not depend on, or affect, other properties.
- There are also atomic variables, which are objects that offer atomic implementations for common operations like incrementing and decrementing the value of the variable. Currently they are available only for the JVM.
- Atomic variables are useful in simple cases, but will be difficult to scale if the state being shared is more than just one variable.
- We also used a real-life scenario to do an actual implementation of an actor. We wrote a counter for the news found during a search backed by an actor, and extended the implementation so that the UI could react to changes in the counter by listening to a channel.
- We put into practice an important principle: information hiding. We hid the implementation of our counter so that in the future we could change it to use mutexes, atomic variables, or thread confinement without actors.

The next chapter will bring some important topics to the table. We will discuss how to write tests that can validate our concurrent implementations and give us some peace of mind about our code. We will discuss good practices when testing concurrent code, and also how to debug coroutines.

8
Testing and Debugging Concurrent Code

One of the biggest challenges with concurrency is that bugs are often found late in the development process. Often, a concurrency error will first be seen in a production environment, when Murphy's law becomes a reality and all the things that can go wrong actually do go wrong.

This is because many of the bugs related to concurrency will happen during edge cases that either are thought of as impossible, or are so unlikely that they didn't even cross the minds of the developer or the code reviewers. This chapter aims to provide you with some advice on how to write tests that can help you to identify the scenarios you aren't prepared to handle, as well as some information on how to successfully write logs and debug your coroutines.

The topics in this chapter are interleaved with general advice that may not be directly related to coroutines. Writing proper tests and knowing how to debug are important and will help you build stable applications, and advice for proper testing is always relevant.

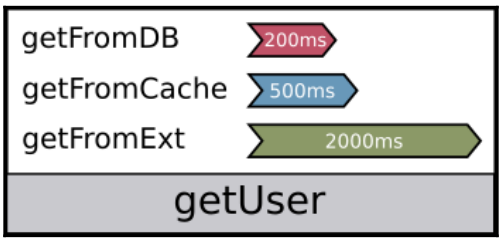
In this chapter, we will cover these topics:

- Writing tests for concurrent code
- Looking for edge cases
- Enabling debug settings for coroutines
- Using logs and breakpoints to debug coroutines
- General advice for testing and debugging

Testing concurrent code

When it comes to tests, the important thing is not to just have them, but to have the correct ones. If we are thinking specifically about concurrency, there are two principles that I consider the foundations of correct testing. Let's turn our minds to a basic example, as a way to discuss those principles.

Consider a simple application that takes the identifier of a user and needs to retrieve, organize, and then return a defined set of information for that user. Now, some of that information will be readily available in the database of that application. Other information may need to come from a cache shared among instances of the application. And finally, some specific information will be retrieved from a different application, which exposes it through a REST API. Here is a simple image detailing the usual behavior for the functionality:



This diagram represents **getUser** retrieving the information concurrently from three places, along with the average time that it will take it to receive the response from each of them.

Throwing away assumptions

An important detail in the preceding diagram is that it indicates the average response time for each of the data sources. As expected, going to the database is the fastest, with the cache being a close second. Retrieving information from an external system is the slowest of them.

These types of statistics will become a headache for many of you in the future. That is, if you allow them to become assumptions of how the world works. A lot of people, when faced with the previous example and having to write a concurrent solution, will write the application around the premise that the information coming from the DB will always be ready by the time the cache responds, and that both of them will be ready when the external application provides us with the needed information.

And not only will the application be written around those assumptions, the tests for it will never challenge them. And the day that the DB takes longer than the external application, maybe because of changes in the infrastructure, the application will either stop working or behave unexpectedly.

This is the reason why the first and most important principle is to *throw away assumptions*. If not when coding the application – because you are being told that it's *impossible* for the DB to take longer than an external system – at least do it when writing the tests. Have tests that challenge any assumptions in order to guarantee that it will not be a spectacular fail when it takes longer—because believe me, it will.

Focus on the forest, not the trees

The second principle, *focus on the forest, not the trees*, is perhaps so basic that it can be easily overlooked. I am sure that, for the example given in this section, a good developer would write the necessary unit tests to cover each of the next scenarios:

- Successfully retrieving information from the DB
- Successfully retrieving information from the cache
- Successfully retrieving information from the external application
- Incomplete or missing information on the DB
- Incomplete or missing information on the cache
- Incomplete or missing information on the external application

These tests are all good for validating the basic units of the application but, when testing for concurrency, they will be just trees in the forest.

What we really need to test is what happens when the DB takes longer than the external application, and comes with incomplete or missing information. We need to test what happens if only one of them brings back information. How much resilience is expected? Should the application return what was found, throw an error, or do both? These questions are important because they really put the focus on the application being written in a way that will guarantee the expected behavior when one or more of the concurrent tasks moves away from the happy/expected path.

Writing Functional Tests

Commonly, these types of tests are called Functional Tests. They differ from Unit Tests in that they don't test the small units of code—like functions—but the overall behavior of a functionality. Functional Tests are needed for concurrent code because they exercise the functionality as a whole, allowing us to create tests that depict the complexity that comes with having the application do operations asynchronously.

A concurrent application without Functional Tests will be fragile, so any change to its code may unleash a chaos of race conditions, atomicity violations, deadlocks, and so on.

More advice on tests

I consider this chapter particularly important. That is because, in my experience, the only way to guarantee the stability of an application is to have the right tests for it. Here is some advice that isn't necessarily tied to concurrency, but is important to keep in mind anyway:

- The fix for a bug must be accompanied with a test covering the scenario. This is the only way to keep regressions under control; if you don't add a test for the fix you just made, then eventually the code will be modified in a way that will bring back the bug.
- Always consider how a concurrency bug may affect other parts of the application. Often, a bug will be reported for a specific feature, but because we implement many features in a similar way, that same bug may exist in other places. And if you have even a slight suspicion that it could be happening elsewhere, add a test to validate if it exists—if it doesn't, you have added a test that will prevent it from happening in the future.
- Don't test all the permutations for your concurrent tasks. The objective is not to cover every single scenario, but to find those that challenge assumptions while adding value.
- Talk about resiliency before doing the implementation, and always test for resiliency. I have seen many projects where no one asked what the expected behavior should be in *gray* scenarios, like only partial information being available. It's important to consider those ahead of time because exception handling is something that you don't want to design after the fact.
- To find edge cases, use the branch analysis from your coverage report. This may not always be useful when testing concurrent code, but it's worth to try. The branch report will let you know if your tests are following the same scenarios always—for example, if, for all your tests, the same path is being executed.

- Learn when to write unit tests and when to write Functional Tests. Functional tests will often require more effort, so it's important to do them when they really add value.
- Wire your dependencies using interfaces—that will ease the job of mocking them to replicate complex scenarios for Functional Tests.

Writing the tests

Let's write a simple and flawed application that can help us, as an example of how to write Functional Tests for concurrent applications. In practice, writing these tests will require more work, but the principles should still apply.

Creating a flawed UserManager

Using interfaces to define the dependencies of our code will allow us to provide mocks during the tests. Let's start by defining a basic `DataSource` interface that contains the methods that will be used concurrently:

```
interface DataSource {
    fun getNameAsync(id: Int): Deferred<String>
    fun getAgeAsync(id: Int): Deferred<Int>
    fun getProfessionAsync(id: Int): Deferred<String>
}
```

Now, let's define a data class that will be the return type of the functionality we want to test. In this case, we just need a simple class:

```
data class User(
        val name: String,
        val age: Int,
        val profession: String
)
```

And finally, let's add `UserManager`, which has the functionality that we want to test. This class contains a method that calls `datasource` in order to retrieve all the different chunks of information, then it compiles that together and returns `User`:

```
class UserManager(private val datasource: Datasource) {
    suspend fun getUser(id: Int): User {
        val name = datasource.getNameAsync(id)
        val age = datasource.getAgeAsync(id)
        val profession = datasource.getProfessionAsync(id)
```

```
            // Wait for profession, since it "will" take longer
            profession.await()

            return User(
                name.getCompleted(),
                age.getCompleted(),
                profession.getCompleted()
            )
        }
    }
```

In this first implementation, we have decided that we will only suspend while `profession` is being obtained. This decision is made because we *know* that profession will always take longer to be retrieved since it's coming from an external system. Now, let's add some tests to validate that it works.

If you follow **Test-Driven Development** (**TDD**), you would write the tests before writing the code. In this case, I did it in this order because it seemed to me that it would be more simple to understand what we are doing this way.

Adding the kotlin-test library

The first step is to add the required dependencies to our project. The `kotlin-test` library provides annotations and base implementations for tests and assertions. Let's go to the `build.gradle` file and append the lines in bold:

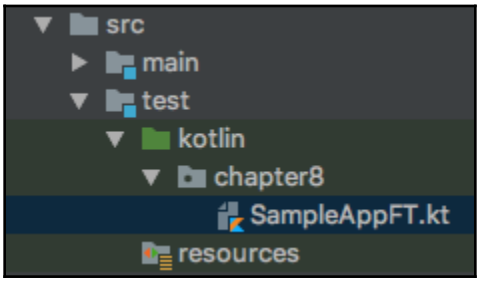
```
dependencies {
    compile "org.jetbrains.kotlin:kotlin-stdlib-jdk8:$kotlin_version"
    compile "org.jetbrains.kotlinx:kotlinx-coroutines-
core:$coroutines_version"

    // Test libraries
    testCompile "org.jetbrains.kotlin:kotlin-test"
    testCompile "org.jetbrains.kotlin:kotlin-test-junit"

}
```

Once the changes are applied, we will be able to start writing our Functional Tests.

We are using the standard testing library for Kotlin because I don't want the book to include third-party libraries unless needed. But be aware that there are other testing frameworks for Kotlin that may have features that adjust better to your project.

Adding a happy path test

Let's add a class that will contain our Functional Tests. We will call it `SampleAppFT` and put it in the `chapter8` package of the test module:



Inside it, let's start by creating a class with two empty tests. We will implement those in a bit:

```
class SampleAppFT {

    @Test
    fun testHappyPath() = runBlocking {
        // TODO: Implement a happy path test
    }

    @Test
    fun testOppositeOrder() = runBlocking {
        // TODO: Implement a test with an
        // unexpected order of retrieval
    }
}
```

Next to the `SampleAppFT` class, let's create a mock implementation of `DataSource` that fits the expected behavior when retrieving the information. In it, getting data from the database will be faster than the cache, which will be faster still than the external application:

```
// Mock a datasource that retrieves the data in the expected order
class MockDataSource: DataSource {
    // Mock getting the name from the database
    override fun getNameAsync(id: Int) = async {
        delay(200)
        "Susan Calvin"
    }

    // Mock getting the age from the cache
    override fun getAgeAsync(id: Int) = async {
```

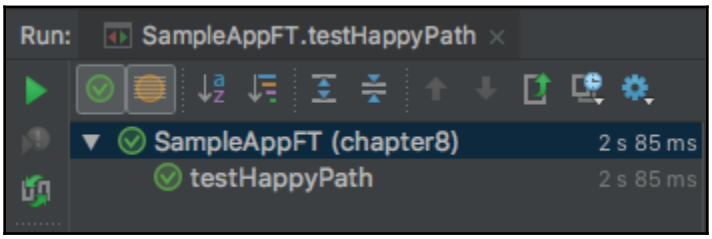
```
            delay(500)
            Calendar.getInstance().get(Calendar.YEAR) - 1982
        }

        // Mock getting the profession from an external system
        override fun getProfessionAsync(id: Int) = async {
            delay(2000)
            "Robopsychologist"
        }
    }
```

With this mocked `DataSource`, we are able to complete our happy path test:

```
@Test
fun testHappyPath() = runBlocking {
    val manager = UserManager(MockDataSource())

    val user = manager.getUser(10)
    assertTrue { user.name == "Susan Calvin" }
    assertTrue { user.age == Calendar.getInstance().get(Calendar.YEAR) -
1982 }
    assertTrue { user.profession == "Robopsychologist" }
}
```

`assertTrue()` is from the `kotlin.test` package.

By running this test—right-click on its name and select Run `testHappyPath()`—we will be able to assert that, under the expected timing, our `UserManager` behaves well:



Testing for an edge case

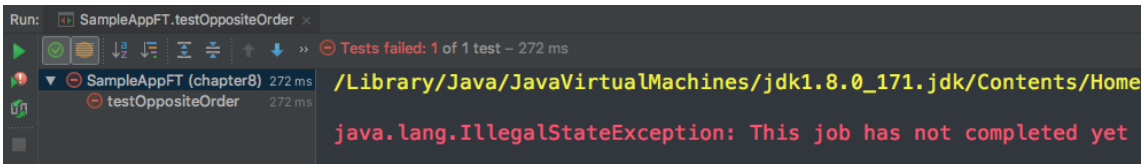
So now it's time to throw away our assumptions and write a mock `DataSource` that will retrieve from an external application faster than from the DB. Let's add another implementation of `Datasource` in the same `SampleAppFT`. We will call it `MockSlowDbDataSource` for lack of a better name:

```
// Mock a datasource that retrieves the data in a different order
class MockSlowDbDataSource: DataSource {
    // Mock getting the name from the database
    override fun getNameAsync(id: Int) = async {
        delay(1000)
        "Susan Calvin"
    }

    // Mock getting the age from the cache
    override fun getAgeAsync(id: Int) = async {
        delay(500)
        Calendar.getInstance().get(Calendar.YEAR) - 1982
    }

    // Mock getting the profession from an external system
    override fun getProfessionAsync(id: Int) = async {
        delay(200)
        "Robopsychologist"
    }
}
```

We will complete our second test using it:

```
@Test
fun testOppositeOrder() = runBlocking {
    val manager = UserManager(MockSlowDbDataSource())

    val user = manager.getUser(10)
    assertTrue { user.name == "Susan Calvin" }
    assertTrue { user.age == Calendar.getInstance().get(Calendar.YEAR) -
1982 }
    assertTrue { user.profession == "Robopsychologist" }
}
```

If we run this test, we will make the application crash:



Identifying the issue

If we look at the stack of the exception, it indicates that the error happened when we tried to create `User`. The line in question is this:

```
return User(
        name.getCompleted(),
        age.getCompleted(),
        profession.getCompleted()
)
```

If we read the documentation of `getCompleted()` we will see that `IllegalStateException` is thrown because the job has not completed—this can be seen by reading the stack trace of the exception as well. The exception makes sense because we are running `getCompleted()` as soon as `profession` is ready, regardless of the readiness of `name`.

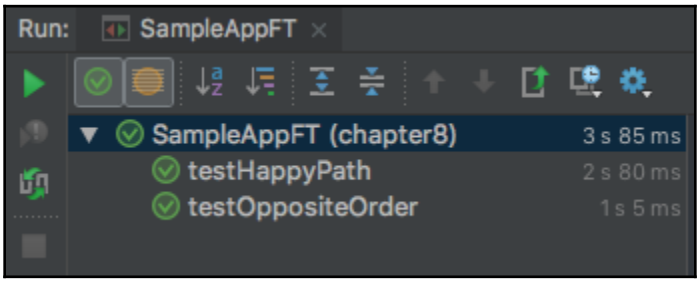
Fixing the crash

The obvious solution to this issue is not to assume that a value is ready, but to always wait for it to be. In this case, we simply need to change our implementation to stop using `getCompleted()` and use `await()` for all the jobs to complete:

```
suspend fun getUser(id: Int): User {
    val name = datasource.getNameAsync(id)
    val age = datasource.getAgeAsync(id)
    val profession = datasource.getProfessionAsync(id)

    // Wait for each of them, don't assume they are ready
    return User(
        name.await(),
        age.await(),
        profession.await()
    )
}
```

Retesting

Now that we have a fix, we just need to run both tests and validate that they are working. To do this, you can right-click in the test class and select Run `SampleAppFT`:



Debugging

Often, you will find yourself debugging errors that happen inside a coroutine. Even when you have the steps to recreate a bug, following the code and making sense of what's happening may require some hard debugging work. In this section, we will cover good practices to debug your concurrent code, and I am sure that they will come in handy sometime.

Identifying a coroutine in the logs

As you already know, you can create hundreds or thousands of coroutines, and those can be executed in one or more threads during their life cycle. Some of those coroutines will last for a long time, while others will be short-lived, maybe because they are tied to a temporary task. So, during debugging, it becomes a necessity to identify them.

Consider this simple application:

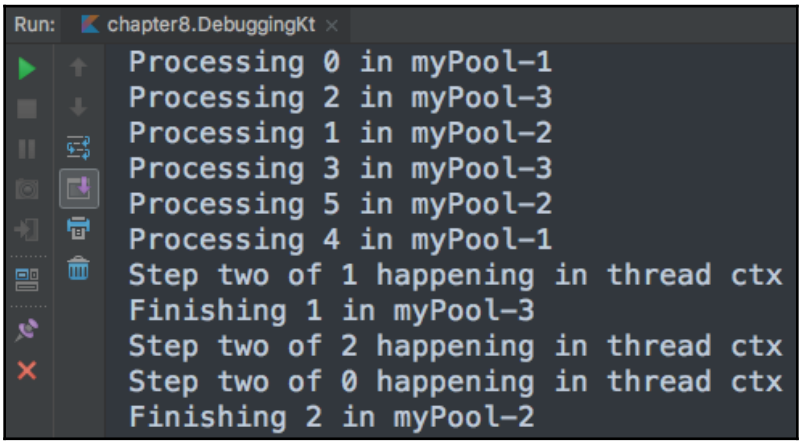
```
val pool = newFixedThreadPoolContext(3, "myPool")
val ctx = newSingleThreadContext("ctx")

val tasks = mutableListOf<Deferred<Unit>>()
for (i in 0..5) {
    val task = async(pool) {
        println("Processing $i in ${threadName()}")

        withContext(ctx) {
```

```
                println("Step two of $i happening in thread ${threadName()}")
            }

            println("Finishing $i in ${threadName()}")
        }

        tasks.add(task)
    }

    for (task in tasks) {
        task.await()
    }
```

The implementation of `threadName()` looks like this:

```
private fun threadName() = Thread.currentThread().name
```

This sample application has a pool of three threads, as well as a single thread context, `ctx`. Five coroutines are created; each of them will be initially placed in one of the threads of `pool`, but will soon be moved to the single thread of `ctx`, and then moved back to one thread in `pool`. On every context switch, we are printing `i`, as well as the name of the current thread:



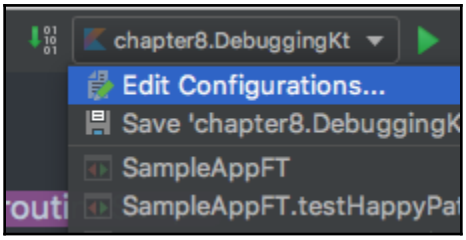
Thanks to the fact that we print `i`, we can make some sense of this and we can figure out which log entry corresponds to which specific element in our loop. But there are a couple of things that can be improved here:

- Once you have a certain amount of coroutines, it will be cumbersome to track each coroutine using the parameters that it received
- If the coroutine doesn't have something that can be used as an unique identifier, you won't be able to precisely identify which entries pertain to a specific one

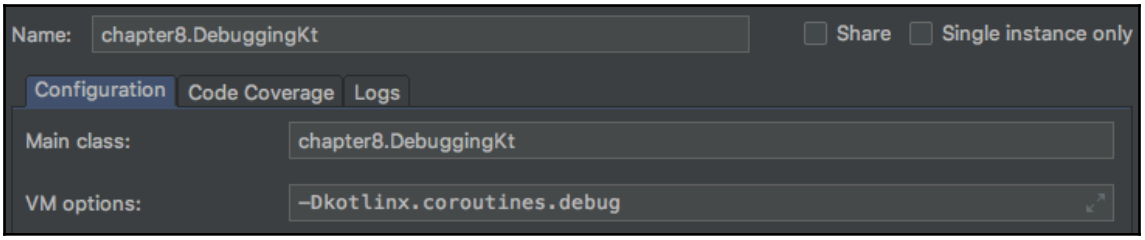
Using automatic naming

Currently, we have a group of coroutines that are both short-lived and created inside a loop, so we are not interested in giving them specific names. In cases like this, we can have Kotlin assign an automatic identifier to each of them.

To do so, we simply need to change the configuration that we use to execute our application. By passing the `-Dkotlinx.coroutines.debug` VM flag, Kotlin will append a unique identifier to every coroutine that is created when we request the name of the current thread. In order to add this flag, you need to click the dropdown in the selector of **Run/Debug Configuration**, shown as follows, and select **Edit Configurations**:

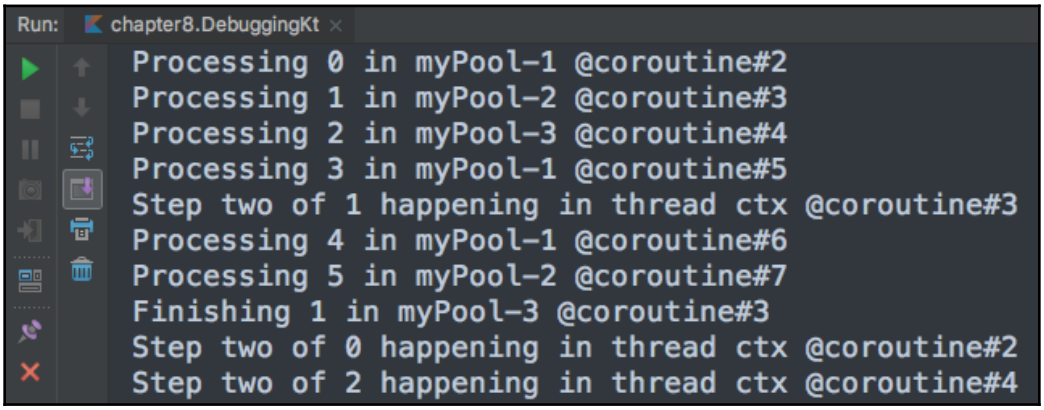


Once this is done, you can add the flag in the **VM Options** section:



I recommend you keep this flag on during development and during test execution, including any automatic test execution that you may have as part of your pipeline—for example, if you use a continuous integration workflow, make sure that the tests are run with the flag enabled. This will allow you to have meaningful logs for errors that may be difficult to replicate.

As mentioned, adding this flag will affect the value of `Thread.currentThread().name`; when obtained inside a coroutine it will now include a unique ID for it. Because our current code is already depending on that property, we don't need to change any code. Executing it now will print a log that will be easier to analyze:



Now you can `grep` the log to find all the entries related to a specific coroutine, as long as you keep printing the name of the current thread.

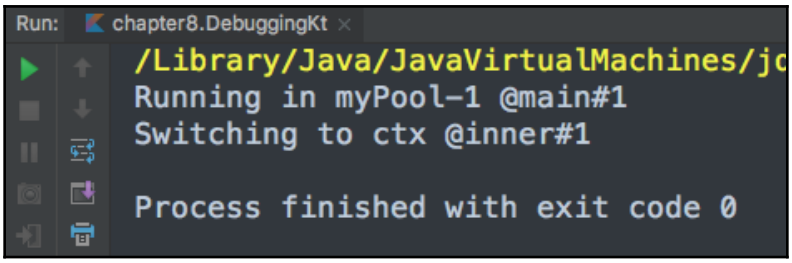
You can create or use a logger that always prints the name of the thread. That way, you will automatically have the identifier of the coroutine in your logs; just make sure to configure the flag.

Setting a specific name

In many cases, we have coroutines that are long-lasting, for example, if we have an actor or a producer and we want to specify which name should be used for them when the VM flag is passed. We can pass a parameter to the coroutine builders to specify the name that we want, and then we can easily find the entries in the log for that coroutine. Let's do a simple implementation of this:

```
val pool = newFixedThreadPoolContext(3, "myPool")
val ctx = newSingleThreadContext("ctx")

withContext(pool + CoroutineName("main")) {
    println("Running in ${threadName()}")

    withContext(ctx + CoroutineName("inner")) {
        println("Switching to ${threadName()}")
    }
}
```

This example uses `CoroutineName` to give a specific name to the coroutine. Because `CoroutineName` is a context element, it can be set using the plus operator. If we execute this code, `Thread.currentThread().name` will contain the name of the thread pool, the number of the thread in the pool—unless it is a single thread context—and the numeric identifier of the coroutine:



Remember that for the ID to be printed, the `-Dkotlinx.coroutines.debug` VM flag needs to be set as well. I advise you to always set a coroutine name for long-lasting coroutines like actors and producers.

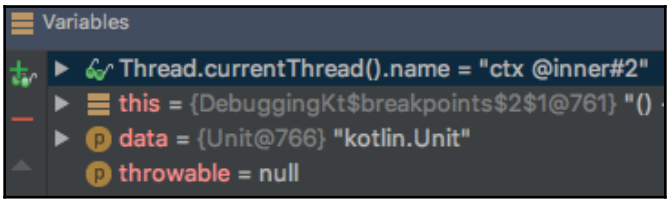
Identifying a coroutine in the debugger

When using the debugger in IntelliJ IDEA or Android Studio, we can take advantage of the `-Dkotlinx.coroutines.debug` flag as well. It can facilitate the general debugging of the app, and it can be used for setting breakpoints that will happen only for a given coroutine. Let's work with the following piece of code:

```
val pool = newFixedThreadPoolContext(3, "myPool")
val ctx = newSingleThreadContext("ctx")

for (i in 0..5) {
    async(pool + CoroutineName("main")) {
        val year = Calendar.getInstance().get(Calendar.YEAR)

        withContext(ctx + CoroutineName("inner")) {
            println(year)
        }
    }.await()
}
```

Adding a debugger watch

Because the debug flag affects the value of the name of the thread, you can add a watch in the IDE—to do so, press the **New Watch** button in the **Variables** section of the **Debug Tool** window. Once the execution is stopped at a breakpoint inside a coroutine, and if you have a watch for the name of the thread there, you will be able to see which coroutine is the one that is currently being executed:

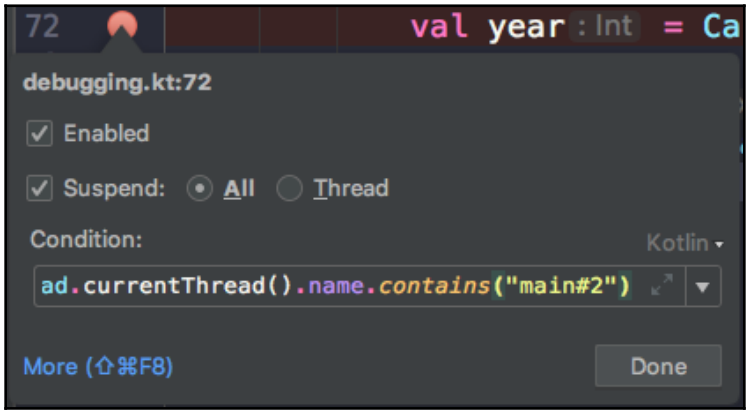


This example uses a watch variable set to `Thread.currentThread().name` and a breakpoint where the year is printed. We can see that in my case it was the second *inner* coroutine that was stopped.

Conditional breakpoint

Likewise, we can use the value of `Thread.currentThread().name` to configure a breakpoint that will only affect a given coroutine. This will be particularly useful in two scenarios: whenever you have a group of coroutines created in a loop but are interested in only one, or when you want to set the breakpoint in a piece of code that can be called from any part of the application but you are interested in a specific coroutine.

This can be easily achieved by doing a right-click in the breakpoint indicator and setting a condition based on the name of the thread:



In this case, we set the condition to `Thread.currentThread().name.contains("main#2")`, in the line where the year is obtained. So the breakpoint will only happen for the `@main#2` coroutine.

Get used to having the watch for the name of the thread always set and keep an eye on it when debugging. When you have more than a few coroutines, you need to keep the one you are debugging present. Likewise, consider setting conditional breakpoints so that you target a reduced scope when doing analysis.

Resiliency and stability

This was already mentioned as a bullet point in the section *More Advice on Tests*, but it's important enough for me to decide to include it as its own section—even if it's a small one.

Depending on the nature of the application that you are writing, it may be acceptable—maybe even expected—that your application will crash if certain conditions aren't met. In other scenarios, like for mobile apps, it's often expected that exceptions will be handled and the application will recover without side effects.

Resiliency is something that needs to be taken into account at the very start of the project. As we saw in previous chapters, you can easily set exception handlers for coroutines, and it doesn't require much work to validate if `Deferred` or `Job` ended with an exception, but if you try to add exception handling after a feature is *complete*, you will have to heavily modify your code, maybe even rewrite the whole thing. This is because resiliency has to come by design; you need to plan ahead and structure your code in a way that will meet the expected behavior.

If I am too late with this advice and you are already in a situation where you will need to significantly rewrite your application, I can still provide you with some wisdom: before you start making any changes, make sure that you have valuable tests for the things that currently work as expected—that will be your safety net. That way, you will be able to guarantee that a certain amount of functionality has been kept between changes.

Summary

This chapter was written around the idea of providing you with good practices that would make it easier for you to write and maintain concurrent applications written in Kotlin. Let's summarize the topics and practices discussed in this book so far, so that they stay fresh in your mind:

- There are two important principles when it comes to tests for concurrent code: *throw away assumptions*, meaning write tests for scenarios that should not happen to make sure that your app is resilient; and *focus on the forest, not the trees*, meaning that when testing for concurrency, you have to work on Functional Tests that allow you to replicate both the expected and the unexpected from a higher level.

- More advice on testing: always write tests for a bug when fixing it; analyze if a bug can be replicated in places where it has not been reported and write tests to cover those scenarios. Write tests that matter; consider the resiliency requirements as part of your design and test for resiliency; use branch analysis to make sure that your tests are heterogeneous enough; learn when to write Functional Tests and when to write unit tests; and always hide your dependencies behind an interface—this will help during testing by allowing you to provide mocks transparently. There are, of course, other well-known advantages of using interfaces, including allowing you to inject dependencies and facilitating new implementations of a functionality.
- Use the `-Dkotlinx.coroutines.debug` debug flag to have Kotlin append the ID of the coroutine to `Thread.currentThread().name` so that your logs are easier to analyze.
- Use the flag when debugging your code, and get used to having a variable watch with the thread name on your debugger so that you always have the coroutine and thread information at first glance.
- Always give a name to long-lasting coroutines; this will make debugging easier.
- Use conditional breakpoints both when testing ephemeral and long-lasting coroutines; that way you can focus your analysis on just the coroutines that you are interested in.
- The only way to guarantee that an application is stable is by having the right tests for it. It's up to the developer to write and maintain tests that meet the expected quality of the application and that really add value to the project.

By this point of the book, I hope to have helped you learn enough to start writing concurrent applications using Kotlin. But the journey hasn't ended yet. In order to completely be able to write effective concurrent code, it's important that you know the inner workings of Kotlin's concurrency. In the next chapter, we will explore how all of this works behind the scenes, and with that you will have a complete picture that should allow you to understand the concurrent code that you are writing at all levels.

9
The Internals of Concurrency in Kotlin

It's important that you have an idea of how suspending computations actually work. In this chapter, we will analyze how compiler transforms suspending functions into state machines, how the thread switching happens, and how exceptions are propagated. Some things that will be covered in this chapter are listed here:

- **Continuation Passing Style** (**CPS**) and how it's related to suspending computations
- Many different internal classes that are used when the coroutines are compiled into bytecode
- The Flow of interception of a coroutine, including how threads are switched
- Exception propagation with and without a `CoroutineExceptionHandler`

During the early sections of this chapter, we will be transforming a suspending function, imitating the work that the compiler does. Please notice that the code we write will not match the bytecode generated by the compiler exactly, but will be accurate enough to allow you to understand what is really happening.

This chapter doesn't include code examples for you to run in the IDE. The first half of the chapter uses pseudo Kotlin to represent what the compiler does, and the second half explains some of Kotlin's internal code.

Continuation Passing Style

The actual implementation of suspending computations is done using CPS. This paradigm is based on the premise of sending a continuation to a function that is invoked, so that upon completion, the function will invoke the continuation. You can think of continuations as callbacks: whenever a suspending computation invokes another, it will pass a continuation that should be called upon completion or error.

All the heavy lifting is done by the compiler, which transforms all the suspending computations so that they send and receive said continuations—as we will see, this means that the actual signatures of suspending functions are not the same as what we define. On top of that, the suspending computations are transformed into state machines that can save and restore their state, and execute one portion of their code at a time—so whenever they are resumed, they restore their state and continue execution where they left off.

Coupling CPS with state machines, the compiler creates computations that can be suspended while waiting for other computations to complete. Let's see this in more detail.

Continuations

It all starts with continuations, which can be considered the building blocks of suspending computations—after all, they are even in the name of the paradigm. The reason continuations are so important is that they are what allows a coroutine to be resumed. To make this clearer, here is the definition of the `Continuation` interface:

```
public interface Continuation<in T> {
    public val context: CoroutineContext
    public fun resume(value: T)
    public fun resumeWithException(exception: Throwable)
}
```

This interface is rather simple. Let's look at a quick overview of what it defines:

- The `CoroutineContext` that is to be used with this `Continuation`.
- A `resume()` function that takes a value `T` as a parameter. This value is the result of the operation that caused the suspension—so, if the function was suspended to call a function that returns an `Int`, value will be that integer.
- A `resumeWithException()` function that allows for propagation of an exception.

So, a continuation is pretty much an extended callback, one that also contains information about the context in which it should be called. As we will see later in the chapter, this context is an important part of the design, because it's what allows for each continuation to be executed in a specific thread or pool of threads—or with different configurations like exception handlers—but still remain sequential.

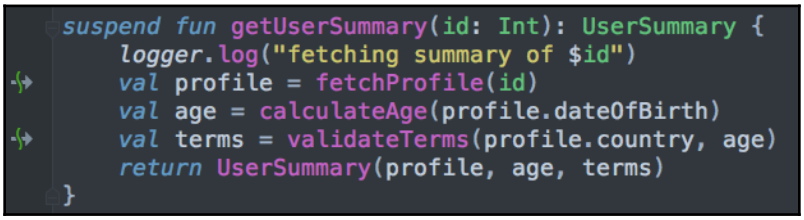
The suspend modifier

One specific goal of the Kotlin team was to make as few language changes as possible in order to support concurrency. Instead, the impact of supporting coroutines and concurrency was to be taken by the compiler, the standard library, and the coroutines library. So the only relevant change from a language perspective is the addition of the `suspend` modifier.

This modifier indicates to the compiler that the code in the given scope—function or lambda—will work using continuations. So whenever a suspending computation is compiled, its bytecode will be a big continuation. For example, let's consider this suspending function:

```
suspend fun getUserSummary(id: Int): UserSummary {
    logger.log("fetching summary of $id")
    val profile = fetchProfile(id) // suspending fun
    val age = calculateAge(profile.dateOfBirth)
    val terms = validateTerms(profile.country, age) // suspending fun
    return UserSummary(profile, age, terms)
}
```

What we are telling the compiler here is that the execution of `getUserSummary()` will happen through a `Continuation`. So the compiler will use a `Continuation` to control the execution of `getUserSummary()`. In this case, the function suspends twice: first when `fetchProfile()` is invoked, and later for the execution of `validateTerms()`. You can always count on IntelliJ IDEA and Android Studio to show you the suspension points of a function:



The arrows on the left represent a call to a suspending function, so they represent suspension points.

That means that our function's execution will happen in three steps. First, the function will be started and the log will be printed, then the invocation of `fetchProfile()` will cause the execution to suspend; once `fetchProfile()` has ended, our function will calculate the age of the user, and then the execution will suspend again for `validateTerms()` to be executed. The last step will occur once the terms are validated, when the function resumes one last time, and all the data from the previous steps is used to create the summary of the user.

State machine

Once the compiler has analyzed the code—the same way we did—it will transform the suspending function into a state machine. The idea is that the suspending function can behave as a continuation by executing different portions of its code each time it's resumed, based on its current state.

Labels

To better understand this, let's include a label where the execution starts, and also on each of the places were the execution can be resumed:

```
suspend fun getUserSummary(id: Int): UserSummary {
    // label 0 -> first execution
    logger.log("fetching summary of $id")
    val profile = fetchProfile(id)
    // label 1 -> resuming
    val age = calculateAge(profile.dateOfBirth)
    val terms = validateTerms(profile.country, age)
    // label 2 -> resuming
    return UserSummary(profile, age, terms)
}
```

Now, let's pretend that we can somehow receive the label that indicates which part of the code to execute. Then, we could write a `when` statement to separate the code to be executed:

```
when(label) {
    0 -> { // Label 0 -> first execution
        logger.log("fetching summary of $id")
        fetchProfile(id)
```

```
            return
        }
        1 -> { // label 1 -> resuming
            calculateAge(profile.dateOfBirth)
            validateTerms(profile.country, age)
            return
        }
        2 -> // label 2 -> resuming and terminating
            UserSummary(profile, age, terms)
    }
```

Notice that this snippet and many more in this chapter are a simplified representation of the generated bytecode, and some of them are pseudo Kotlin. I don't intend to transform the bytecode generated by the compiler into valid Kotlin, but rather to give you a fairly accurate idea of how it works.

Continuations

So, now that we have a bare function that can resume execution in a different point, we need to find a way to indicate the label of the function. Here is where `Continuation` takes the spotlight. Say that we implement a continuation at the very start of our function, one that will simply redirect any invocation to the callback back at the same function.

To be able to resume, we need to have at least the label. So we'll create an implementation of `CoroutineImpl`, which is an abstract implementation of `Continuation` that already includes a label property. The only abstract function in `CoroutineImpl` is `doResume()`, so that's the only thing we need to implement at this point:

```
suspend fun getUserSummary(id: Int): UserSummary {
    val sm = object : CoroutineImpl {
        override fun doResume(data: Any?, exception: Throwable?) {
            // TODO: Call getUserSummary to resume it
        }
    }
}
```

So, then, we just need to receive `Continuation<Any?>` as a parameter. That way, we can have `doResume()` forward the callback to `getUserSummary()`, as shown here:

```
suspend fun getUserSummary(id: Int,
        cont: Continuation<Any?>): UserSummary {

    val sm = object : CoroutineImpl {
        override fun doResume(data: Any?, exception: Throwable?) {
```

```
                getUserSummary(id, this)
            }
        }

        val state = sm as CoroutineImpl
        when(state.label) {
            ...
        }
    }
```

Notice that we aren't receiving `CoroutineImpl` directly in `getUserSummary()`. That is because we want to invoke `cont` at the completion of `getUserSummary()`, so that the caller of it gets resumed as well; so for compatibility, it makes more sense to receive a `Continuation<Any?>` in case the caller didn't use `CoroutineImpl`.

Callbacks

So, now that we have the ability to resume at a give point by using the label, we need to modify the other suspending functions that are called from `getUserSummary()`, so that they also receive `CoroutineImpl`. Let's say that we, as the compiler, have modified the `fetchProfile()` and `validateTerms()` functions so that they received a `Continuation<Any?>`—just like we did for `getUserSummary()`—and called `doResume()` upon the completion of their execution. Then we could invoke them like this:

```
when(state.label) {
    0 -> { // Label 0 -> first execution
        logger.log("fetching summary of $id")
        fetchProfile(id, sm)
        return
    }
    1 -> { // label 1 -> resuming
        calculateAge(profile.dateOfBirth)
        validateTerms(profile.country, age, sm)
        return
    }
    2 -> // label 2 -> resuming and terminating
        UserSummary(profile, age, terms)
}
```

By having both `fetchProfile()` and `validateTerms()` call the continuation that they receive whenever they finish their execution, we are having them call the continuation that we implemented in `getUserSummary()`, thus resuming its execution.

Incrementing the label

But, currently, we aren't actually incrementing the label, so the function will loop on label zero. That should be done right before the call to the other suspending functions:

```
when(state.label) {
    0 -> { // Label 0 -> first execution
        logger.log("fetching summary of $id")
        sm.label = 1
        fetchProfile(id, sm)
        return
    }
    1 -> { // label 1 -> resuming
        calculateAge(profile.dateOfBirth)
        sm.label = 2
        validateTerms(profile.country, age, sm)
        return
    }
    2 -> // label 2 -> resuming and terminating
        UserSummary(profile, age, terms)
}
```

Notice that, by default, the `label` is zero in `CoroutineImpl`. Also, notice that we set the value for the next piece of code, not for the current one.

Storing the result from the other operations

Currently, we aren't storing the result of the other suspending functions. To do so, we need to create a more complete implementation of the state machine, which can be easy if we use `CoroutineImpl` as the base. In this case, we could create this private class outside of the function:

```
private class GetUserSummarySM: CoroutineImpl {

    var value: Any? = null
    var exception: Throwable? = null
    var cont: Continuation<Any?>? = null
    val id: Int? = null
    var profile: Profile? = null
    var age: Int? = null
    var terms: Terms? = null

    override fun doResume(data: Any?, exception: Throwable?) {
        this.value = data
```

```
            this.exception = exception
            getUserSummary(id, this)
        }
    }
```

Here we have done the following:

- Mapped all the different variables that exist in the function to the class (`id`, `profile`, `age`, `terms`)
- Added one value to store the data returned by the caller in `doResume()`
- Added one value to store the exception that can be sent in `doResume()`
- Added one value to store the initial continuation that is sent when the execution of `getUserSummary()` is first started

So now we can update the function itself to both use this class and to set the properties as they become available:

```
val sm = cont as? GetUserSummarySM ?: GetUserSummarySM()

when(sm.label) {
    0 -> { // Label 0 -> first execution
        sm.cont = cont
        logger.log("fetching summary of $id")
        sm.label = 1
        fetchProfile(id, sm)
        return
    }
    1 -> { // label 1 -> resuming
        sm.profile = sm.value as Profile
        sm.age = calculateAge(sm.profile!!.dateOfBirth)
        sm.label = 2
        validateTerms(sm.profile!!.country, sm.age!!, sm)
        return
    }
    2 -> {// label 2 -> resuming and terminating
        sm.terms = sm.value as Terms
        UserSummary(sm.profile!!, sm.age!!, sm.terms!!)
    }
}
```

There are many important things in this code:

- We check whether `cont` is an instance of `GetUserSummarySM`; if that's the case, we use it as the state. If not, that means that it's the initial execution of the function, so a new one is created.

- As part of the first label, we store the current `cont` in the state machine. This will be used later to resume the caller of `getUserSummary()`.
- The second and third labels start by casting `sm.value` into the result of the last operation and storing it in the correct variable of the state machine.
- We use all the variables directly from the state machine.

Returning the result of the suspending computation

Now, our state machine is able to do almost everything it needs to do. The big missing part is to actually return the result of the operation somehow. Because this uses CPS, it isn't actually going to *return* the value in the classical sense; instead, it will use the first continuation it received as a callback, where it will send the result. This would be a complete implementation—minus error handling:

```
suspend fun getUserSummary(id: Int, cont: Continuation<Any?>) {

    val sm = cont as? GetUserSummarySM ?: GetUserSummarySM()

    when(sm.label) {
        0 -> { // Label 0 -> first execution
            sm.cont = cont
            logger.log("fetching summary of $id")
            sm.label = 1
            fetchProfile(id, sm)
            return
        }
        1 -> { // label 1 -> resuming
            sm.profile = sm.value as Profile
            sm.age = calculateAge(sm.profile!!.dateOfBirth)
            sm.label = 2
            validateTerms(sm.profile!!.country, sm.age!!, sm)
            return
        }
        2 -> {// label 2 -> resuming and terminating
            sm.terms = sm.value as Terms
            sm.cont!!.resume(UserSummary(sm.profile!!, sm.age!!,
sm.terms!!))
        }
    }
}
```

Notice that not only are we now using `sm.cont` as a callback to return the result, but we have also removed the return type from the function. But in reality, the signature of suspending functions indicates that they return `Any?`, and this happens because suspending computations can either return the value `COROUTINE_SUSPENDED` to indicate that the suspension did happen, or they can directly return a result if they didn't suspend. For example, imagine a suspending function that will only suspend in a given condition; if that condition doesn't occur, the function doesn't need to suspend and can instead return the result directly.

So in the actual bytecode, `getUserSummary()` will only suspend if the suspending functions it calls returns `COROUTINE_SUSPENDED` otherwise, it will cast the result of the function to the expected type— `Profile` and `Terms` in this case—and continue executing the next label. This guarantees that no unnecessary suspensions occur.

Context switching

One particularity of coroutines is that they can be resumed in a different context than the one they were initially started in. The `CoroutineContext`, as we have seen throughout the book, is not only about the thread or pool of threads to be used. It can contain other important configurations, such as exception handling.

As we saw at the beginning of the previous section, the `Continuation` interface defines that the `CoroutineContext` has to be stored inside the continuation. This guarantees that during execution, it will be possible to use that context when starting or resuming the continuation.

Let's take a look at how the context is set in order to allow thread switching.

Thread switching

There is one interface and two classes that we need to get familiar with in order to understand how coroutines are executed according to the context.

ContinuationInterceptor

As we discussed earlier in the book, the `CoroutineContext` behaves like a map where all the different `CoroutineContext.Element` are stored with their own unique key. One of those elements is defined by the `ContinuationInterceptor` interface. Let's take a look at it:

```
public interface ContinuationInterceptor : CoroutineContext.Element {
    companion object Key : CoroutineContext.Key<ContinuationInterceptor>
    fun <T> interceptContinuation(cont: Continuation<T>): Continuation<T>
}
```

Notice that apart from its key, it only defines a function that takes a continuation and returns another. The idea is for the implementation of `ContinuationInterceptor` to wrap the received continuation into another, one that is intercepted in order to guarantee that the correct thread is used.

CoroutineDispatcher

`CoroutineDispatcher` is an abstract implementation of `ContinuationInterceptor` that is used for the implementation of all the provided dispatchers, such as `CommonPool`, `Unconfined`, and `DefaultDispatcher`. Let's look at its code:

```
public abstract class CoroutineDispatcher :
        AbstractCoroutineContextElement(ContinuationInterceptor),
        ContinuationInterceptor {

    open fun isDispatchNeeded(context: CoroutineContext): Boolean = true
    abstract fun dispatch(context: CoroutineContext, block: Runnable)
    override fun <T> interceptContinuation(continuation: Continuation<T>):
        Continuation<T> = DispatchedContinuation(this, continuation)
    public operator fun plus(other: CoroutineDispatcher) = other
    override fun toString(): String = "$classSimpleName@$hexAddress"
}
```

Notice that this class provides an implementation for `interceptContinuation()` that returns a `DispatchedContinuation`. It also defines an abstract function, `dispatch()`, which takes the context and a `Runnable`. This `Runnable` is an expected declaration of an interface with a single `run()` function:

```
public expect interface Runnable {
    public fun run()
}
```

Expected declarations were added in Kotlin 1.2 in order to support multi-platform projects. Using the keyword `expect` you can define an API to be implemented on each platform—it supports classes, interfaces, functions, and more. So as we will see here, each platform can have its own implementation as long as it adjusts to the signature that is expected.

The actual implementation for the JVM is of course `java.lang.Runnable`:

```
public actual typealias Runnable = java.lang.Runnable
```

Notice that the implementation of an expected declaration is marked with the keyword `actual`. In this case, Kotlin's `Runnable` interface is mapped to Java's for execution in the JVM.

Before we move on to look at `DispatchedContinuation`, it's important that we see some of the implementations of the `dispatch` function. Bear in mind that this `dispatch()` function is the one that actually enforces the thread switching, if needed .

The `isDispatchNeeded()` function will be called before `dispatch()`. If it returns `false` then the framework will not call `dispatch()`; instead it will allow the resuming of the `Continuation` to happen in whichever thread the code is running—which, by default, will match the thread of the previous continuation in the chain.

CommonPool

First, it's worth mentioning that `isDispatchNeeded()` is not overridden in `CommonPool`, so it will always return `true`. This guarantees that the `dispatch()` function is always called when resuming a `Continuation`, and at this point the `Runnable` block will be executed using the pool. So let's see the implementation of `dispatch()` in `CommonPool`:

```
override fun dispatch(context: CoroutineContext, block: Runnable) =
    try {
        (pool ?: getOrCreatePoolSync())
            .execute(timeSource.trackTask(block))
    } catch (e: RejectedExecutionException) {
        timeSource.unTrackTask()
        DefaultExecutor.execute(block)
    }
```

Notice that in the JVM, `pool` is an instance of `java.util.concurrent.Executor`, and it's created by either using `java.util.concurrent.ForkJoinPool` or by using `newFixedThreadPool()` from `Executor`. The highlighted call to `execute()` is what is actually having the block run in the thread pool.

If there is a `SecurityManager` or there are errors trying to use `ForkJoinPool`, `newFixedThreadPool()` will be used.

Unconfined

Because `Unconfined` doesn't care to enforce a specific thread or pool of threads, it first overrides `isDispatchNeeded()` to indicate that it isn't needed:

```
override fun isDispatchNeeded(context: CoroutineContext): Boolean = false
```

And it throws an exception if you actually try to dispatch something with it:

```
override fun dispatch(context: CoroutineContext, block: Runnable) {
    throw UnsupportedOperationException()
}
```

Android's UI

Android's UI dispatcher has a rather interesting implementation. First's there's a `HandlerContext` class that takes an `android.os.Handler` and a `name`. And the dispatch function simply forwards the `Runnable` to the `post()` function of the handler:

```
public class HandlerContext(
    private val handler: Handler,
    private val name: String? = null
) : CoroutineDispatcher(), Delay {

    override fun dispatch(context: CoroutineContext, block: Runnable) {
        handler.post(block)
    }
}
```

Then, an instance of it called `UI` is created. It passes the main looper and the name `UI` to the constructor:

```
val UI = HandlerContext(Handler(Looper.getMainLooper()), "UI")
```

In practice, the only dispatcher that should override `isDispatchNeeded()` is `Unconfined`. All the other dispatchers should always enforce a thread or pool of threads.

DispatchedContinuation

As mentioned, the implementation of `interceptContinuation()` in `CoroutineDispatcher` is returning a `DispatchedContinuation`. This class is what pairs a `CoroutineDispatcher` with a `Continuation<T>`.

So let's take a look at it. First let's start with its constructor and the interfaces that it implements:

```
internal class DispatchedContinuation<in T>(
    @JvmField val dispatcher: CoroutineDispatcher,
    @JvmField val continuation: Continuation<T>
) : Continuation<T> by continuation, DispatchedTask<T> {
    ...
}
```

Notice that it takes a dispatcher and a continuation in the constructor, and it implements both `Continuation<T>` and `DispatchedTask<T>`. Its implementation of `resume()` and `resumeWithException()` is where the continuation and the dispatcher are connected together:

```
override fun resume(value: T) {
    val context = continuation.context
    if (dispatcher.isDispatchNeeded(context)) {
        _state = value
        resumeMode = MODE_ATOMIC_DEFAULT
        dispatcher.dispatch(context, this)
    } else
        resumeUndispatched(value)
}

override fun resumeWithException(exception: Throwable) {
    val context = continuation.context
    if (dispatcher.isDispatchNeeded(context)) {
        _state = CompletedExceptionally(exception)
        resumeMode = MODE_ATOMIC_DEFAULT
        dispatcher.dispatch(context, this)
    } else
        resumeUndispatchedWithException(exception)
}
```

As you can see, an important part of the implementation lies here. Whenever `resume()` or `resumeWithException()` are called, `DispatchedContinuation` will use the dispatcher if needed.

DispatchedTask

But there is still one gap. We know that `CoroutineDispatcher` defines `dispatch()` to take a `CoroutineContext` and a `Runnable`. Here is its signature:

```
public abstract fun dispatch(context: CoroutineContext, block: Runnable)
```

So how is it that `DispatchedContinuation` sends itself as the `Runnable`, as we just saw? Here is the snippet again:

```
dispatcher.dispatch(context, this)
```

As mentioned before, `DispatchedContinuation` also implements the `DispatchedTask` interface. This interface extends `Runnable` by adding a default implementation of `run()` that can trigger `resume()` and `resumeWithException()` in the continuation, as shown here:

```
public override fun run() {
    try {
        val delegate = delegate as DispatchedContinuation<T>
        val continuation = delegate.continuation
        val context = continuation.context
        val job = if (resumeMode.isCancellableMode) context[Job] else null
        val state = takeState()
        withCoroutineContext(context) {
            if (job != null && !job.isActive) {
                continuation.resumeWithException(
                    job.getCancellationException())
            } else {
                val exception = getExceptionalResult(state)
                if (exception != null) {
                    continuation.resumeWithException(exception)
                } else {
                    continuation.resume(getSuccessfulResult(state))
                }
            }
        }
    } catch (e: Throwable) {
        throw DispatchException("Unexpected exception running $this", e)
    }
}
```

Notice that `withCoroutineContext()` is called before calling `resume();` this guarantees that all elements that are part of the `CoroutineContext` are set before its execution.

Recap

This is a good time to recap how thread switching works. Now that we have all the parts explained, we can just put it in a few sentences.

The initial `Continuation` gets wrapped into a `DispatchedContinuation`; this is still a `Continuation`, but can dispatch using a `CoroutineDispatcher` if needed—which should be the case, except for `Unconfined`. The `CoroutineDispatcher` will use whichever executor fits its requirements, sending it a `DispatchedTask`, which is a `Runnable` that sets the correct context using `withCoroutineContext()` and invokes the `resume()` and `resumeWithException()` functions from the `DispatchedContinuation`.

So, the actual work of changing the thread happens in the `CoroutineDispatcher`, but is only possible thanks to the whole pipeline that allows it to intercept a continuation before its execution.

Exception handling

As we saw in previous chapters, it's possible to use a `CoroutineExceptionHandler` so that we can handle the exceptions that occur inside a coroutine. Let's take a look at how the exceptions are propagated from a coroutine.

The handleCoroutineException() function

Whenever an exception happens, it's redirected to the `handleCoroutineException()` function, which looks like this:

```
public fun handleCoroutineException(context: CoroutineContext, exception:
Throwable) {
    try {
        context[CoroutineExceptionHandler]?.let {
            it.handleException(context, exception)
            return
        }
        if (exception is CancellationException) return
        context[Job]?.cancel(exception)
```

```
            handleCoroutineExceptionImpl(context, exception)
        } catch (handlerException: Throwable) {
            if (handlerException === exception) throw exception
            throw RuntimeException(
                "Exception while trying to handle coroutine exception",
                exception).apply {
                    addSuppressedThrowable(handlerException)
                }
        }
    }
```

Let's break down the different parts of this code and analyze what it does.

CoroutineExceptionHandler

The first section of `handleCoroutineException()` looks for a `CoroutineExceptionHandler` in the `CoroutineContext`. If found, the `handleException()` function will be called, passing both the context and the exception:

```
context[CoroutineExceptionHandler]?.let {
    it.handleException(context, exception)
    return
}
```

Notice that the execution of `handleCoroutineException()` will be stopped after `handleException()` is called because of the `return`.

CancellationException

Because `CancellationException` is used to cancel a coroutine, these are ignored. This is done by calling `return`:

```
if (exception is CancellationException) return
```

Cancelling the job

The next step is to cancel the execution of the job. If the `Job` exists in the `CoroutineContext`, its `cancel()` function is invoked:

```
context[Job]?.cancel(exception)
```

Platform specific logic

Finally, the `handleCoroutineExceptionImpl()` function is invoked. This function is an expected declaration, so the implementation may be different for each platform. Here is the declaration of the function:

```
internal expect fun handleCoroutineExceptionImpl(
        context: CoroutineContext,
        exception: Throwable
)
```

JVM

The implementation for the JVM looks like this:

```
internal actual fun handleCoroutineExceptionImpl(
        context: CoroutineContext,
        exception: Throwable) {
    ServiceLoader.load(CoroutineExceptionHandler::class.java)
        .forEach { handler ->
            handler.handleException(context, exception)
        }
    val currentThread = Thread.currentThread()
    currentThread.uncaughtExceptionHandler
        .uncaughtException(currentThread, exception)
}
```

It looks for exception handlers using the `ServiceLoader` and forwards the exception to all the ones it can find. It also forwards the exception to the handler of the current thread.

JavaScript

In JavaScript, it simply logs the exception:

```
internal actual fun handleCoroutineExceptionImpl(context: CoroutineContext,
        exception: Throwable) {
    console.error(exception)
}
```

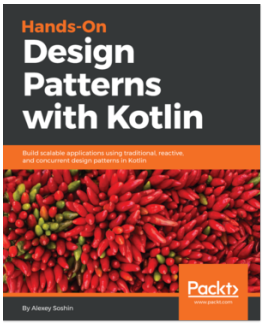
Summary

In this chapter, we covered some of the most important details of how coroutines actually work. We put ourselves in the place of the compiler and transformed a suspending function into a state machine. We also analyzed big chunks of Kotlin's code to understand how a coroutine is intercepted in order to change (or not change) its thread, as well as to understand how exceptions are propagated. Here are some things you should remember:

- The suspending computations are transformed into state machines, and via the use of CPS they become callbacks for other suspending functions.
- The Kotlin language itself had little changes made to it in order to support coroutines. Most of the work is done by the compiler and the coroutines library.
- Continuations are wrapped into `DispatchedContinuation`s at runtime. This allows the `CoroutineDispatcher` to intercept the coroutine, both when started and when resumed. At this point, the thread will be enforced—except for `UNCONFINED`.
- If the `CoroutineContext` doesn't have a `CoroutineExceptionHandler` and the uncaught exception is not `CancellationException`, the framework will cancel the `Job`—if any—and allow platform-specific code to handle the exception.
- In JavaScript, the platform-specific handling will simply log the exception.
- In the JVM, it will try to find `CoroutineExceptionHandler`s using `ServiceLoader`. All the handlers found will receive a notification of the exception. After this, the JVM implementation of `handleCoroutineExceptionImpl()` will forward the exception to the `uncaughtExceptionHandler` of the current thread.

Other Books You May Enjoy

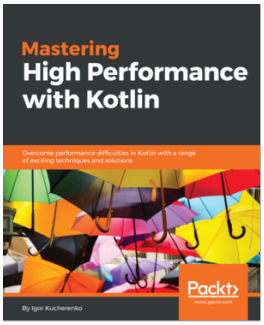
If you enjoyed this book, you may be interested in these other books by Packt:



Hands-on Design Patterns with Kotlin
Alexey Soshin

ISBN: 9781788998017

- Get to grips with Kotlin principles, including its strengths and weaknesses
- Understand classical design patterns in Kotlin
- Explore functional programming using built-in features of Kotlin
- Solve real-world problems using reactive and concurrent design patterns
- Use threads and coroutines to simplify concurrent code flow
- Understand antipatterns to write clean Kotlin code, avoiding common pitfalls
- Learn about the design considerations necessary while choosing between architectures



Mastering High Performance with Kotlin
Igor Kucherenko

ISBN: 9781788996648

- Understand the importance of high performance
- Learn performance metrics
- Learn popular design patterns currently being used in Kotlin
- Understand how to apply modern Kotlin features to data processing
- Learn how to use profling tools
- Discover how to read bytecode
- Learn to perform memory optimizations
- Uncover approaches to the multithreading environment

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

D

R

S

T

U

V